Fundamental Data Structures and Algorithms 05 - Trees

Unit 3: Basic Data Structures (continued)

3.6 Trees

One of the most important nonlinear data structures is the **tree**. The *tree* basically resembles a family tree. As the relationships in a tree are hierarchical, the *tree* also uses the similar terminologies such as *parent* and *child*.

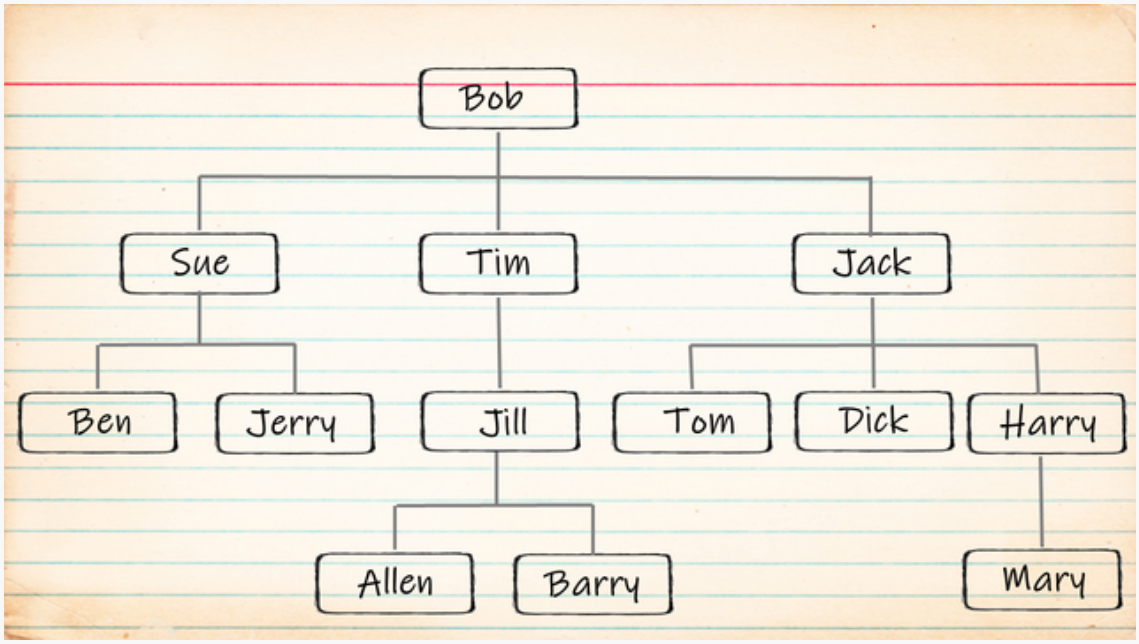

Fig 3.8. A family tree showing the descendants of Bob

Tree structures allow us to implement a host of algorithms much faster than when using linear data structures such as arrays or linked list. As a result, *trees* have become ubiquitous data structures that are used in file systems, graphical user interfaces (GUI), databases and even websites.

3.6.1 Tree Structures and Terminologies

A *tree* is an abstract data type that stores elements hierarchically. With the exception of the top node, each element in a *tree* has a ***parent*** element with zero or more ***children*** nodes.

However, we will be mainly looking at tree structures that have at most 2 children nodes. Each child is labeled as being either a ***left child*** or a ***right child***. Such trees are known as ***binary trees***.

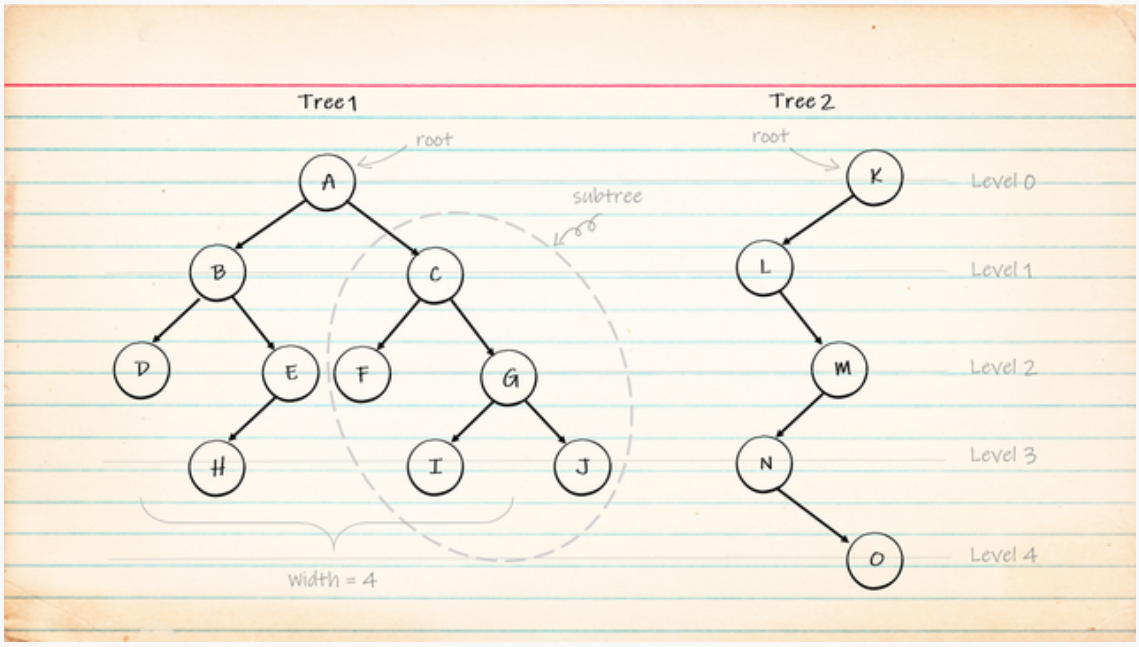

Fig 3.9. Tree structures and terminologies

The top element is also known as the ***root*** node of the tree (think of an actual tree but inverted). Typically, the root node is used as the access point into the *tree* structure. In Fig 3.9, node A and K are the root nodes.

Every node can be the root of its own ***subtree***, which consists of a subset of nodes and edges of the larger tree. The subtree rooted at a left or right child of an internal node is called a left subtree or right subtree. This property makes trees recursive in nature. Fig 3.9 shows that C, F, G, I and J is a subtree where C is the root of the subtree.

A node is ***external***, if it has no children. An external node is also known as ***leaf*** node. On the other hand, a node is ***internal*** if it has one or more children. In Fig 3.9, nodes D, H, F, I, J and O are all leaf nodes, while the rest are internal nodes.

An ***edge*** of a tree is a pair of nodes such that the pair have a parent-child relationship. A ***path*** is a sequence of nodes such that any two consecutive nodes in the sequence form an edge. In Fig 3.9, the nodes K, L, M, N and O form a *path* from node K to node O.

The nodes in a tree are organized into ***levels*** with the root node at level 0, and its children are at level 1, and so on. The ***depth*** of a node is its distance from the root, with distance being the number of levels that separate the two. In other words, a node's depth corresponds to the level it occupies. For example, node C has a depth of 1 while node N has a depth of 3.

The ***height*** of a tree is the number of levels in the tree. Tree 1 has a height of 3 while tree 2 has a height of 4. The ***width*** of a tree is the number of nodes on the level containing the most nodes. Tree 1 has a width of 4 while Tree 2 has a width of only 1. Finally, the ***size*** of a tree is simply the number of nodes within the tree. Tree 1 has a size of 10 while Tree 2 has a size of 5. An empty tree has a height of 0, a width of 0, and its size is 0.

There are many types of binary trees:

- A ***full / proper*** binary tree is when each node has either 0 or 2 children.
- A ***complete*** binary tree is a binary tree where every level, except possibly the last, is completely filled and all nodes in the last level are as far left as possible.
- A ***perfect*** binary tree is a full binary tree in which all leaf nodes are at the same level. The perfect tree has all possible node slots filled from top to bottom with no gaps

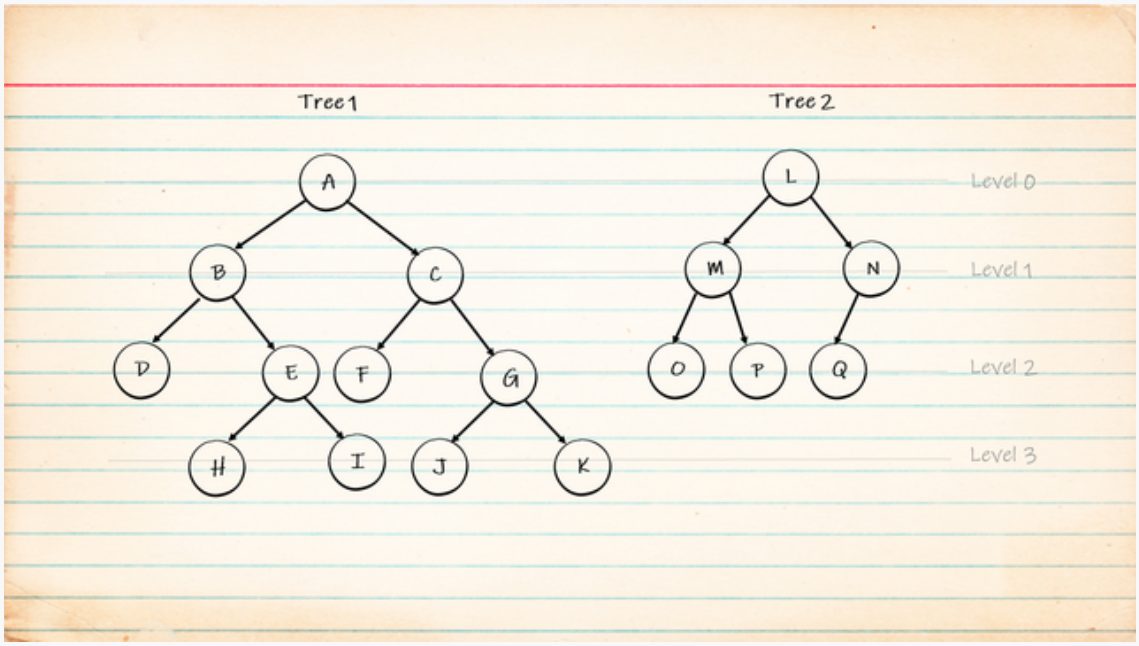

Fig 3.10. Full and Complete Binary Trees

In Fig 3.10, Tree 1 is a full binary tree but it is not proper as the the leaf nodes are not left aligned. Tree 2 is an not a full binary tree but it is a complete binary tree.

3.6.2 Binary Search Tree

A ***binary search tree*** (BST) is an ordered binary tree with the following properties:

1. Every node has at most two children.
2. Most importantly, the left child precedes a right child in the order of children of a node. This also means that a binary search tree should never contain duplicate nodes.

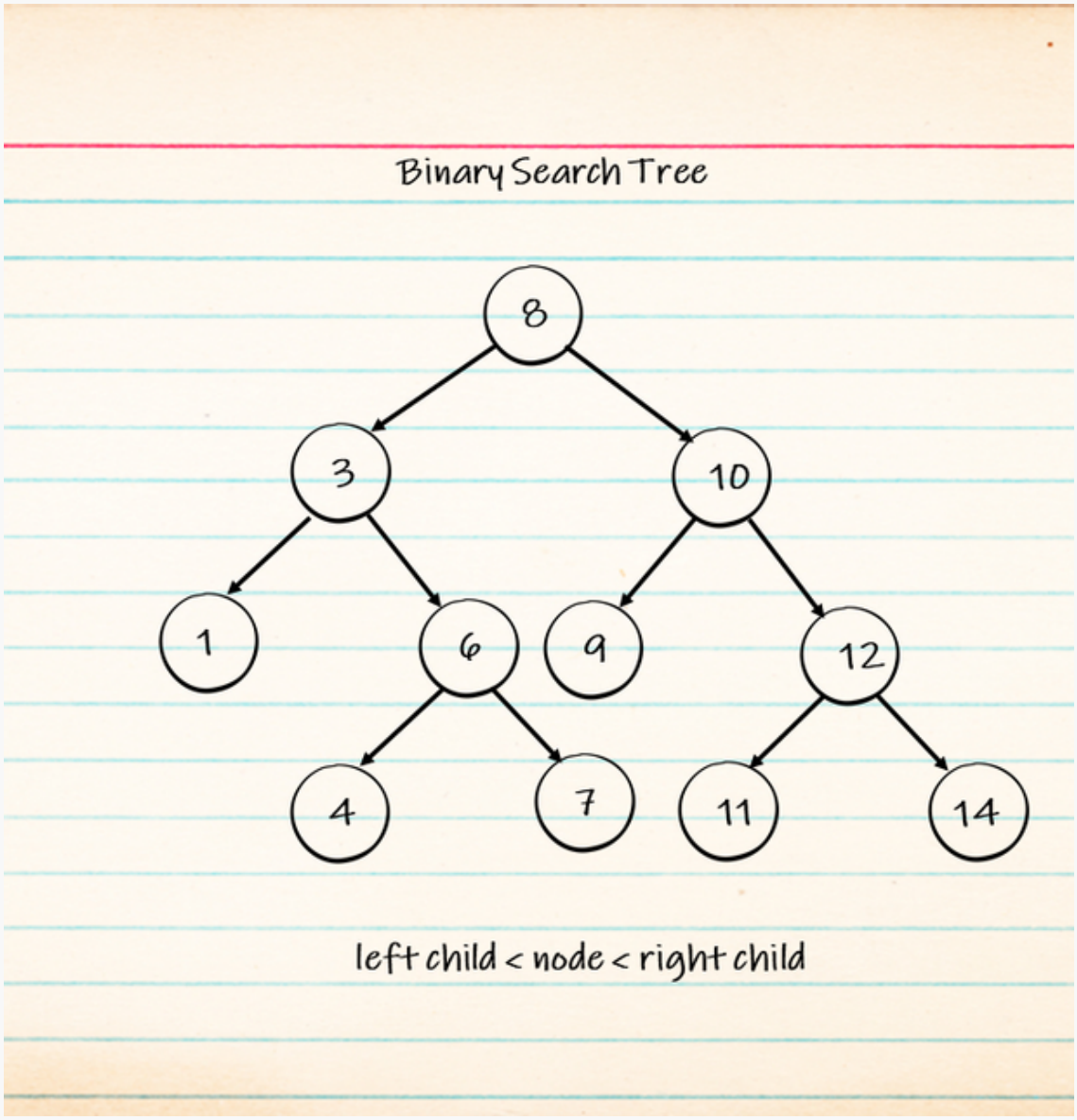

Fig 3.11. An example of a Binary Search Tree

3.6.2.1 Binary Search Tree Implementation

Implementing a BST is similar to that in linked list but instead of next we implement *left* and *right* instead to represent the 2 children.

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

    # A utility function to insert a new node with the given key
def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if root.val == key:
            return root
        elif root.val < key:
            root.right = insert(root.right, key)
        else:
            root.left = insert(root.left, key)
    return root

# A function to do inorder tree traversal
def printInorder(root):
    if root:
        # First recur on left child
        printInorder(root.left)

        # then print the data of node
        print(root.val),

        # now recur on right child
        printInorder(root.right)

# A function to do postorder tree traversal
def printPostorder(root):
    if root:
        # First recur on left child
        printPostorder(root.left)

        # the recur on right child
        printPostorder(root.right)

        # now print the data of node
        print(root.val),


# A function to do preorder tree traversal
def printPreorder(root):
    if root:
        # First print the data of node
        print(root.val),

        # Then recur on left child
        printPreorder(root.left)
```

```
        # Finally recur on right child
        printPreorder(root.right)

# A utility function to search a given key in BST
def search(root,key):

    # Base Cases: root is null
    if root is None:
        print(f"Element {key} does not exist")

    # key is present at root
    elif root.val == key:
        print(f"Element {root.val} found in the Tree")
        return True

    # Key is greater than root's key
    elif root.val < key:
        return search(root.right,key)

    # Key is smaller than root's key
    elif root.val > key:
        return search(root.left,key)

    #key is not present
    else:
        print("Element not found in Tree")
        return False
```

To create the same structure as shown in Fig 3.11, we can do the following:

```
r = Node(8)
insert(r, 3)
insert(r, 10)
insert(r, 1)
insert(r, 6)
insert(r, 9)
insert(r, 12)
insert(r, 4)
insert(r, 7)
insert(r, 11)
insert(r, 14)
```

To visualize this algorithm, you can visit here.

3.6.2.2 Tree Traversal (Inorder, Preorder and Postorder)

A *traversal* of a tree is a systematic way of accessing, or "visiting", all the nodes of the tree.

To verify we can print the BST by invoking `printInOrder()` method. This is one of the three tree traversals methods. *Inorder* traversals of a BST will always give nodes in a sorted (ascending) order:

```
1
3
4
6
7
8
9
10
11
12
14
```

You might be wondering, what the other two methods, `postOrder` and `preOrder` are for. Although it may just seem like it is just to printing the nodes in different orders (based on the difference at which `print(self.data)` is called), there are several use cases for the different traversal method. You can think of the print function as a placeholder for other things i.e. it can be replaced by other functions depending on the application.

One of the use cases for the *preorder* traversal is that it is typically used to create a copy of the BST. If we were to invoke `printPreOrder()`, you will see that the order at which the nodes are printed can be used to create a copy of the BST:

```
8
3
1
6
4
7
10
9
12
11
14
```

Another interesting use case of a *preorder* traversal is used in this notes - in particular, accessing chapters or sub-chapter of a book or document (refer to Fig 3.12)

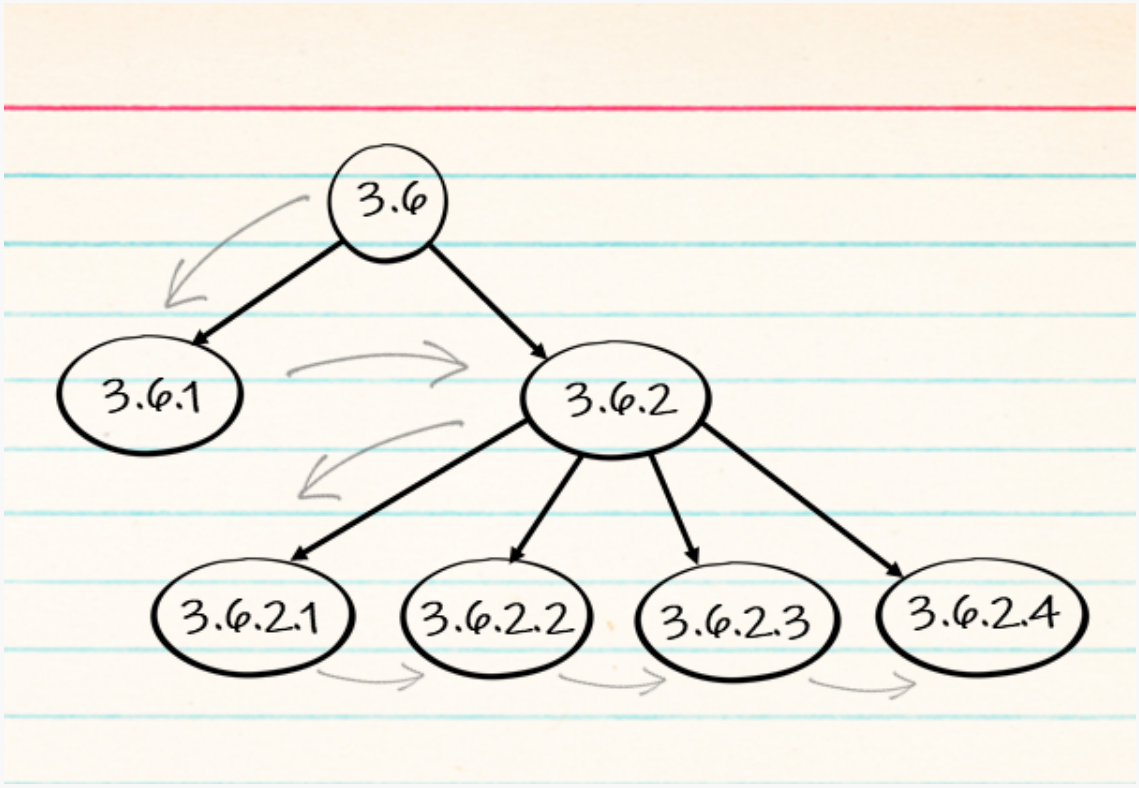

Fig 3.12. Preorder traversal on tree structure used to access chapters or sub-chapters of a book or document

On the other hand, *postorder* traversal is typically used to delete the tree:

```
1
4
7
6
3
9
11
14
12
10
8
```

If we follow the nodes listed above we can see that the nodes being called, are all leaf nodes, which can be deleted without affecting the other parts of the tree. However, this is form of deletion is only applicable if the nodes we intend to delete are leaf nodes. To remove interior nodes, we have to consider a different approach. This will be left as an exercise for readers.

3.6.2.3 Search, Minimum and Maximum Values

As the name suggests, binary search trees are particularly useful for searching items. Searching in a BST follows a very simple rule:

1. Start from the root node.
2. If target node < root node, search the left subtree.
3. If target node > root node, search the right subtree.

In fact, this implementation can be seen in the `insert()` method.

Let's say we need to search for node 11 in Fig 3.11. We simply have to compare the node we are visiting with node 11 - moving to the right subtree if node 11 is greater than the visited node or left subtree if is lesser than the visited node. Using this simple algorithm, we can see that we will visit the root node 8, then node 10, and then node 12 before finally reaching the targeted node 11 .

Another case is if we want to search for say node 5. Starting from the root node, we move the left child (since 5 is smaller than 8) which is node 3. We then repeat the comparison process by visiting node 6 and finally node 4. However, since node 5 is greater than node 4, we expect node 5 to be the right child of node 4 but we find that node 4 is a leaf node thus we can return a *node not found* message.

Based on the same algorithm, we can easily find the minimum and maximum values contained in the BST. To reach the minimum value, all we have to do is just recursively visit the left subtree until a leaf node is reached and thus we can ignore step 3 entirely. The maximum value can also be found in a similar fashion.

3.6.2.4 Efficiency of Binary Search Trees

This evaluation assumes the tree contains n nodes. We begin with the *search* method. In searching for a target node, the function starts at the root node and works its way down into the tree until either the node is located or a null link is encountered. The worst case time for the search operation depends on the number of nodes n that have to be examined.

Earlier, we saw that the worst case time of a tree traversal was linear since it visited every node in the tree. When a null child link is encountered, the tree traversal backtracks to follow the other branches. During a search, however, the function never backtracks; it only moves down the tree, following one branch or the other at each node. Note that the recursive function does unwind in order to return to the location in the program where it was first invoked, but during the unwinding other branches are not examined. The search follows a single path from the root down to the target node or to the end of the search at which the target node is not found.

The worst case occurs when the longest path in the tree is followed in search of the target and the longest path is based on the height of the tree. Binary trees come in many shapes and sizes and their heights can vary. But, as we saw in the
previous chapter, a tree of size n can have a minimum height of roughly $\log n$ when the tree is complete and a maximum height of n when there is one node per level. If we have no knowledge about the shape of the tree and its height, we have to assume the worst and in this case that would be a tree of height n. Thus, the time required to find a node in a binary search tree is O(n) in the worst case.

Searching for the minimum (or maximum) value in a binary search tree is also a linear time operation. Even though it does not compare nodes, it does have to navigate through the tree by always taking the left branch starting from the root. In the worst case, there will be one node per level with each node linked to its parent via the left (or right) child link.

The `insert()` method, which implements the algorithm for inserting a new element into the tree, performs a search to find where the new node belongs in the tree. We know from our earlier analysis the search operation requires linear time in the worst case. How much work is done after locating the parent of the node that will contain the new element? The only work done is the creation of a new node and returning its link to the parent, which can be done in constant time. Thus, the insertion operation requires O(n) time in the worst case.

Therefore, each operation in a BST has a time complexity of O(n).

3.6.3 AVL Tree

The binary search tree provides a convenient structure for storing and searching data collections. The efficiency of the search, insertion, and deletion operations depend on the height of the tree. In the best case, a binary tree of size n has a
height of $\log n$, but in the worst case, there is one node per level, resulting in a height of n. Thus, it would be to our advantage to try to build a binary search tree that has height $\log n$.

If we were constructing the tree from the complete set of search nodes, this would be easy to accomplish. The nodes can be sorted in ascending order and then using a technique similar to that employed with the linked list version of the merge sort, the interior nodes can be easily identified. But this requires knowing all of the nodes up front, which is seldom the case in real applications where nodes are routinely being added and removed. We could rebuild the binary search tree each time a new node is added or an existing one is removed. But the time to accomplish this would be extreme in comparison to using a simple brute-force search on one of the sequential list structures. What we need is a way to maintain the optimal tree height in real time, as the entries in the tree change.

The AVL tree, which was invented by G. M. Adel'son-Velskii and Y. M. Landis in 1962, improves on the binary search tree by always guaranteeing the tree is height *balanced*, which allows for more efficient operations. A binary tree
is balanced if the heights of the left and right subtrees of every node differ by at most 1.

With each node in an AVL tree, we associate a *balance factor*, which indicates the height difference between the left and right sub-trees . The *balance factor* can be one of three states:

- left high: When the left subtree is higher than the right subtree.
- equal high: When the two subtrees have equal height.
- right high: When the right subtree is higher than the left subtree.

The balance factors of the tree nodes in our illustrations are indicated by symbols: $>$ for a left high state, $=$ for the equal high state, and $<$ for a right high state. When a node is out of balance, we will use either $<<$ or $>>$ to indicate which subtree is higher.

The search and traversal operations are the same with an AVL tree as with a binary search tree. The insertion and deletion operations have to be modified in order to maintain the balance property of the tree as new nodes are inserted and existing ones removed. By maintaining a balanced tree, we ensure its height never exceeds O($\log n$) time operations even in the worst case.

3.6.3.1 Insertions

Inserting a node into an AVL tree begins with the same process used with a binary search tree. We search for the new node in the tree and add a new node at the child link where we fall off the tree. When a new node is inserted into an AVL tree, the balance property of the tree must be maintained. If the insertion of the new node causes any of the subtrees to become unbalanced, they will have to be rebalanced.

Some insertions are simpler than others. For example, suppose we want to add node 120 to the sample AVL tree from Fig 3.13(a). Following the insertion operation of the binary search tree, the new node will be inserted as the right child
of node 100, as illustrated in Fig 3.13(b).

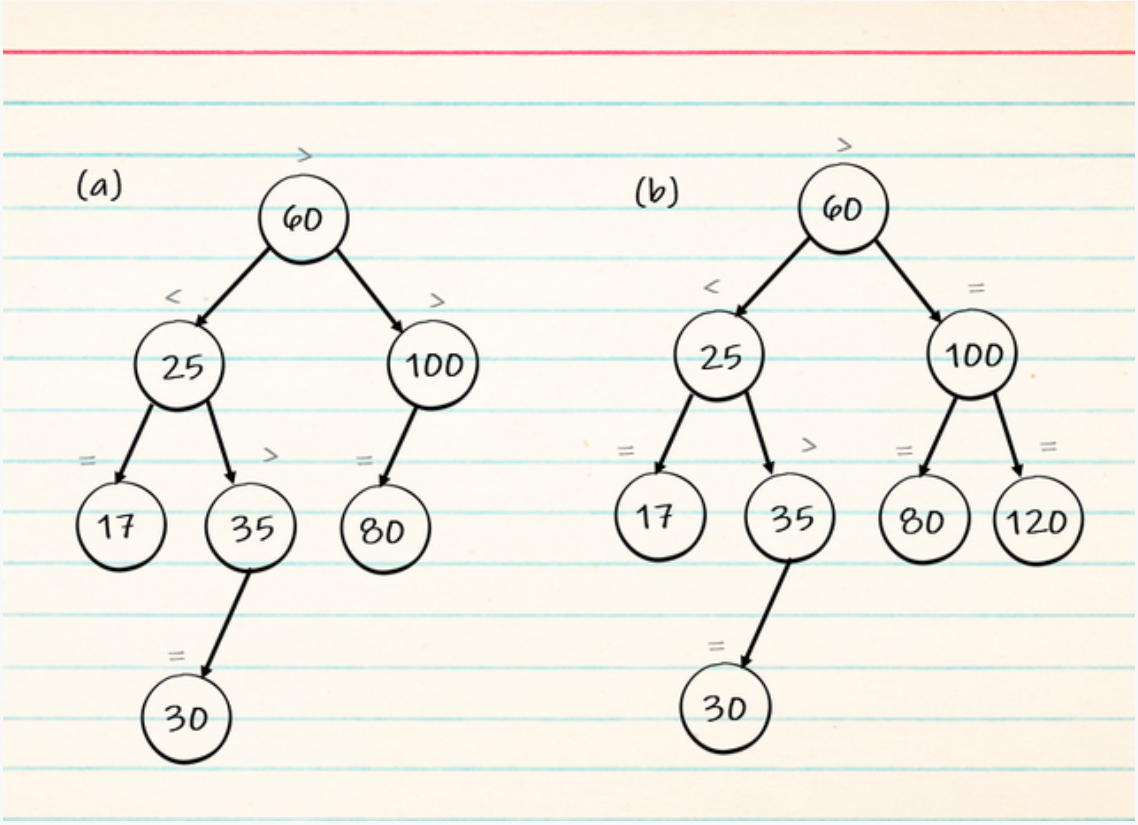

Fig 3.13. A simple insertion into an AVL tree: (a) original; and (b) with node 120 inserted.

The tree remains balanced since the insertion does not change the height of any subtree, but it does cause a change in the balance factors. After the node is inserted, the balance factors have to be adjusted in order to determine if any subtree is out of balance. There is a limited set of nodes that can be affected when a new node is added. This set is limited to the nodes along the path to the insertion point. Fig 3.13(b) also shows the new balance factors after node 120 is added.

What happens if we add node 28 to the AVL? The new node is inserted as the left child of node 30, as illustrated in Fig 3.14(b). When the balance factors are recalculated, as in Fig 3.14(c), we can see all of the subtrees along the path that are above node 30 are now out of balance, which violates the AVL balance property. For this example, we can correct the imbalance by *rotating* the subtree rooted at node 35.

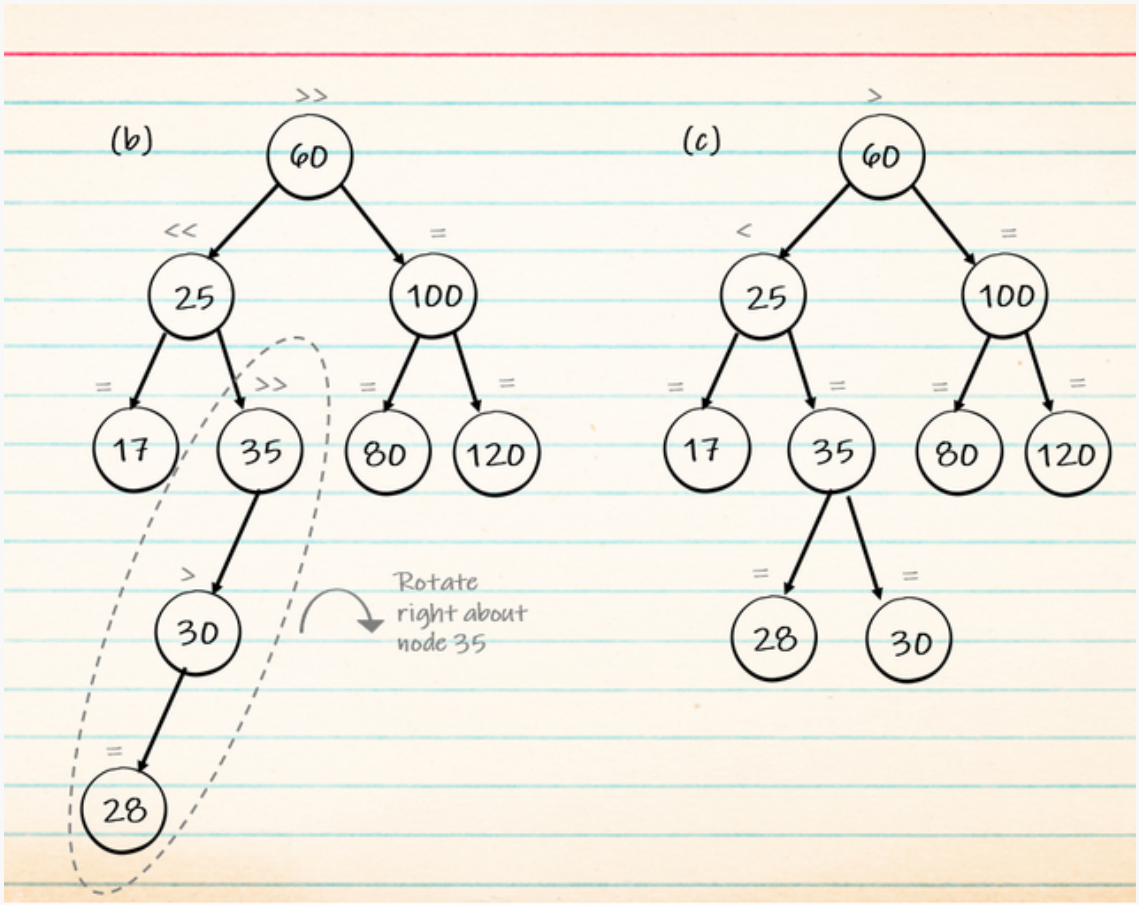

Fig 3.14. (b) Unbalanced tree after inserting node 28; and (c) Balanced tree after rotating at node 35

3.6.3.2 Rotations

Tree rotations are best understood as follows: Any binary search tree containing at least two nodes can clearly be drawn in one of the two forms:

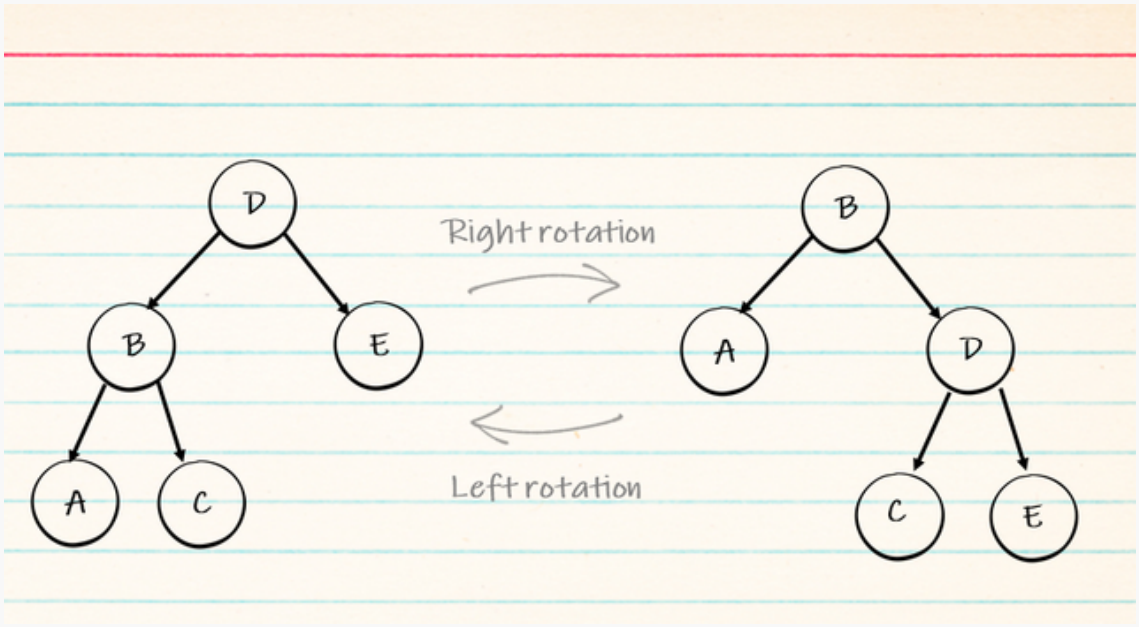

Fig 3.15. Left and right tree rotations.

where B and D are the required two nodes to be rotated, and A, C and E are binary search sub-trees (any of which may be empty). The two forms are related by left and right tree rotations which clearly preserve the binary search tree property. In this case, any nodes in sub-tree A would be shifted up the tree by a right rotation, and any nodes in sub-tree E would be shifted up the tree by a left rotation.

Multiple subtrees can become unbalanced after inserting a new node, all of which have roots along the insertion path. But only one will have to be rebalanced: the one deepest in the tree and closest to the new node. After inserting the node, the balance factors are adjusted during the unwinding of the recursion. The first subtree encountered that is out of balance has to be rebalanced. The root node of this subtree is known as the *pivot node*.

An AVL subtree is rebalanced by performing a rotation around the pivot node. This involves rearranging the links of the pivot node, its children, and possibly one of its grandchildren. The actual modifications depend on which descendant's subtree of the pivot node the new node was inserted into and the balance factors. There are four possible cases:

- Case 1: This case, as illustrated in Fig 3.16, occurs when the balance factor of the pivot node (P) is left high before the insertion and the new node is inserted into the left child (C) of the pivot node. To rebalance the subtree, the pivot node has to be rotated right over its left child. The rotation is accomplished by changing the links such that P becomes the right child of C and the right child of C becomes the left child of P.

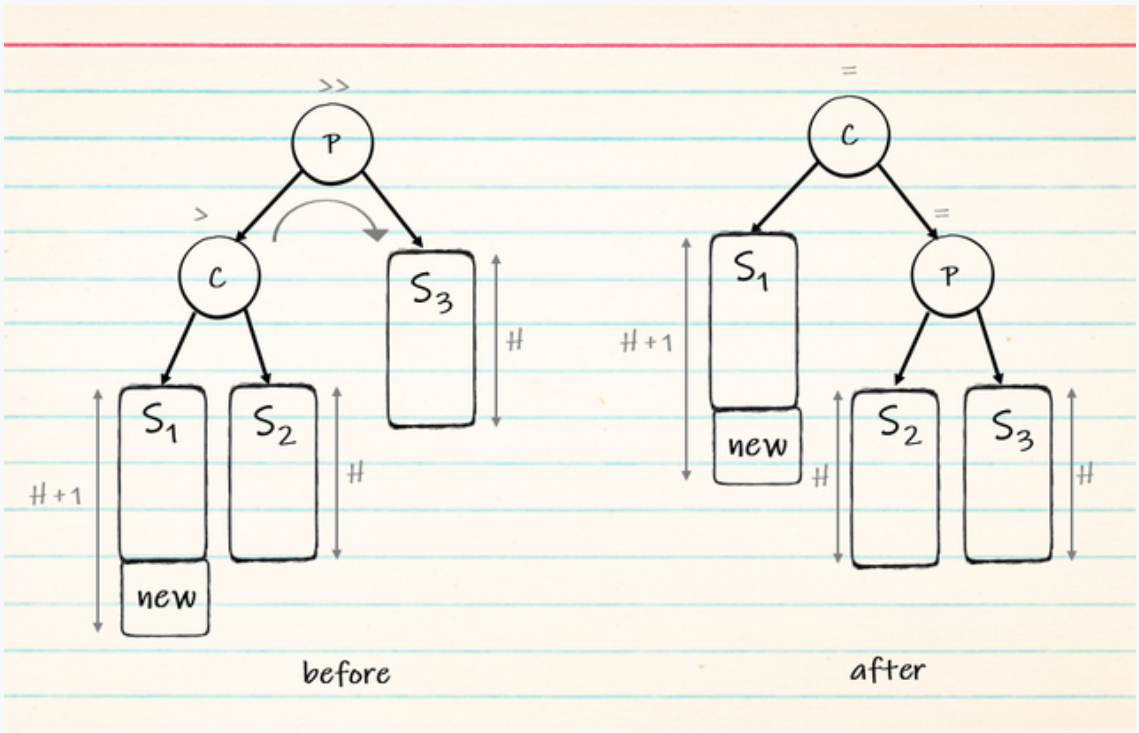

Fig 3.16. Case 1: a right rotation of the pivot node over its left child.

- Case 2: This case involves three nodes: the pivot (P), the left child of the pivot (C), and the right grandchild (G) of P. For this case to occur, the balance factor of the pivot is left high before the insertion and the new node is inserted into either the right subtree of C. This case, which is illustrated in Fig 3.17, requires two rotations. Node C has to be rotated left and the pivot node has to be rotated right. The link modifications required to accomplish this rotation include setting the right child of G as the new left child of the pivot node, changing the left child of G to become the right child of C, and setting C to be the new left child of G.

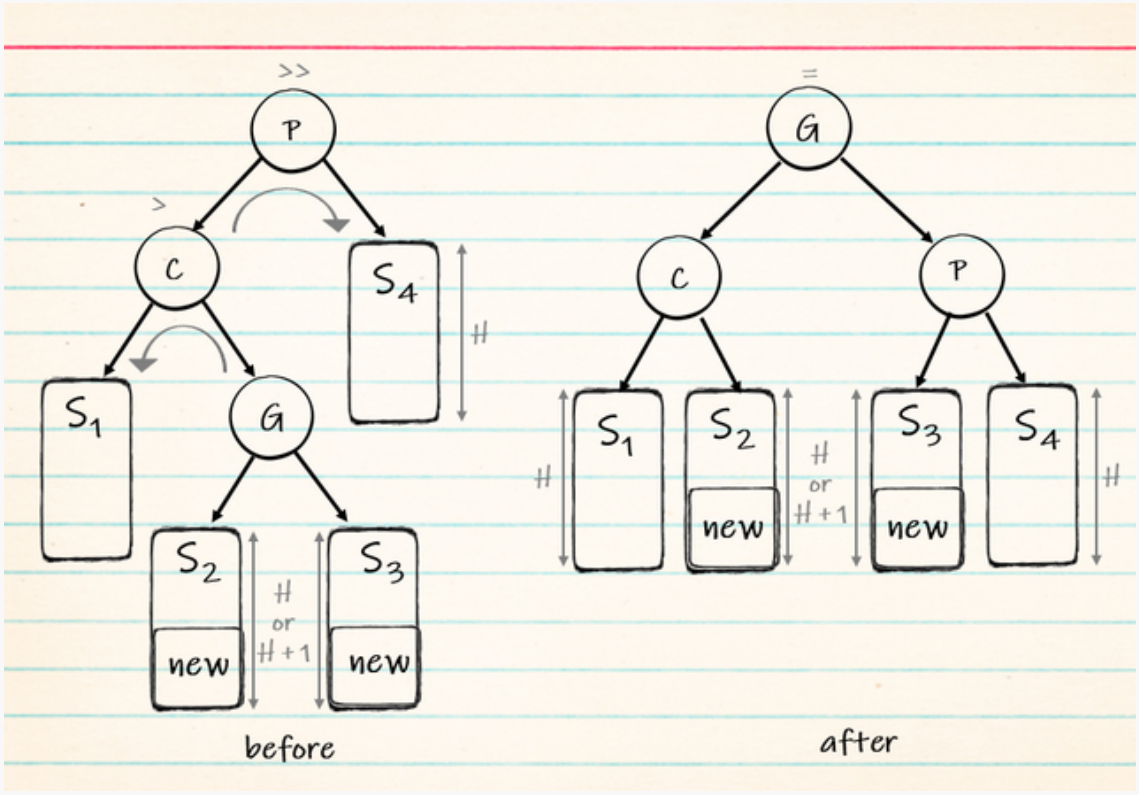

Fig 3.17. Case 2: a double rotation with the pivotâ€™s left child rotated left and the pivot rotated right.

- Case 3 and 4: The third case is a mirror image of the first case and the fourth case is a mirror image of the second case. The difference is the new node is inserted in the right subtree of the pivot node or a descendant of its right subtree.

When a new key is inserted into the tree, the balance factors of the nodes along the path from the root to the insertion point may have to be modified to reflect the insertion. The balance factor of a node along the path changes if the subtree into which the new node was inserted grows taller. The new balance factor of a node depends on its current balance factor and the subtree into which the new node was inserted. The resulting balance factors are provided here:

Current BF	**New BF** **(after inserting into left subtree)**	**New BF** **(after inserting right subtree)**
>	>>	=
=	>	<
<	=	<<

Modifications to the balance factors are made in reverse order as the recursion unwinds. When a node has a left high balance and the new node is inserted into its left child or it has a right high balance and the new node is inserted into its right child, the node is out of balance and its subtree has to be rebalanced. After rebalancing, the subtree will shrink by one level, which results in the balance factors of its ancestors remaining the same. The balance factors of the ancestors will also remain the same when the balance factor changes to equal high. After a rotation is performed, the balance factor of the impacted nodes have to be changed to reflect the new node heights. The changes required depend on which of the four cases triggered the rotation. The balance factor settings in cases 2 and 4 depend on the balance factor of the original pivot nodes grandchild (the right child of node L or the left child of node R). The new balance factors for the nodes involved in a rotation are provided below:

Case	**original G**	**new P**	**new L**	**new R**	**new G**
1	.	=	=	.	.
2	> = <	< = =	= = >	. . .	= = =
3	.	=	.	=	.
4	> = <	= = =	. . .	= = =	< = >

Fig 3.18 illustrates the construction of an AVL tree by inserting the nodes from the list [60, 25, 35, 100, 17, 80], one node at a time. Each tree in the figure shows the results after performing the indicate operation. Two double rotations are required to construct the tree: one after node 35 is inserted and one after node 80 is inserted.

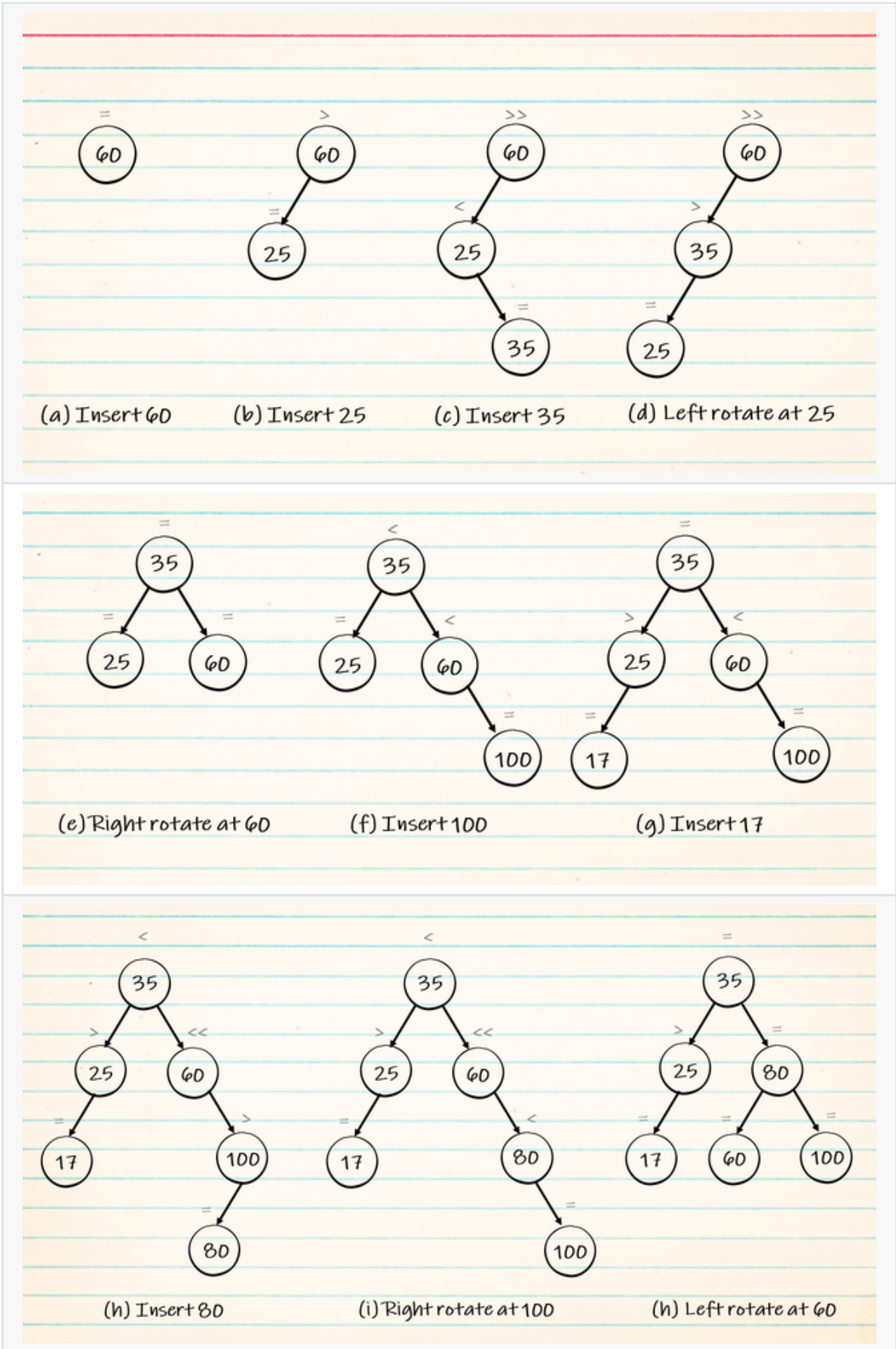

Fig 3.18. Building an AVL tree from the list of keys [60, 25, 35, 100, 17, 80].

3.6.3.3 Deletions

When an entry is removed from an AVL tree, we must ensure the balance property is maintained. As with the insert operation, deletion begins by using the corresponding operation from the binary search tree. After removing the targeted entry, subtrees may have to be rebalanced. For example, suppose we want to remove node 17 from the AVL tree in Fig 3.19(a). After removing the leaf node, the subtree rooted at node 25 is out of balance, as shown in Fig 3.19(b). A left rotation has to be performed pivoting on node 25 to correct the imbalance, as shown in Fig 3.19(c).

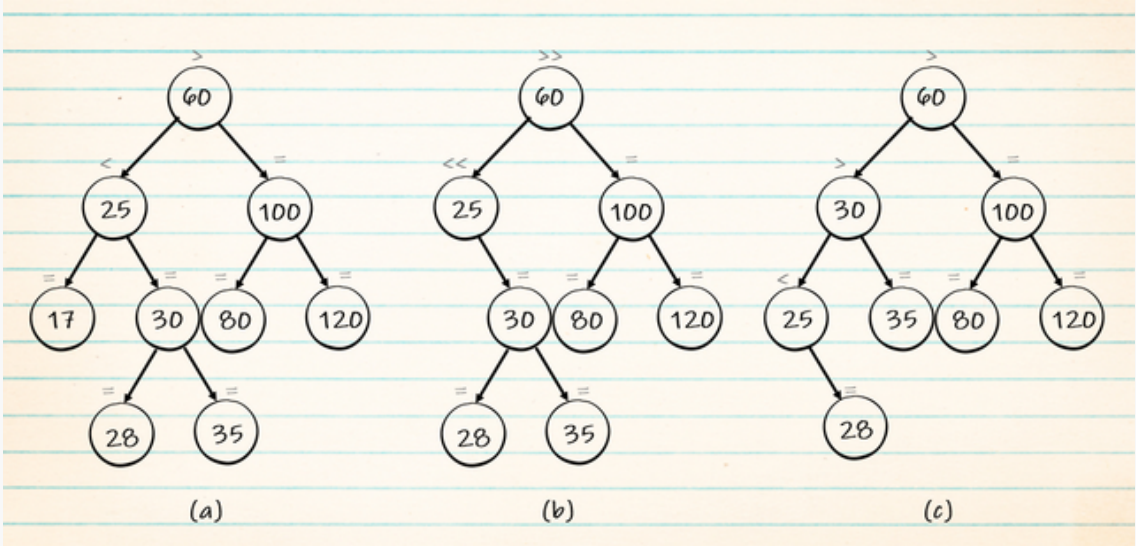

Fig 3.19. (a) Original AVL tree, (b) unbalanced tree after node 17 is deleted, and (c) the tree after a left rotation.

As with an insertion, the only subtrees that can become unbalanced are those along the path from the root to the original node containing the target. Remember, if the key being removed is in an interior node, its successor is located and copied to the node and the successor's original node is removed. In the insertion operation, at most one subtree can become unbalanced. After the appropriate rotation is performed on the subtree, the balance factors of the node's ancestors do not change. Thus, it restores the height-balance property both locally at the subtree and globally for the entire tree. This is not the case with a deletion. When a subtree is rebalanced due to a deletion, it can cause the ancestors of the subtree to then become unbalanced. This effect can ripple up all the way to the root node. So, all of the nodes along the path have to be evaluated and rebalanced if necessary.