Binary Search Tree

Trees are particularly useful in computer science, where they are employed in a wide range of algorithms. For instance, trees are used to construct efficient algorithms for locating items in a list. They can be used in algorithms, such as Huffman coding, that construct efficient codes saving costs in data transmission and storage. Trees can be used to study games such as checkers and chess and can help determine winning strategies for playing these games. Trees can be used to model procedures carried out using a sequence of decisions. Constructing these models can help determine the computational complexity of algorithms based on a sequence of decisions, such as sorting algorithms.

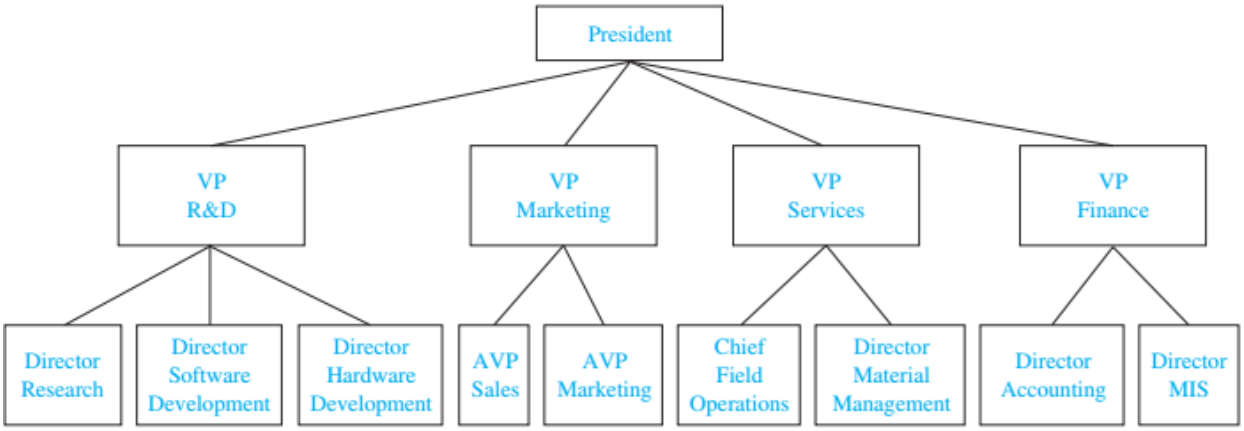

Figure 7.1: An organizational tree for a computer company.

The structure of a large organization can be modeled using a rooted tree. Each vertex in this tree represents a position in the organization. An edge from one vertex to another indicates that the person represented by the initial vertex is the (direct) boss of the person represented by the terminal vertex. The graph shown in Figure 7.1 displays such a tree. In the organization represented by this tree, the Director of Hardware Development works directly for the Vice President of R&D. The root of this tree is the vertex representing the President of the organization.

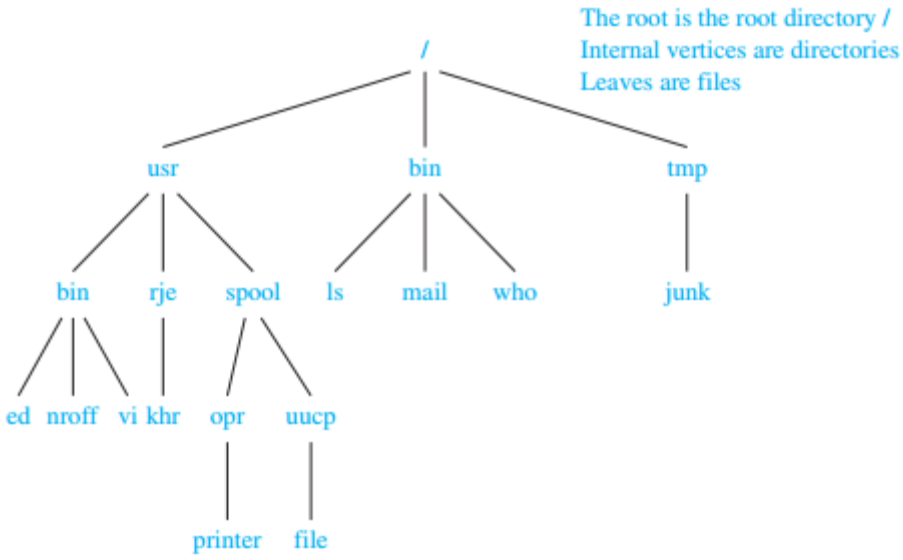

Figure 7.2: A computer file system.

Files in computer memory can be organized into directories. A directory can contain both files and subdirectories. The root directory contains the entire file system. Thus, a file system may be represented by a rooted tree, where the root represents the root directory, internal vertices represent subdirectories, and leaves represent ordinary files or empty directories. One such file system is shown in Figure 7.2. In this system, the file khr is in the directory rje. (Note that links to files where the same file may have more than one pathname can lead to circuits in computer file systems.)

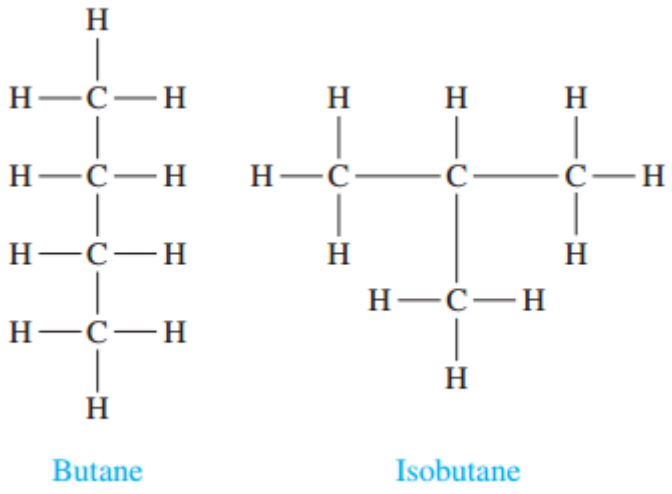

Figure 7.3: The two isomers of butane.

Graphs can be used to represent molecules, where atoms are represented by vertices and bonds between them by edges. One of the examples is shown in Figure 7.3 which are called

saturated hydrocarbons. (Isomers represent compounds with the same chemical formula but different chemical properties.)

1.1 Tree Terminologies

As a data structure, a tree consists of one or more nodes, where each node has a value and a list of references to other (its children) nodes. A tree must have a node designated as root. If a tree has only one node, that node is the root node. Root is never referenced by any other node. Having more than one node indicates that the root have some (at least one) references to its children nodes and the children nodes (might) have references to their children nodes and so on.

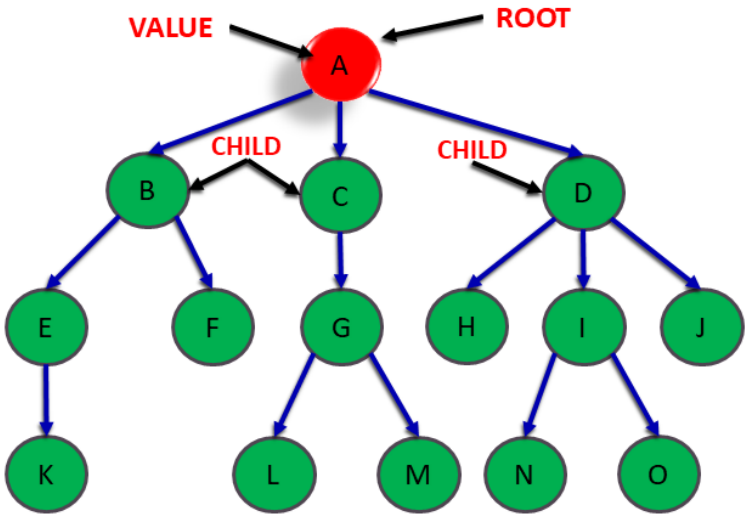

Figure 7.4: A tree

In figure 7.4, an example of tree is shown. Generally it is considered that in a tree, there is only one path going from one node to another node. There cannot be any cycle or loop. The link from a node to other node is called an edge.

The terminology for trees has botanical and genealogical origins. Suppose that T is a tree. If v is a **vertex** or **node** in T other than the **root**, the **parent** of v is the unique vertex u such that there is a directed edge from u to v. When u is the parent of v, v is called a child of u. Vertices with the same parent are called **siblings** or **children** of the parent. The ancestors of a vertex other than the root are the vertices in the path from the root to this vertex, excluding the vertex itself and including the root (that is, its parent, its parent's parent, and so on, until the root is reached). The descendants of a vertex v are those vertices that have v as an ancestor. A vertex of a tree is called a **leaf** if it has no **children**. Vertices that have children are called **internal** vertices. The root is an internal vertex unless it is the only vertex in the graph, in which case it is a leaf.

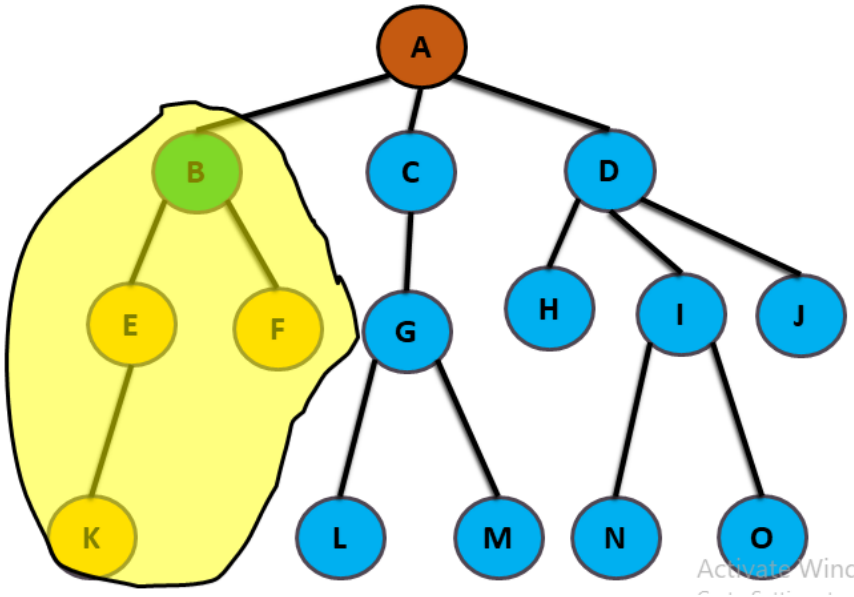

Figure 7.5: An example of Sub tree

The tree rooted by a child is called a sub tree. Thus a tree may have many sub trees. In Figure 7.5, a sub tree is shown (tree encircled in yellow colored area). Similarly, there will be other sub trees rooted at vertex C and D. And a sub tree may have many other sub trees inside it.

1.2 m-ary tree and Binary Tree

m-ary tree: A Tree is an m-ary Tree when each of its node has no more than m children.

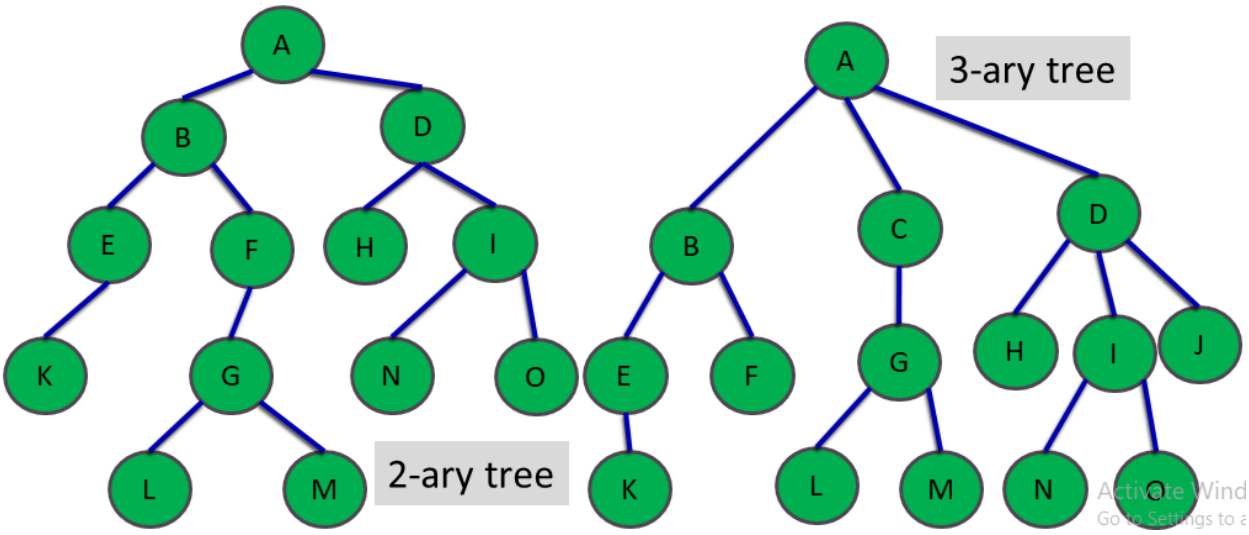

Figure 7.6: m-ary tree

A binary tree is a basically a 2-ary tree where a node can have no more than 2 children.

Complete Binary Tree: A complete binary tree is a binary tree, which is completely filled with nodes from top to bottom and left to right. In complete binary tree some of the nodes only for the bottom level might be absent (here nodes after N as shown in Figure 7.7).

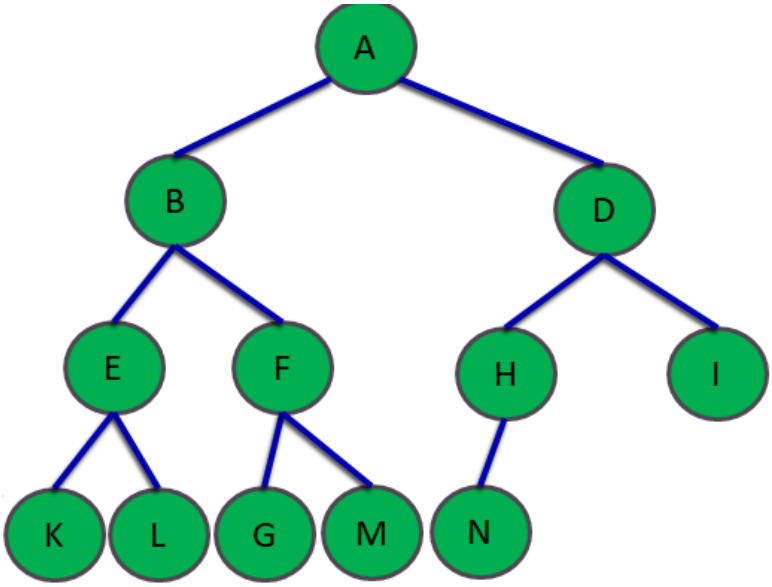

Figure 7.7: A complete binary tree

Full Binary Tree: In addition to a complete binary tree, in a full binary tree the last/bottom level must also be filled up. So it can be easily noticed that a full binary tree is always a complete binary tree but the vice-versa may not be true. Figure 7.8 shows a full binary tree where all the levels full with maximum nodes.

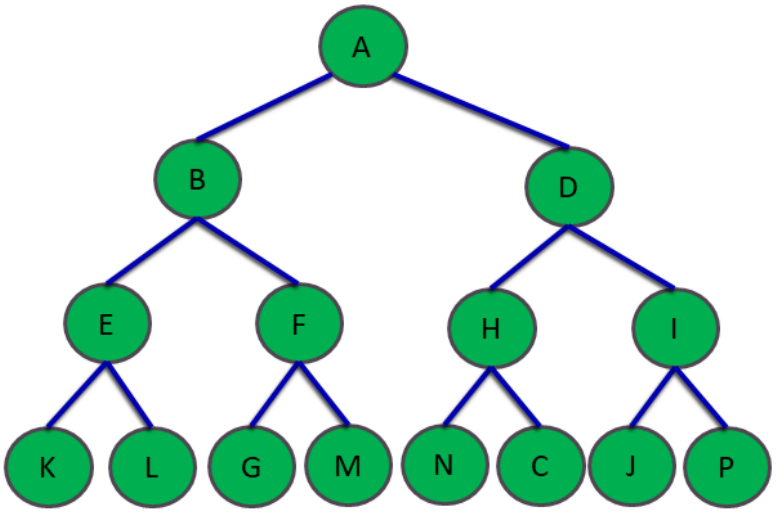

Figure 7.8: A full binary tree

In a full binary tree, number of nodes at each level l is 2^l

If there are L labels then total number of nodes will be $2^0 + 2^1 + 2^2 + \cdots + 2^{L-1} = 2^L - 1$

If total number of nodes is n then the height of the tree is $\lceil log_2 n \rceil$

1.3 Tree Traversal

Procedures for systematically visiting every vertex of a tree are called **traversal** algorithms. We will describe three of the most commonly used such algorithms, **preorder** traversal, **inorder** traversal, and **postorder** traversal. Each of these algorithms can be defined recursively. We will explain the traversal algorithms for binary tree.

Inorder traversal: In this traversal, the left sub tree is visited first then the root is visited then the right sub tree. Recursively the left sub tree and the right sub tree will be visited in the same manner.

Preorder traversal: In this traversal the root is visited first then the left sub tree is visited then the right sub tree. Recursively the left sub tree and the right sub tree will be visited in the same manner.

Postorder traversal: In this traversal, the left sub tree is visited first then the right sub tree is visited then the root. Recursively the left sub tree and the right sub tree will be visited in the same manner.

Figures[1] 7.9-7.11 shows the inorder, preorder and postorder traversal.

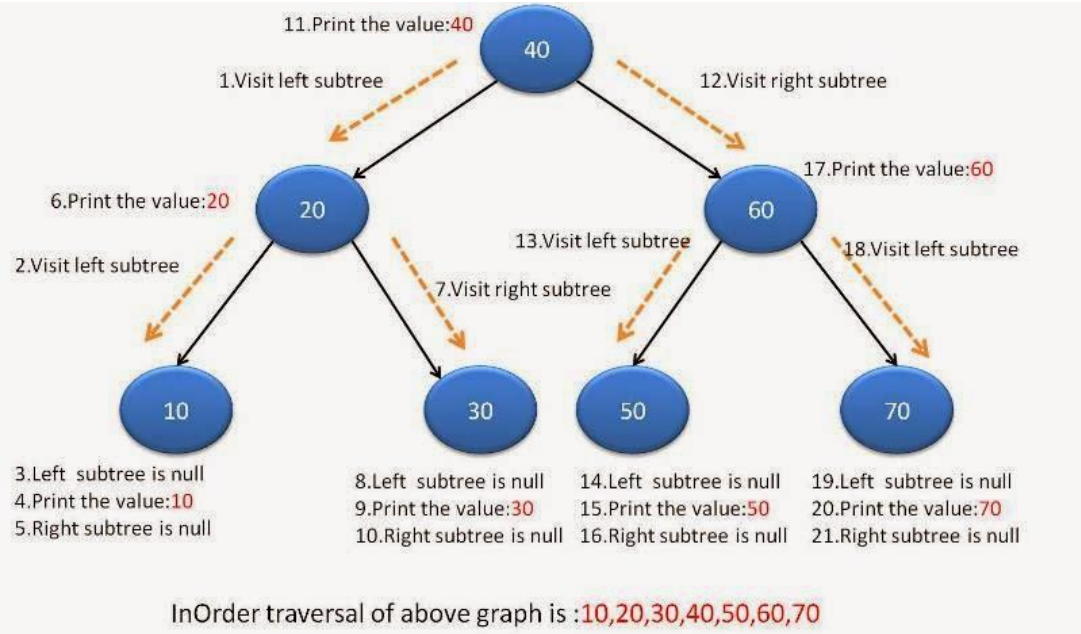

Figure 7.9: inorder traversal

[1] https://java2blog.com/binary-tree-inorder-traversal-in-java/

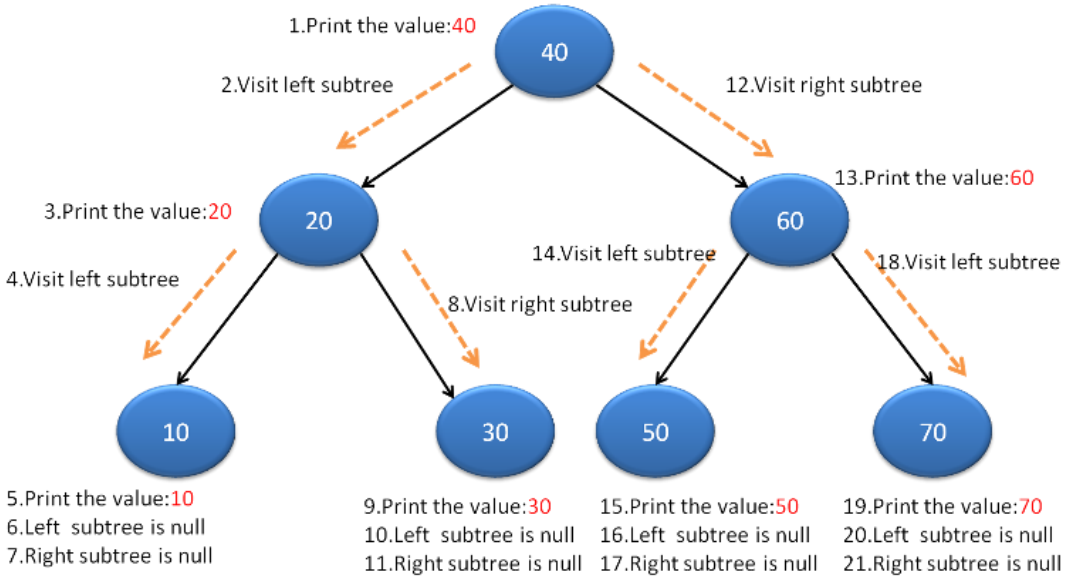

Figure 7.10: preorder traversal

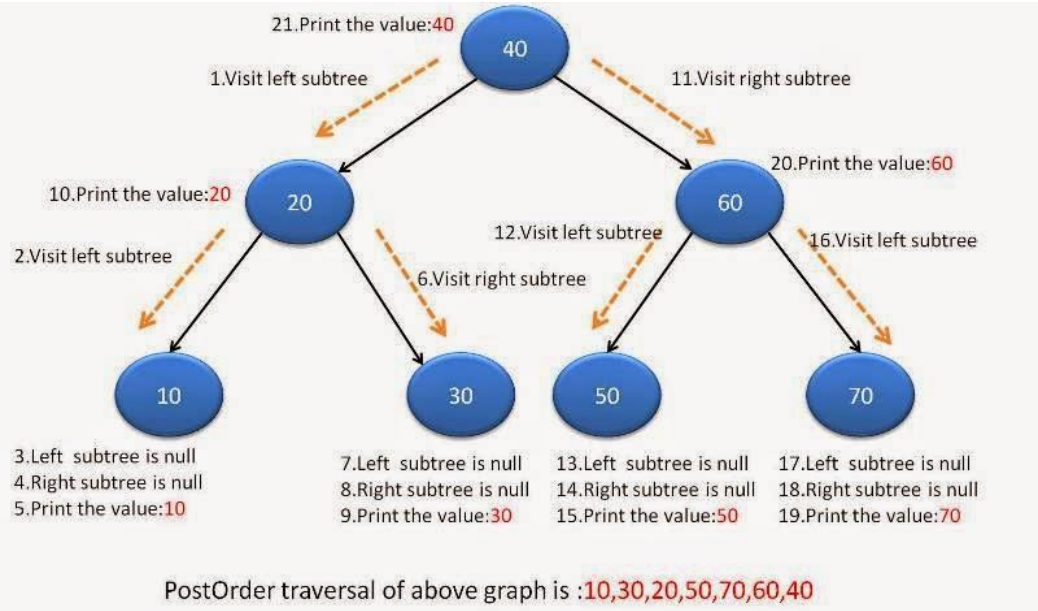

Figure 7.11: postorder traversal

1.4 Binary Search Tree

Consider the problem of searching a linked list for some target key. There is no way to move through the list other than one node at a time, and hence searching through the list must always reduce to a sequential search. As you know, sequential search is usually very slow in comparison with binary search. Hence, assuming we

can keep the keys in order, searching becomes much faster if we use a contiguous list and binary search. Suppose we also frequently need to make changes in the list, inserting new entries or deleting old entries. Then it is much slower to use a contiguous list than a linked list, because insertion or removal in a contiguous list requires moving many of the entries every time, whereas a linked list requires only adjusting a few pointers.

Binary trees provide an excellent solution to this problem. By making the entries of an ordered list into the nodes of a binary tree, we shall find that we can **search** for a target key in O(log n) steps, just as with binary search, and we shall obtain algorithms for **inserting** and deleting entries also in time O(log n).

A **binary search tree (BST)** is a binary tree that is either empty or in which every node has a key (within its data entry) and satisfies the following conditions:

1. The key of the root (if it exists) is greater than the key in any node in the left subtree of the root.
2. The key of the root (if it exists) is less than the key in any node in the right subtree of the root.
3. The left and right subtrees of the root are again binary search trees.

No two entries in a binary search tree may have equal keys.

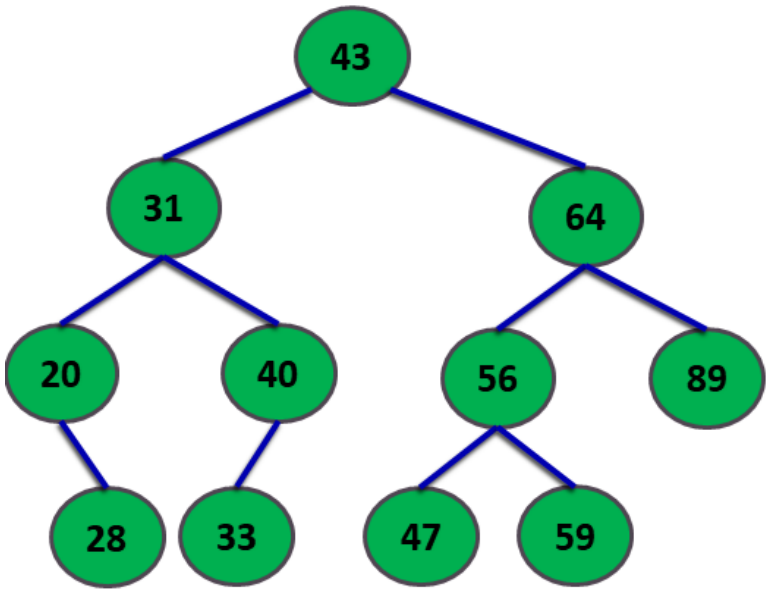

Figure 7.12: An example of a Binary Search Tree

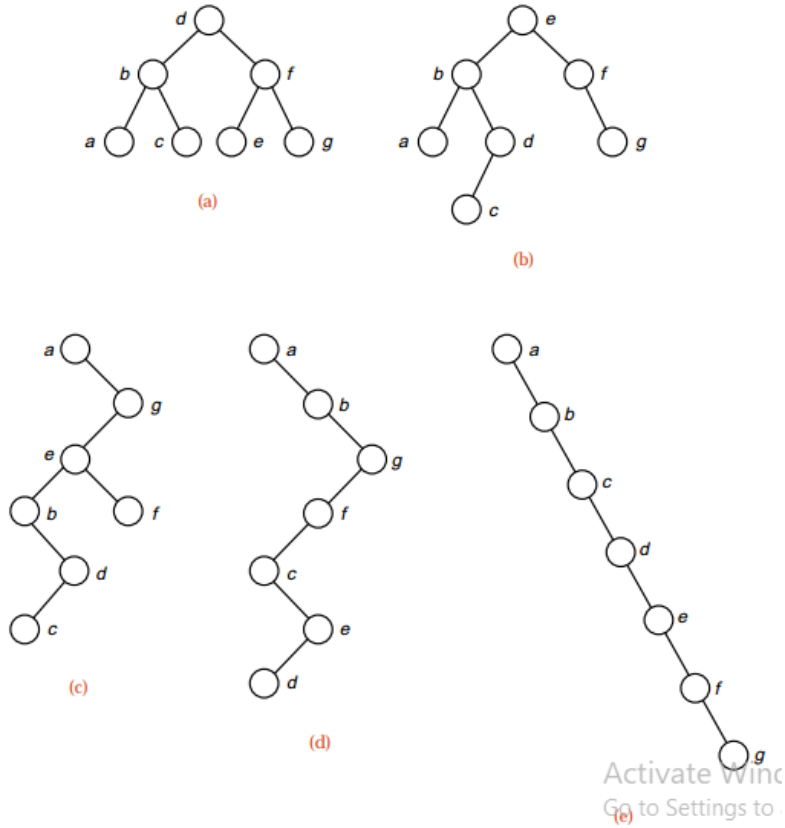

Figure 7.13: Several binary search trees with the same keys

1.5 Insertion into a BST

The next important operation for us to consider is the insertion of a new node into a binary search tree in such a way that the keys remain properly ordered; that is, so that the resulting tree satisfies the definition of a binary search tree.

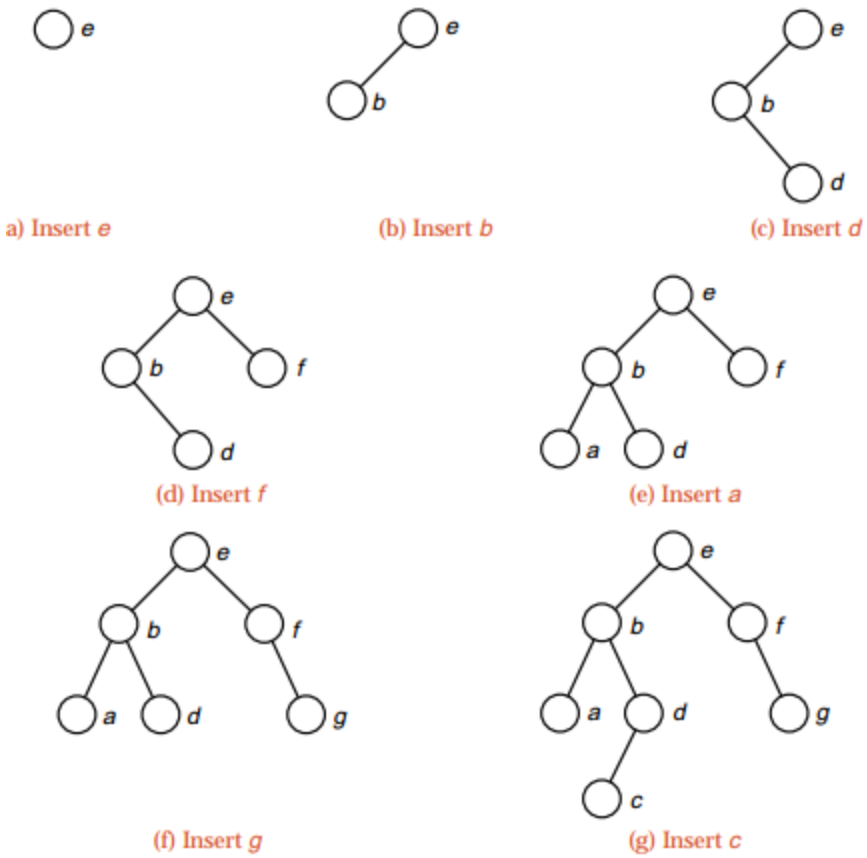

Figure 7.14: Insertions into a binary search tree

When the first entry, e, is inserted, it becomes the root, as shown in part (a). Since b comes before e, its insertion goes into the left subtree of e, as shown in part (b). Next we insert d, first comparing it to e and going left, then comparing it to b and going right. The next insertion, f, goes to the right of the root, as shown in part (d) of Figure 10.9. Since a is the earliest key inserted so far, it moves left from e and then from b. The key g, similarly, comes last in alphabetical order, so its insertion moves as far right as possible, as shown in part (f). The insertion of c, finally, compares first with e, goes left, then right from b and left from d. Hence we obtain the binary search tree shown in the last part of Figure 7.14.

It is quite possible that a different order of insertion can produce the same binary search tree. The final tree in Figure 7.14, for example, can be obtained by inserting the keys in either of the orders

e, f, g, b, a, d, c or e, b, d, c, a, f, g,

as well as several other orders.

1.6 Searching into a BST

To search for the target, we first compare it with the entry at the root of the tree. If their keys match, then we are finished. Otherwise, we go to the left subtree or right subtree as appropriate and repeat the search in that subtree.

Let us, for example, search for the key 33 in the binary search tree of Figure 7.12. We first compare 33 with the entry in the root, 43. Since 33 is smaller than 43, we move to the left and next compare 33 with 31. Since 33 is larger than 31, we move right and compare 33 with 40. Now 33 is smaller than 40, so we move to the left and find the desired target.

The tree shown as part (a) of Figure 7.13 is the best possible for searching. It is as "bushy" as possible: It has the smallest possible height for a given number of vertices. The number of vertices between the root and the target, inclusive, is the number of comparisons that must be done to find the target. The bushier the tree, therefore, the smaller the number of comparisons that will usually need to be done.

1.7 Deletion from a BST

If the node to be removed is a leaf, then the process is easy: We need only replace the link to the removed node by NULL. The process remains easy if the removed node has only one nonempty subtree: We adjust the link from the parent of the removed node to point to the root of its nonempty subtree.

When the node to be removed has both left and right subtrees nonempty, however, the problem is more complicated. To which of the subtrees should the parent of the removed node now point? What is to be done with the other subtree? This problem is illustrated in Figure 7.15. First, we find the immediate predecessor of the node under inorder traversal by moving to its left child and then as far right as possible. (The immediate successor would work just as well.) The immediate predecessor has no right child (since we went as far right as possible), so it can be removed from its current position without difficulty. It can then be placed into the tree in the position formerly occupied by the node that was supposed to be removed, and the properties of a binary search tree will still be satisfied, since there were no keys in the original tree whose ordering comes between the removed key and its immediate predecessor.

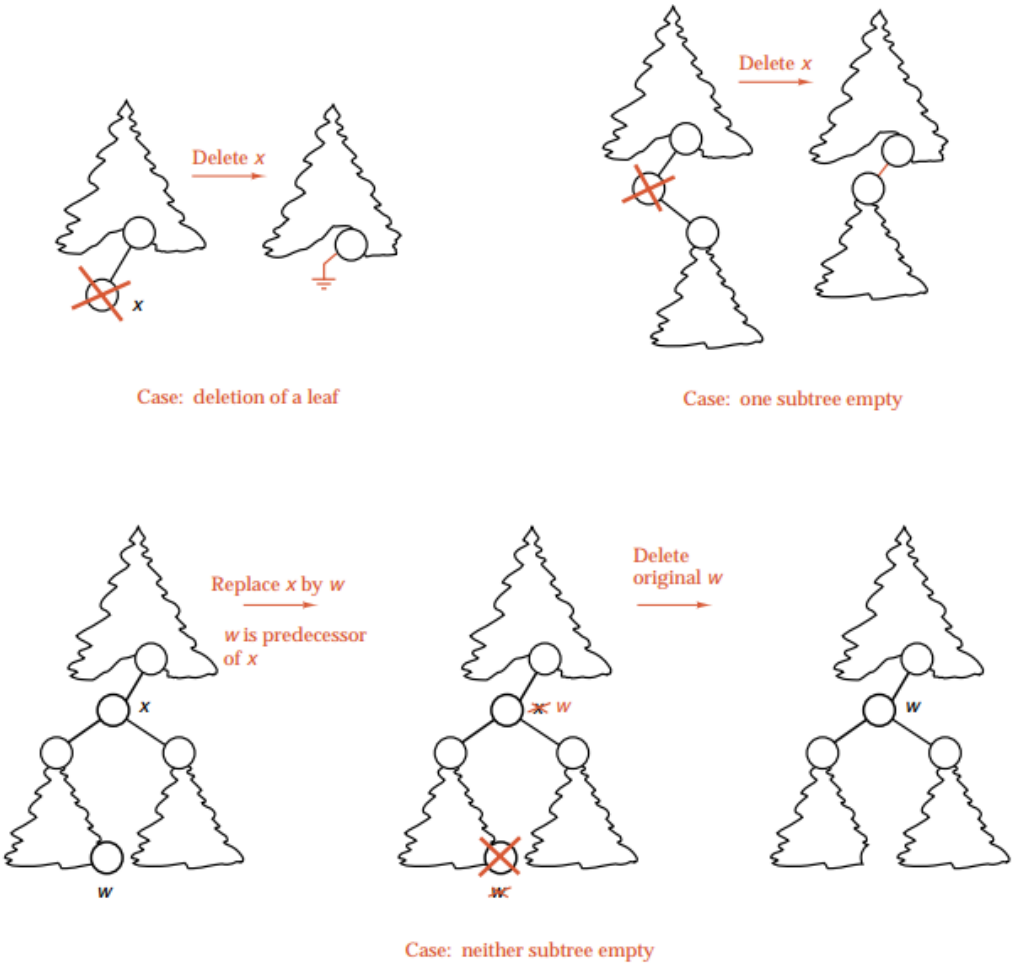

Figure 7.15: Deletion of a node from a binary search tree

1.8 References

- Kruse, R. L., Ryba, A. J., & Ryba, A. (1999). Data structures and program design in C++ (p. 280). New York: Prentice Hall.
- Shaffer, C. A. (2012). Data structures and algorithm analysis. Update, 3, 0-3.
- Dale, N. B. (2003). C++ plus data structures. Jones & Bartlett Learning.