6

Trees

A **tree** is a hierarchical form of data structure. Data structures such as lists, queues, and stacks are linear in that the items are stored in a sequential way. However, a tree is a non-linear data structure, as there is a **parent-child relationship** between the items. The top of the tree's data structure is known as a **root node**. This is the ancestor of all other nodes in the tree.

Tree data structures are very important, owing to their use in various applications, such as parsing expressions, efficient searches, and priority queues. Certain document types, such as `XML` and `HTML`, can also be represented in a tree.

In this chapter, we will cover the following topics:

- Terms and definitions of trees
- Binary trees and binary search trees
- Tree traversal
- Binary search trees

Terminology

Let's consider some of the terminology associated with tree data structures.

To understand trees, we need to first understand the basic concepts related to them. A tree is a data structure in which data is organized in a hierarchical form.

Figure 6.1 contains a typical tree consisting of character nodes lettered `A` through to `M`:

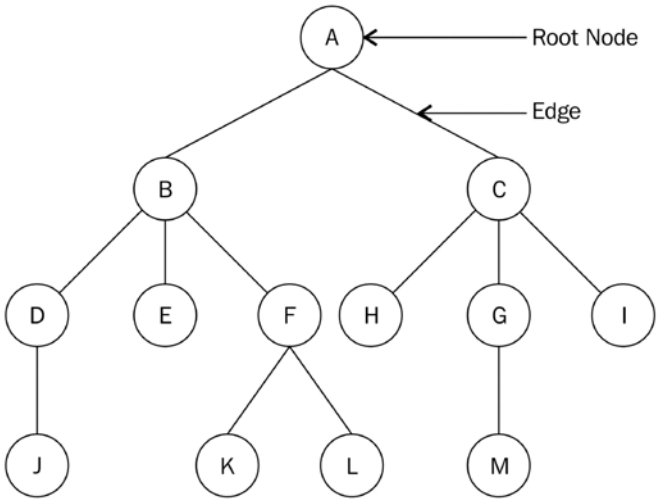

Figure 6.1: Example tree data structure

Here is a list of terms associated with a tree:

- **Node**: Each circled letter in the preceding diagram represents a node. A node is any data structure that stores data.
- **Root node**: The root node is the first node from which all other nodes in the tree descend from. In other words, a root node is a node that does not have a parent node. In every tree, there is always one unique root node. The root node is node `A` in the above example tree.
- **Subtree**: A subtree is a tree whose nodes descend from some other tree. For example, nodes `F`, `K`, and `L` form a subtree of the original tree.
- **Degree**: The total number of children of a given node is called the **degree of the node**. A tree consisting of only one node has a degree of 0. The degree of node `A` in the preceding diagram is 2, the degree of node `B` is 3, the degree of node `C` is 3, and, the degree of node `G` is 1.
- **Leaf node**: The leaf node does not have any children and is the terminal node of the given tree. The degree of the leaf node is always 0. In the preceding diagram, the nodes `J`, `E`, `K`, `L`, `H`, `M`, and `I` are all leaf nodes.

- **Edge**: The connection among any given two nodes in the tree is called an edge. The total number of edges in a given tree will be a maximum of one less than the total nodes in the tree. An example edge is shown in *Figure 6.1*.
- **Parent**: A node that has a subtree is the parent node of that subtree. For example, node `B` is the parent of nodes `D`, `E`, and `F`, and node `F` is the parent of nodes `K` and `L`.
- **Child**: This is a node that is descendant from a parent node. For example, nodes `B` and `C` are children of parent node `A`, while nodes `H`, `G`, and `I` are the children of parent node `C`.
- **Sibling**: All nodes with the same parent node are siblings. For example, node `B` is the sibling of node `C`, and, similarly, nodes `D`, `E`, and `F` are also siblings.
- **Level**: The root node of the tree is considered to be at level 0. The children of the root node are considered to be at level 1, and the children of the nodes at level 1 are considered to be at level 2, and so on. For example, in *Figure 6.1*, root node `A` is at level 0, nodes `B` and `C` are at level 1, and nodes `D`, `E`, `F`, `H`, `G`, and `I` are at level 2.
- **Height of a tree**: The total number of nodes in the longest path of the tree is the height of the tree. For example, in *Figure 6.1*, the height of the tree is 4, as the longest paths, `A-B-D-J`, `A-C-G-M`, and `A-B-F-K`, all have a total number of four nodes each.
- **Depth**: The depth of a node is the number of edges from the root of the tree to that node. In the preceding tree example, the depth of node `H` is 2.

In linear data structures, data items are stored in sequential order, whereas non-linear data structures store data items in a non-linear order, where a data item can be connected to more than one other data item. All of the data items in linear data structures, such as *arrays*, *lists*, *stacks*, and *queues*, can be traversed in one pass, whereas this is not possible in the case of non-linear data structures such as trees; they store the data differently from other linear data structures.

In a tree data structure, the nodes are arranged in a parent-child relationship. There should not be any cycle among the nodes in trees. The tree structure has nodes to form a hierarchy, and a tree that has no nodes is called an **empty tree**.

First, we'll discuss one of the most important kind of trees, that is, the **binary tree**.

Binary trees

A binary tree is a collection of nodes, where the nodes in the tree can have zero, one, or two child nodes. A simple binary tree has a maximum of two children, that is, the left child and the right child.

For example, in the binary tree shown in *Figure 6.2*, there is a root node that has two children (a left child, a right child):

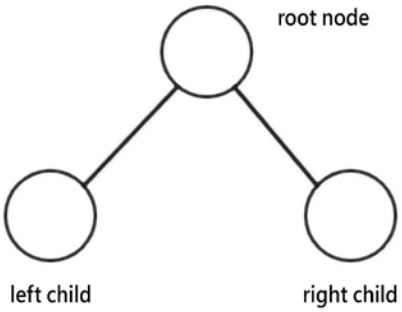

Figure 6.2: Example of a binary tree

The nodes in the binary tree are organized in the form of the left subtree and right subtree. For example, a tree of five nodes is shown in *Figure 6.3* that has a root node, `R`, and two subtrees, i.e. left subtree, `T1`, and right subtree, `T2`:

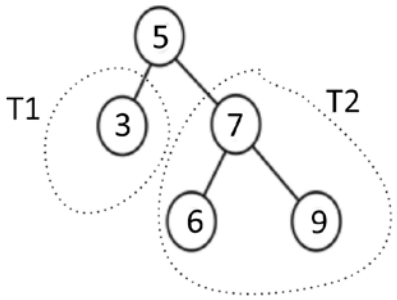

Figure 6.3: An example binary tree of five nodes

A regular binary tree has no other rules as to how elements are arranged in the tree. It should only satisfy the condition that each node should have a maximum of two children.

A tree is called a **full binary** tree if all the nodes of a binary tree have either zero or two children, and if there is no node that has one child. An example of a full binary tree is shown in *Figure 6.4*:

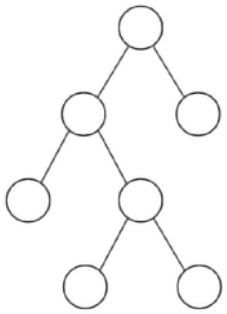

Figure 6.4: An example of a full binary tree

A perfect binary tree has all the nodes in the binary tree filled, and it doesn't have space vacant for any new nodes; if we add new nodes, they can only be added by increasing the tree's height. A sample perfect binary tree is shown in *Figure 6.5*:

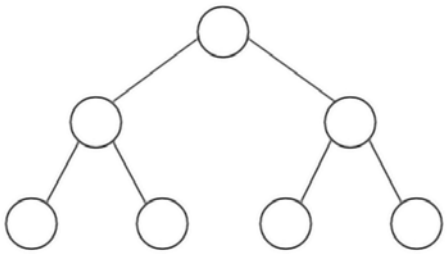

Figure 6.5: An example of a perfect binary tree

A **complete binary tree** is filled with all possible nodes except with a possible exception at the lowest level of the tree. All nodes are also filled on the left side. A complete binary tree is shown in *Figure 6.6*:

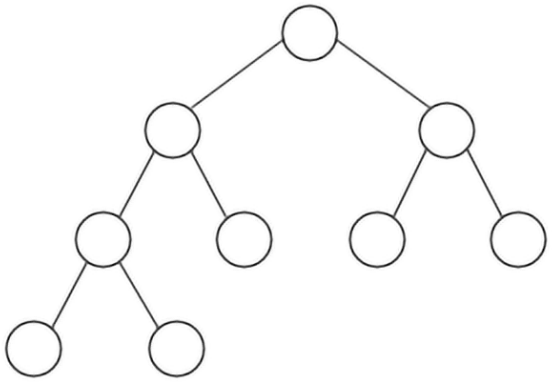

Figure 6.6: An example of a complete binary tree

A binary tree can be balanced or unbalanced. In a balanced binary tree, the difference in height of the left and right subtrees for every node in the tree is no more than 1. A balanced tree is shown in *Figure 6.7*:

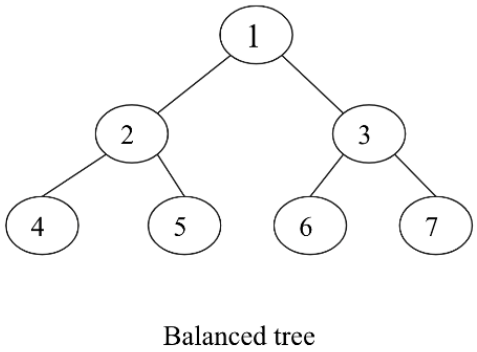

Figure 6.7: An example of a balanced tree

An unbalanced binary tree is a binary tree that has a difference of more than 1 between the right subtree and left subtree. An example of an unbalanced tree is shown in *Figure 6.8*:

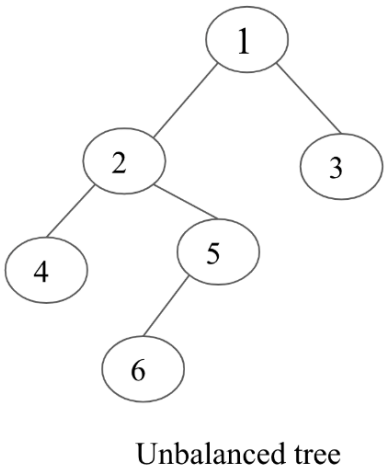

Figure 6.8: An example of an unbalanced tree

Next, we'll discuss the details of the implementation of a simple binary tree.

Implementation of tree nodes

As we have already discussed in previous chapters, a node consists of data items and references to other nodes.

In a binary tree node, each node will contain data items and two references that will point to their left and right children, respectively. Let's look at the following code for building a binary tree `Node` class in Python:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None
```

To better understand the working of this class, let's first create a binary tree of four nodes—`n1`, `n2`, `n3`, and `n4`—as shown in *Figure 6.9*:

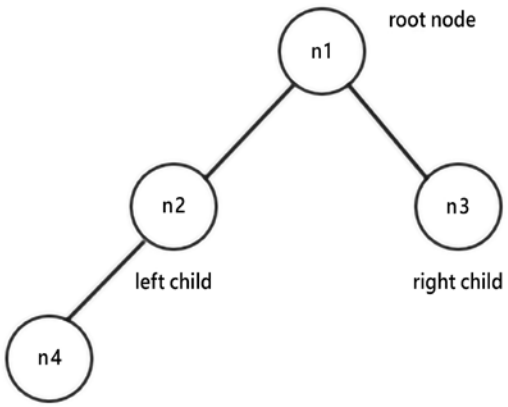

Figure 6.9: An example binary tree of four nodes

For this, we firstly create four nodes—`n1`, `n2`, `n3`, and `n4`:

```
n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")
```

Next, we connect the nodes to each other according to the previously discussed properties of a binary tree. `n1` will be the root node, with `n2` and `n3` as its children. Furthermore, `n4` will be the left child of `n2`. The next code snippet shows the connections among different nodes of the tree according to the desired tree, as shown in *Figure 6.9*:

```
n1.left_child = n2
n1.right_child = n3
n2.left_child = n4
```

Here, we have created a very simple tree structure of four nodes. After creating a tree, one of the most important operations that is to be applied to trees is **traversal**. Next, we'll understand how we can traverse the tree.

Tree traversal

The method to visit all the nodes in a tree is called **tree traversal**. In the case of a linear data structure, data element traversal is straightforward since all the items are stored in a sequential manner, so each data item is visited only once. However, in the case of non-linear data structures, such as trees and graphs, traversal algorithms are important. To understand traversing, let's traverse the left subtree of the binary tree we created in the previous section. For this, we start from the root node, print out the node, and move down the tree to the next left node. We keep doing this until we have reached the end of the left subtree, like so:

```
current = n1
while current:
    print(current.data)
    current = current.left_child
```

The output of traversing the preceding code block is as follows:

```
root node
left child node
left grandchild node
```

There are multiple ways to process and traverse the tree that depend upon the sequence of visiting the root node, left subtree, or right subtree. Mainly, there are two kinds of approaches, firstly, one in which we start from a node and traverse every available child node, and then continue to traverse to the next sibling. There are three possible variations of this method, namely, **in-order**, **pre-order**, and **post-order**. Another approach to traverse the tree is to start from the root node and then visit all the nodes on each level, and process the nodes level by level. We will discuss each approach in the following sections.

In-order traversal

In-order tree traversal works as follows: we start traversing the left subtree recursively, and once the left subtree is visited, the root node is visited, and then finally the right subtree is visited recursively. It has the following three steps:

- We start traversing the left subtree and call an ordering function recursively

- Next, we visit the root node
- Finally, we traverse the right subtree and call an ordering function recursively

So, in a nutshell, for in-order tree traversal, we visit the nodes in the tree in the order of left subtree, root, then the right subtree.

Let's consider an example tree shown in *Figure 6.10* to understand in-order tree traversal:

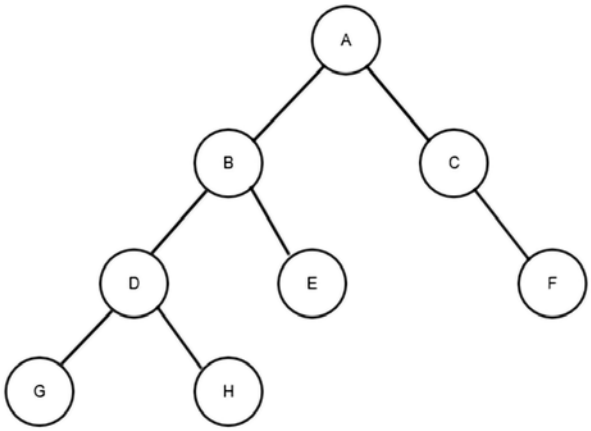

Figure 6.10: An example binary tree for in-order tree traversal

In the binary tree shown in *Figure 6.10*, the working of the in-order traversal is as follows: first, we recursively visit the left subtree of the root node `A`. The left subtree of node `A` has node `B` as the root node, so we again go to the left subtree of root node `B`, that is, node `D`. We recursively go to the left subtree of root node `D` so that we get the left child of root node `D`. We visit the left child, `G`, then visit the root node, `D`, and then visit the right child, `H`.

Next, we visit node `B` and then visit node `E`. In this manner, we have visited the left subtree of root node `A`. Next, we visit root node `A`. After that, we visit the right subtree of root node `A`. Here, we first go to the left subtree of root node `C`, which is null, so next, we visit node `C`, and then we visit the right child of node `C`, that is, node `F`.

Therefore, the in-order traversal for this example tree is `G-D-H-B-E-A-C-F`.

The Python implementation of a recursive function to return an in-order listing of nodes in a tree is as follows:

```
def inorder(root_node):
    current = root_node
    if current is None:
```

```
        return
    inorder(current.left_child)
    print(current.data)
    inorder(current.right_child)

inorder(n1)
```

Firstly, we check if the current node is null or empty. If it is not empty, we traverse the tree. We visit the node by printing the visited node. In this case, we first recursively call the `inorder` function with `current.left_child`, then we visit the root node, and finally, we recursively call the `inorder` function with `current.right_child`.

Finally, when we apply the above in-order traversal algorithm on the above sample tree of four nodes. With `n1` as the root node, we get the following output:

```
left grandchild node
left child node
root node
right child node
```

Next, we will discuss pre-order traversal.

Pre-order traversal

Pre-order tree traversal traverses the tree in the order of the root node, the left subtree, and then the right subtree. It works as follows:

1. We start traversing with the root node
2. Next, we traverse the left subtree and call an ordering function with the left subtree recursively
3. Next, we visit the right subtree and call an ordering function with the right subtree recursively

Consider the example tree shown in *Figure 6.11* to understand pre-order traversal:

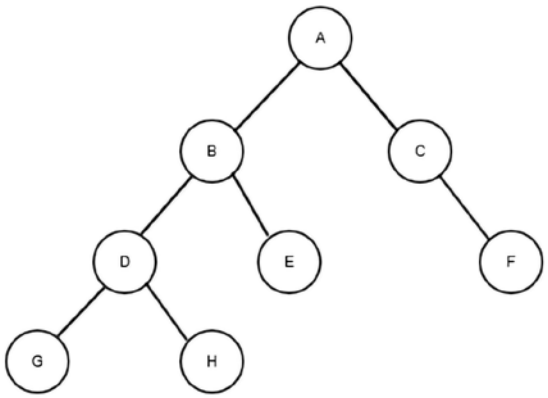

Figure 6.11: An example tree to understand pre-order traversal

The pre-order traversal for the example binary tree shown in *Figure 6.11* works as follows: first, we visit root node `A`. Next, we go to the left subtree of root node `A`. The left subtree of node `A` has node `B` as the root, so we visit this root node, and then go to the left subtree of root node `B`, node `D`. We visit node `D` and then the left subtree of root node `D`, and then we visit the left child, `G`, which is the subtree of root node `D`. Since there is no child of node `G`, we visit the right subtree. We visit the right child of the subtree of root node `D`, node `H`. Next, we visit the right child of the subtree of root node `B`, node `E`.

In this manner, we have visited root node `A` and the left subtree of root node `A`. Next, we visit the right subtree of root node `A`. Here, we visit root node `C`, and then we go to the left subtree of root node `C`, which is null, so we visit the right child of node `C`, node `F`.

The pre-order traversal for this example tree would be `A-B-D-G-H-E-C-F`.

The recursive function for the pre-order tree traversal is as follows:

```
def preorder(root_node):
    current = root_node
    if current is None:
        return
    print(current.data)
    preorder(current.left_child)
    preorder(current.right_child)

preorder(n1)
```

First, we check if the current node is null or empty. If it is empty, it means the tree is an empty tree, and if the current node is not empty, then we traverse the tree using the pre-order algorithm. The pre-order traversal algorithm traverses the tree in the order of root, left subtree, and right subtree recursively, as shown in the above code. Finally, when we apply the above pre-order traversal algorithm on the above sample tree of four nodes with `n1` node as the root node, we get the following output:

```
root node
left child node
left grandchild node
right child node
```

Next, we will discuss post-order traversal.

Post-order traversal

Post-order tree traversal works as follows:

1. We start traversing the left subtree and call an ordering function recursively
2. Next, we traverse the right subtree and call an ordering function recursively
3. Finally, we visit the root node

So, in a nutshell, for post-order tree traversal, we visit the nodes in the tree in the order of the left subtree, the right subtree, and finally the root node.

Consider the following example tree shown in *Figure 6.12* to understand post-order tree traversal:

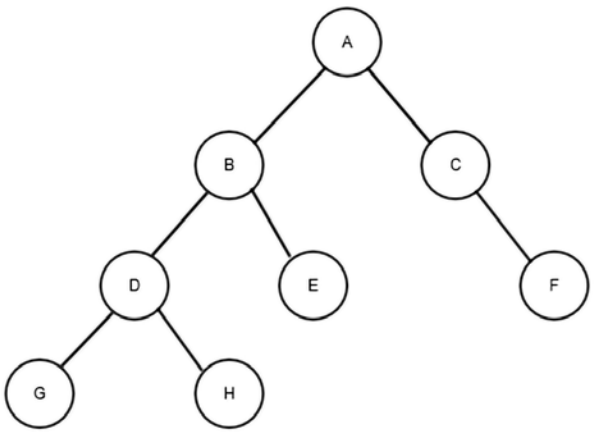

Figure 6.12: An example tree to understand pre-order traversal

In the preceding figure, *Figure 6.12*, we first visit the left subtree of root node `A` recursively. We get to the last left subtree, that is, root node `D`, and then we visit the node to the left of it, which is node `G`. We visit the right child, `H`, after this, and then we visit root node `D`. Following the same rule, we next visit the right child of node `B`, node `E`. Then, we visit node `B`. Following on from this, we traverse the right subtree of node `A`. Here, we first reach the last right subtree and visit node `F`, and then we visit node `C`. Finally, we visit root node `A`.

The post-order traversal for this example tree would be `G-H-D-E-B-F-C-A`.

The implementation of the post-order method for tree traversal is as follows:

```
def postorder( root_node):
    current = root_node
    if current is None:
        return
    postorder(current.left_child)
    postorder(current.right_child)
    print(current.data)

postorder(n1)
```

First, we check if the current node is null or empty. If it is not empty, we traverse the tree using the post-order algorithm as discussed, and finally, when we apply the above post-order traversal algorithm on the above sample tree of four nodes with `n1` as the root node. We get the following output:

```
left grandchild node
left child node
right child node
root node
```

Next, we will discuss level-order traversal.

Level-order traversal

In this traversal method, we start by visiting the root of the tree before visiting every node on the next level of the tree. Then, we move on to the next level in the tree, and so on. This kind of tree traversal is how breadth-first traversal in a graph works, as it broadens the tree by traversing all the nodes in a level before going deeper into the tree.

Let's consider the following example tree and traverse it:

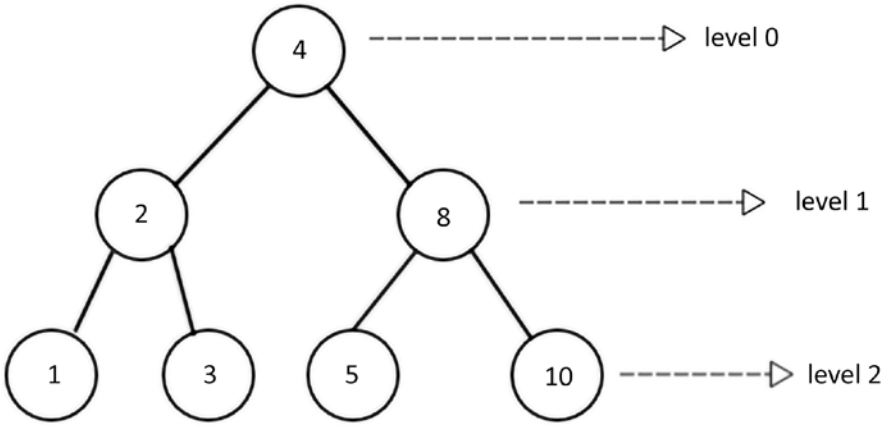

Figure 6.13: An example tree to understand level-order traversal

In *Figure 6.13*, we start by visiting the root node at level 0, which is the node with a value of `4`. We visit this node by printing out its value. Next, we move to level 1 and visit all the nodes at this level, which are the nodes with the values `2` and `8`. Finally, we move to the next level in the tree, that is, level 3, and we visit all the nodes at this level, which are `1`, `3`, `5`, and `10`. Thus, the level-order tree traversal for this tree is as follows: `4`, `2`, `8`, `1`, `3`, `5`, and `10`.

This level-order tree traversal is implemented using a queue data structure. We start by visiting the root node, and we push it into a queue. The node at the front of the queue is accessed (dequeued), which can then be either printed or stored for later use. After adding the root node, the left child node is added to the queue, followed by the right node. Thus, when traversing at any given level of the tree, all the data items of that level are firstly inserted in the queue from left to right. After that, all the nodes are visited from the queue one by one. This process is repeated for all the levels of the tree.

The traversal of the preceeding tree using this algorithm will enqueue root node `4`, dequeue it, and visit the node. Next, nodes `2` and `8` are enqueued, as they are the left and right nodes at the next level. Node `2` is dequeued so that it can be visited. Next, its left and right nodes, nodes `1` and `3`, are enqueued. At this point, the node at the front of the queue is node `8`. We dequeue and visit node `8`, after which we enqueue its left and right nodes. This process continues until the queue is empty.

The Python implementation of breadth-first traversal is as follows. We enqueue the root node and keep a list of the visited nodes in the `list_of_nodes` list. The `dequeue` class is used to maintain a queue:

```
from collections import deque
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None

n1 = Node("root node")
n2 = Node("left child node")
n3 = Node("right child node")
n4 = Node("left grandchild node")

n1.left_child = n2
n1.right_child = n3
n2.left_child = n4

def level_order_traversal(root_node):
    list_of_nodes = []
    traversal_queue = deque([root_node])
    while len(traversal_queue) > 0:
        node = traversal_queue.popleft()
        list_of_nodes.append(node.data)
        if node.left_child:
            traversal_queue.append(node.left_child)
            if node.right_child:
                traversal_queue.append(node.right_child)
    return list_of_nodes

print(level_order_traversal(n1))
```

If the number of elements in `traversal_queue` is greater than zero, the body of the loop is executed. The node at the front of the queue is popped off and added to the `list_of_nodes` list. The first `if` statement will enqueue the left child node if the `node` provided with a left node exists. The second `if` statement does the same for the right child node. Further, the `list_of_nodes` list is returned in the last statement.

The output of the above code is as follows:

```
['root node', 'left child node', 'right child node', 'left grandchild
node']
```

We have discussed different tree traversal algorithms; we can use any of these algorithms depending upon the application. In-order traversal is very useful when we need sorted contents from a tree. This also applies if we need items in descending order, which we can do by reversing the order, such as right subtree, root, and then left subtree. This is known as reverse in-order traversal. And, if we need to inspect the root before any leaves, we use pre-order traversal. Likewise, if we need to inspect the leaf nodes before the root nodes.

The following are some important applications of binary trees:

1. Binary trees as expression trees are used in compilers
2. It is also used in Huffman coding in data compression
3. Binary search trees are used for efficient searching, insertion, and deletion of a list of items
4. **Priority Queue** (**PQ**), which is used for finding and deleting minimum or maximum items in a collection of elements in logarithm time in the worst case

Next, let us discuss expression trees.

Expression trees

An expression tree is a special kind of binary tree that can be used to represent arithmetic expressions. An arithmetic expression is represented by a combination of operators and operands, where the operators can be unary or binary. Here, the operator shows which operation we want to perform, and the operator tells us what data items we want to apply those operations to. If the operator is applied to one operand, then it is called a unary operator, and if it is applied to two operands, it is called a binary operator.

An arithmetic expression can also be represented using a binary tree, which is also known as an expression tree. The **infix** notation is a commonly used notation to express arithmetic expressions where the operators are placed in between the operands. It is a commonly used method of representing an arithmetic expression. In an expression tree, all the leaf nodes contain operands and non-leaf nodes contain the operators. It is also worth noting that an expression tree will have one of its subtrees (right or left) empty in the case of a unary operator.

The arithmetic expression is shown using three notations: **infix**, **postfix**, or **prefix**. The in-order traversal of an expression tree produces the infix notation. For example, the expression tree for `3 + 4` would look as shown in *Figure 6.14*:

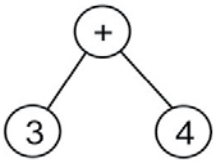

Figure 6.14: An expression tree for the expression 3 + 4

In this example, the operator is inserted (infixed) between the operands, as `3 + 4`. When necessary, parentheses can be used to build a more complex expression. For example, for `(4 + 5) * (5 - 3)`, we would get the following:

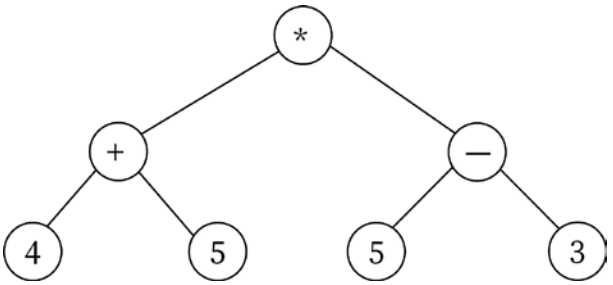

*Figure 6.15: An expression tree for the expression (4 + 5) * (5-3)*

Prefix notation is commonly referred to as *Polish* notation. In this notation, the operator comes before its operands. For example, the arithmetic expression to add two numbers, 3 and 4, would be shown as `+ 3 4`. Let's consider another example, `(3 + 4) * 5`. This can also be represented as `* (+ 3 4) 5` in prefix notation. The pre-order traversal of an expression tree results in the prefix notation of the arithmetic expression. For example, consider the expression tree shown in *Figure 6.16*:

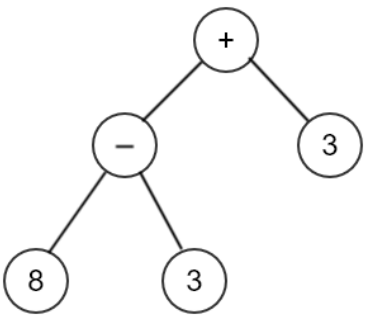

Figure 6.16: An example expression tree to understand pre-order traversal

The pre-order traversal of the expression tree shown in *Figure 6.16* will give the expression in prefix notation as `+- 8 3 3`.

Postfix, or **reverse Polish notation (RPN)**, places the operator after its operands, such as `3 4 +`. The post-order traversal of the expression tree shown in *Figure 6.17* gives the postfix notation of the arithmetic expression.

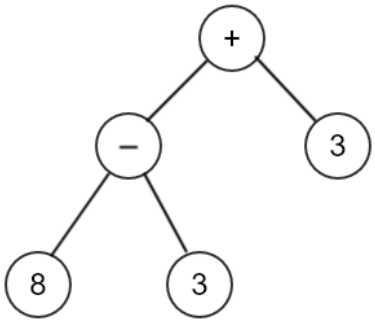

Figure 6.17: An example expression tree to understand post-order traversal

The postfix notation for the preceding expression tree is `8 3 -3 +`. We have now discussed expression trees. It is easy to evaluate an expression tree for the given arithmetic expression using the reverse Polish notation since it provides faster calculations.

Parsing a reverse Polish expression

To create an expression tree from the postfix notation, a stack is used. In this, we process one symbol at a time; if the symbol is an operand, then its references are pushed in to the stack, and if the symbol is an operator, then we pop two pointers from the stack and form a new subtree, whose root is the operator. The first reference popped from the stack is the right child of the subtree, and the second reference becomes the left child of the subtree. Further, a reference to this new subtree is pushed into the stack. In this manner, all the symbols of the postfix notation are processed to create the expression tree.

Let's take an example of `4 5 + 5 3 - *`.

Firstly, we push symbols 4 and 5 onto the stack, and then we process the next symbol + as shown in *Figure 6.18*:

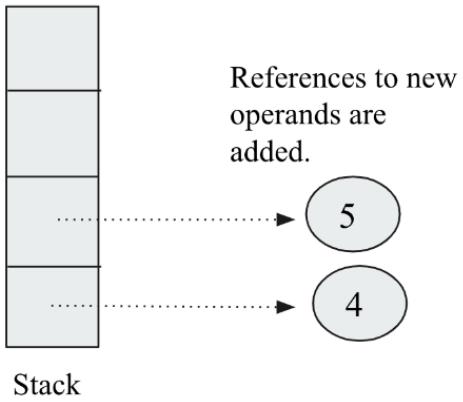

Figure 6.18: Operands 4 and 5 are pushed onto the stack

When the new symbol + is read, it is made into a root node of a new subtree, and then two references are popped from the stack, and the topmost reference is added as the right of the root node, and the next popped reference is added as the left child of the subtree, as shown in *Figure 6.19*:

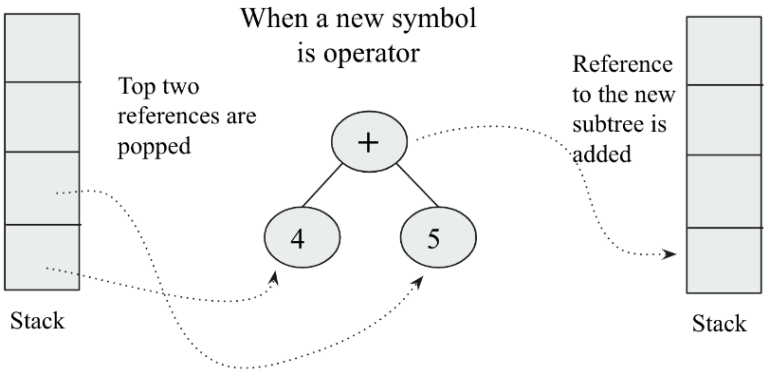

Figure 6.19: Operator + is processed in creating an expression tree

The next symbols are 5 and 3, and they are pushed into the stack. Next, when a new symbol is an operator (-), it is created as the root of the new subtree, and two top references are popped and added to the right and left child of this root respectively, as shown in *Figure 6.20*. Then, the reference to this subtree is pushed to the stack:

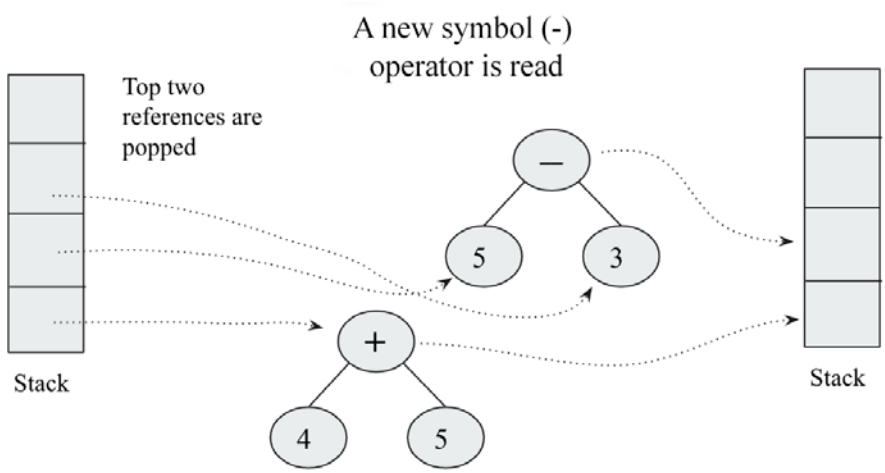

Figure 6.20: Operator (-) is processed in creating an expression tree

The next symbol is the operator *; as we have done so far, this will be created as the root, and then two references will be popped from the stack, as shown in *Figure 6.21*. The final tree is then shown in *Figure 6.21*:

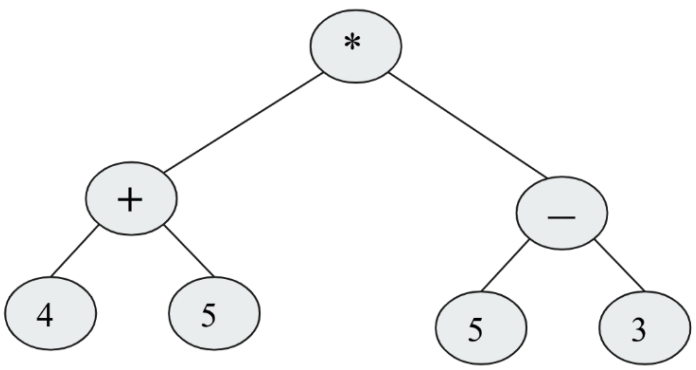

Figure 6.21: Operator (*) is processed in creating an expression tree

To learn how to implement this algorithm in Python, we will look at building a tree for an expression written in postfix notation. For this, we need a tree node implementation; it can be defined as follows:

```
class TreeNode:
    def __init__(self, data=None):
        self.data = data
        self.right = None
        self.left = None
```

The following is the code for the implementation of the stack class that we will be using:

```
class Stack:
    def __init__(self):
        self.elements = []

    def push(self, item):
        self.elements.append(item)

    def pop(self):
        return self.elements.pop()
```

In order to build the tree, we are going to enlist the items with the help of a stack. Let's take an example of an arithmetic expression and set up our stack:

```
expr = "4 5 + 5 3 - *".split()
stack = Stack()
```

In the first statement, the `split()` method splits on whitespace by default. The `expr` is a list with the values `4`, `5`, `+`, `5`, `3`, `-`, and `*`.

Each element of the `expr` list is going to be either an operator or an operand. If we get an operand, then we embed it in a tree node and push it onto the stack. If we get an operator, we embed the operator into a tree node and pop its two operands into the node's right and left children. Here, we have to take care to ensure that the first `pop` reference goes into the right child.

In continuation of the previous code snippet, the below code is a loop to build the tree:

```
for term in expr:
    if term in "+-*/":
        node = TreeNode(term)
        node.right = stack.pop()
        node.left = stack.pop()
    else:
        node = TreeNode(int(term))
    stack.push(node)
```

Notice that we perform a conversion from `string` to `int` in the case of an operand. You could use `float()` instead, if you wish to support floating-point operands.

At the end of this operation, we should have one single element in the stack, and that holds the full tree.

If we want to evaluate the expression, we can use the following function:

```
def calc(node):
    if node.data == "+":
        return calc(node.left) + calc(node.right)
    elif node.data == "-":
        return calc(node.left) - calc(node.right)
    elif node.data == "*":
        return calc(node.left) * calc(node.right)
    elif node.data == "/":
        return calc(node.left) / calc(node.right)
    else:
        return node.data
```

In the preceding code, we pass a node to the function. If the node contains an operand, then we simply return that value. If we get an operator, then we perform the operation that the operator represents on the node's two children. However, since one or more of the children could also contain either operators or operands, we call the `calc()` function recursively on the two child nodes (bearing in mind that all the children of every node are also nodes).

Now, we just need to pop the root node off the stack and pass it onto the `calc()` function. Then, we should have the result of the calculation:

```
root = stack.pop()
result = calc(root)
print(result)
```

Running this program should yield the result `18`, which is the result of `(4 + 5) * (5 - 3)`.

Expression trees are very useful in representing and evaluating complex expressions easily. It is also useful to evaluate the postfix, prefix, and infix expression. It can be used to find out the associativity of the operators in the given expression.

In the next section, we will discuss the binary search tree, which is a special kind of binary tree.

Binary search trees

A **binary search tree** (**BST**) is a special kind of binary tree. It is one of the most important and commonly used data structures in computer science applications. A binary search tree is a tree that is structurally a binary tree, and stores data in its nodes very efficiently. It provides very fast search, insertion, and deletion operations.

A binary tree is called a binary search tree if the value at any node in the tree is greater than the values in all the nodes of its left subtree, and less than (or equal to) the values of all the nodes of the right subtree. For example, if `K1`, `K2`, and `K3` are key values in a tree of three nodes (as shown in *Figure 6.22*), then it should satisfy the following conditions:

- The key values *K2<=K1*
- The key values *K3>K1*

The following figure depicts the above condition of the binary search tree:

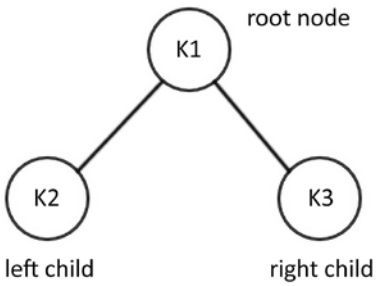

Figure 6.22: An example of a binary search tree

Let's consider another example so that we have a better understanding of binary search trees. Consider the binary search tree shown in *Figure 6.23*:

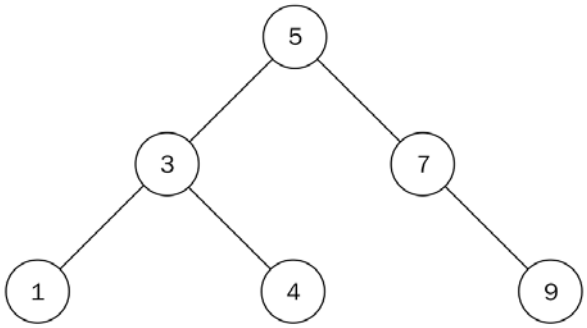

Figure 6.23: Binary search tree of six nodes

In this tree, all the nodes in the left subtree are less than (or equal to) the value of the parent node. All the nodes in the right subtree of this node are also greater than that of the parent node.

To see if the above example tree fulfills the properties of a binary search tree, we see that all the nodes in the left subtree of the root node have a value less than 5. Likewise, all the nodes in the right subtree have a value that is greater than 5. This property applies to all the nodes in the tree with no exceptions. For example, if we take another node with the value 3, we can see that the values for all the left subtree nodes are less than the value 3 and the values for all the right subtree nodes are greater than 3.

Considering another example of a binary tree. Let's check to see if it is a binary search tree. Despite the fact that the following diagram, *Figure 6.24*, looks similar to the previous diagram, it does not qualify as a binary search tree, as node 7 is greater than the root node 5; even though it is located in the left subtree of the root node. Node 4 is to the right subtree of its parent node 7, which is also violating a rule of binary search trees. Thus, the following figure, *Figure 6.24*, is not a binary search tree:

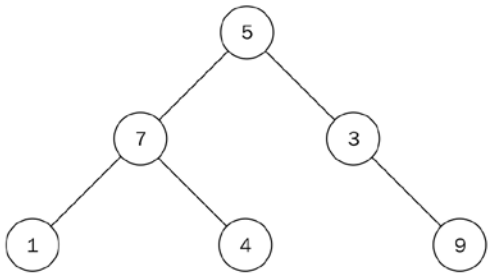

Figure 6.24: An example of a binary tree that is not a binary search tree

Let's begin the implementation of a binary search tree in Python. Since we need to keep track of the root node of the tree, we start by creating a `Tree` class that holds a reference to the root node:

```
class Tree:
    def __init__(self):
        self.root_node = None
```

That's all it takes to maintain the state of a tree. Now, let's examine the main operations used within the binary seach tree.

Binary search tree operations

The operations that can be performed on a binary search tree are `insert`, `delete`, `finding min`, `finding max`, and `searching`. We discuss them in detail one by one in the following sections.

Inserting nodes

One of the most important operations to implement on a binary search tree is to insert data items in the tree. In order to insert a new element into a binary search tree, we have to ensure that the properties of the binary search tree are not violated after adding the new element.

In order to insert a new element, we start by comparing the value of the new node with the root node: if the value is less than the root value, then the new element will be inserted into the left subtree; otherwise, it will be inserted into the right subtree. In this manner, we go to the end of the tree to insert the new element.

Let's create a binary search tree by inserting data items 5, 3, 7, and 1 in the tree. Consider the following:

1. **Insert 5**: We start with the first data item, 5. To do this, we will create a node with its data attribute set to 5, since it is the first node.
2. **Insert 3**: Now, we want to add the second node with a value of 3 so that the data value of 3 is compared with the existing node value, 5, of the root node. Since the node value 3 is less than 5, it will be placed in the left subtree of node 5. The binary search tree will look as shown in *Figure 6.25*:

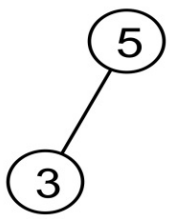

Figure 6.25: Step 2 of the insertion operation in an example binary search tree

   Here, the tree satisfies the binary search tree rule, where all the nodes in the left subtree are less than the parent.

3. **Insert 7**: To add another node with a value of 7 to the tree, we start from the root node with value 5 and make a comparison, as shown in *Figure 6.26*. Since 7 is greater than 5, the node with a value of 7 is placed to the right of this root:

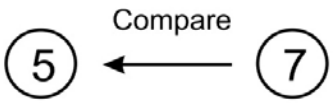

Figure 6.26: Step 3 of the insertion operation in an example binary search tree

4. **Insert 1**: Next, we add another node with the value 1. Starting from the root of the tree, we make a comparison between 1 and 5, as shown in *Figure 6.27*:

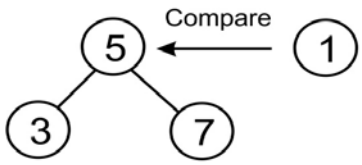

Figure 6.27: Step 4 of the insertion operation in an example binary search tree

This comparison shows that 1 is less than 5, so we go to the left subtree of 5, which has a node with a value of 3, as shown in *Figure 6.28*:

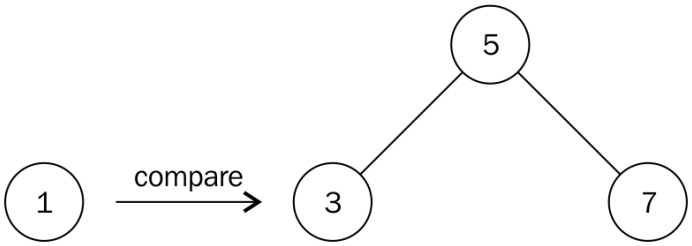

Figure 6.28: Comparison of node 1 and node 3 in an example binary search tree

When we compare 1 against 3, 1 is less than 3, so we move a level below node 3 and to its left, as shown in *Figure 6.28*. However, there is no node there. Therefore, we create a node with a value of 1 and associate it with the left pointer of node 3 to obtain the final tree. Here, we have the final binary search tree of 4 nodes, as shown in *Figure 6.29*:

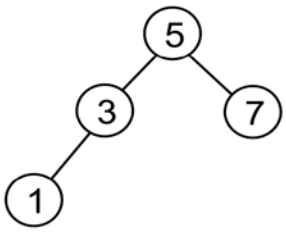

Figure 6.29: Final step of the insertion operation in an example binary search tree

We can see that this example contains only integers or numbers. So, if we need to store string data in a binary search tree, the strings would be compared alphabetically.

If we wanted to store any custom data types inside a binary search tree, we would have to make sure that the binary search tree class supports ordering.

The Python implementation of the `insert` method to add the nodes in the binary search tree is given as follows:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.right_child = None
        self.left_child = None

class Tree:
    def __init__(self):
        self.root_node = None

    def insert(self, data):

        node = Node(data)
        if self.root_node is None:
            self.root_node = node
            return self.root_node
        else:
            current = self.root_node
            parent = None
            while True:
                parent = current
                if node.data < parent.data:
                    current = current.left_child
                    if current is None:
                        parent.left_child = node
                        return self.root_node
                else:
                    current = current.right_child
                    if current is None:
                        parent.right_child = node
                        return self.root_node
```

In the above code, we first declare the `Node` class with the `Tree` class. All the operations that can be applied to the tree are defined in the `Tree` class. Let's understand the steps of the `insert` method. We begin with a function declaration:

```
    def insert(self, data):
```

Next, we encapsulate the data in a node using the `Node` class. We check whether we have a root node or not. If we don't have a root node in the tree, the new node becomes the root node and then root node is returned:

```
        node = Node(data)
        if self.root_node is None:
            self.root_node = node
            return self.root_node
        else:
```

Further, in order to insert a new element, we have to traverse the tree and reach the correct position where we can insert the new element in a way that the properties of the binary search tree are not violated. For this, we keep track of the current node while traversing the tree as well as its parent. The `current` variable is always used to track where a new node will be inserted:

```
        current = self.root_node
        parent = None
        while True:
            parent = current
```

Here, we must perform a comparison. If the data held in the new node is less than the data held in the current node, then we check whether the current node has a left child node. If it doesn't, this is where we insert the new node. Otherwise, we keep traversing:

```
            if node.data < parent.data:
                current = current.left_child
                if current is None:
                    parent.left_child = node
                    return self.root_node
```

After this, we need to take care of the greater than (or equal to) case. If the current node doesn't have a right child node, then the new node is inserted as the right child node. Otherwise, we move down and continue looking for an insertion point:

```
            else:
                current = current.right_child
```

```
                if current is None:
                    parent.right_child = node
                    return self.root_node
```

Now, in order to see what we have inserted in the binary search tree, we can use any of the existing tree traversal algorithms. Let's implement the in-order traversal, which should be defined in the `Tree` class. The code is as follows:

```
    def inorder(self, root_node):
        current = root_node
        if current is None:
            return
        self.inorder(current.left_child)
        print(current.data)
        self.inorder(current.right_child)
```

Now, let us take an example to insert a few elements (e.g. elements `5`, `2`, `7`, `9`, and `1`) in a binary search tree, as shown in *Figure 6.24*, and then we can use the in-order traversal algorithm to see what we have inserted in the tree:

```
tree = Tree()

r = tree.insert(5)
r = tree.insert(2)
r = tree.insert(7)
r = tree.insert(9)
r = tree.insert(1)

tree.inorder(r)
```

The output of the above code is as follows:

```
1
2
5
7
9
```

Insertion of a node in a binary search tree takes `O(h)`, where `h` is the height of the tree.

Searching the tree

A binary search tree is a tree data structure in which all the nodes in the left subtree of a node have lower key values and the right subtree has greater key values. Thus, searching for an element with a given key value is quite easy. Let's consider an example binary search tree that has nodes `1`, `2`, `3`, `4`, `8`, `5`, and `10`, as shown in *Figure 6.30*:

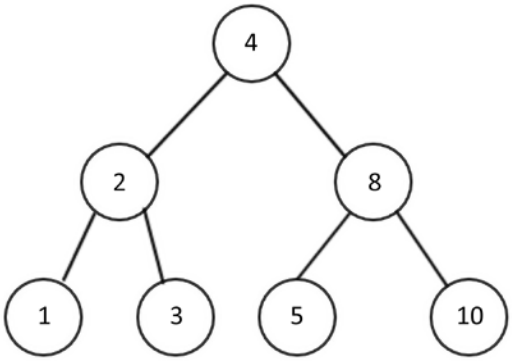

Figure 6.30: An example binary search tree with seven nodes

In the preceding tree shown in *Figure 6.30*, if we wish to search for a node with a value of `5`, for example, then we start from the root node and compare the root with our desired value. As node `5` is a greater value than the root node's value of `4`, we move to the right subtree. In the right subtree, we have node `8` as the root node, so we compare node `5` with node `8`. As the node to be searched has a smaller value than node `8`, we move it to the left subtree. When we move to the left subtree, we compare the left subtree node `5` with the required node value of `5`. This is a match, so we return `"item found"`.

Here is the implementation of the searching method in a binary search tree, which is being defined in the `Tree` class:

```
    def search(self, data):
        current = self.root_node
        while True:
            if current is None:
                print("Item not found")
                return None
            elif current.data is data:
                print("Item found", data)
                return data
```

```
            elif current.data > data:
                current = current.left_child
            else:
                current = current.right_child
```

In the preceding code, we return the data if it was found, or `None` if the data wasn't found. We start searching from the root node. Next, if the data item to be searched for doesn't exist in the tree, we return `None`. If we find the data, it is returned.

If the data that we are searching for is less than that of the current node, we go down the tree to the left. Furthermore, in the `else` part of the code, we check if the data we are looking for is greater than the data held in the current node, which means that we go down the tree to the right.

Finally, the below code can be used to create an example binary search tree with some values between 1 and 10. Then, we search for a data item with the value 9, and also all the numbers in that range. The ones that exist in the tree get printed:

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)

tree.search(9)
```

The output of the above code is as follows:

```
Item found 9
```

In the above code, we see the items that were present in the tree have been correctly found; the rest of the items could not be found in the range 1 to 10. In the next section, we discuss the deletion of a node in binary search tree.

Deleting nodes

Another important operation on a binary search tree is the deletion or removal of nodes. There are three possible scenarios that we need to take care of during this process. The node that we want to remove might have the following:

- **No children**: If there is no leaf node, directly remove the node

- **One child**: In this case, we swap the value of that node with its child, and then delete the node
- **Two children**: In this case, we first find the in-order successor or predecessor, swap their values, and then delete that node

The first scenario is the easiest to handle. If the node about to be removed has no children, we can simply remove it from its parent. In *Figure 6.31*, suppose we want to delete node `A`, which has no children. In this case, we can simply delete it from its parent (node `Z`):

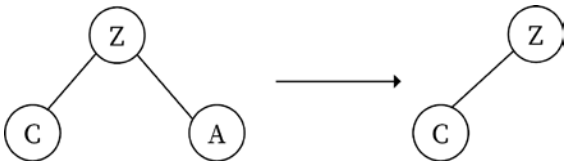

Figure 6.31: Deletion operation when deleting a node with no children

In the second scenario, when the node we want to remove has one child, the parent of that node is made to point to the child of that particular node. Let's take a look at the following diagram, where we want to delete node 6, which has one child, node 5, as shown in *Figure 6.32*:

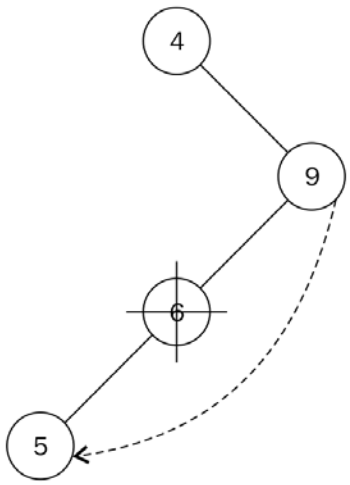

Figure 6.32: Deletion operation when deleting a node with one child

In order to delete node 6, which has node 5 as its only child, we point the left pointer of node 9 to node 5. Here, we need to ensure that the child and parent relationship follows the properties of a binary search tree.

In the third scenario, when the node we want to delete has two children, in order to delete it, we first find a successor node, then move the content of the successor node into the node to be deleted. The successor node is the node that has the minimum value in the right subtree of the node to be deleted; it will be the first element when we apply the in-order traversal on the right subtree of the node to be deleted.

Let's understand it with the example tree shown in *Figure 6.33*, where we want to delete node 9, which has two children:

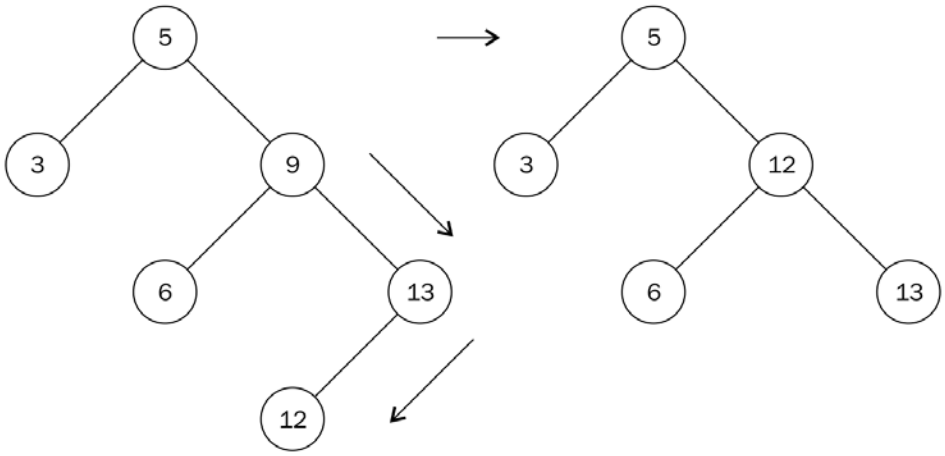

Figure 6.33: Deletion operation when deleting a node with two children

In the example tree shown in *Figure 6.33*, we find the smallest element in the right subtree of the node (i.e. the first element in the in-order traversal in the right subtree) which is node 12. After that, we replace the value of node 9 with the value 12 and remove node 12. Node 12 has no children, so we apply the rule for removing nodes without children accordingly.

To implement the above algorithm using Python, we need to write a helper method to get the node that we want to delete along with the reference to its parent node. We have to write a separate method because we do not have any reference to the parent in the `Node` class. This helper method `get_node_with_parent` is similar to the `search` method, which finds the node to be deleted, and returns that node with its parent node:

```
    def get_node_with_parent(self, data):
        parent = None
        current = self.root_node
        if current is None:
```

```
            return (parent, None)
        while True:
            if current.data == data:
                return (parent, current)
            elif current.data > data:
                parent = current
                current = current.left_child
            else:
                parent = current
                current = current.right_child

        return (parent, current)
```

The only difference is that before we update the current variable inside the loop, we store its parent with `parent = current`. The method to do the actual removal of a node begins with this search:

```
    def remove(self, data):
        parent, node = self.get_node_with_parent(data)

        if parent is None and node is None:
            return False

        # Get children count
        children_count = 0

        if node.left_child and node.right_child:
            children_count = 2
        elif (node.left_child is None) and (node.right_child is None):
            children_count = 0
        else:
            children_count = 1
```

We pass the parent and the found nodes to `parent` and `node`, respectively, with the `parent, node = self.get_node_with_parent(data)` line. It is important to know the number of children that the node has that we want to delete, and we do so in the `if` statement.

Once we know the number of children a node has that we want to delete, we need to handle various conditions in which a node can be deleted. The first part of the `if` statement handles the case where the node has no children:

```
        if children_count == 0:
            if parent:
                if parent.right_child is node:
                    parent.right_child = None
                else:
                    parent.left_child = None
            else:
                self.root_node = None
```

In cases where the node to be deleted has only one child, the `elif` part of the `if` statement does the following:

```
        elif children_count == 1:
            next_node = None
            if node.left_child:
                next_node = node.left_child
            else:
                next_node = node.right_child

            if parent:
                if parent.left_child is node:
                    parent.left_child = next_node
                else:
                    parent.right_child = next_node
            else:
                self.root_node = next_node
```

`next_node` is used to keep track of that single node, which is the child of the node that is to be deleted. We then connect `parent.left_child` or `parent.right_child` to `next_node`.

Lastly, we handle the condition where the node we want to delete has two children:

```
        else:
            parent_of_leftmost_node = node
            leftmost_node = node.right_child
            while leftmost_node.left_child:
                parent_of_leftmost_node = leftmost_node
                leftmost_node = leftmost_node.left_child

            node.data = leftmost_node.data
```

In finding the in-order successor, we move to the right node with `leftmost_node = node.right_child`. As long as a left node exists, `leftmost_node.left_child` will be `True` and the `while` loop will run. When we get to the leftmost node, it will either be a leaf node (meaning that it will have no child node) or have a right child.

We update the node that's about to be removed with the value of the in-order successor with `node.data = leftmost_node.data`:

```
            if parent_of_leftmost_node.left_child == leftmost_node:
                parent_of_leftmost_node.left_child = leftmost_node.right_
child
            else:
                parent_of_leftmost_node.right_child = leftmost_node.right_
child
```

The preceding statement allows us to properly attach the parent of the leftmost node with any child node. Observe how the right-hand side of the equals sign stays unchanged. This is because the in-order successor can only have a right child as its only child.

The following code demonstrates how to use the remove method in the `Tree` class:

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)

tree.search(9)
tree.remove(9)
tree.search(9)
```

The output of the above code is:

```
Item found 9
Item not found
```

In the above code, when we search for item `9`, it is available in the tree, and after the remove method, item `9` is not present in the tree. In the worst-case scenario, the `remove` operation takes `O(h)`, where `h` is the height of the tree.

Finding the minimum and maximum nodes

The structure of the binary search tree makes searching a node that has a maximum or a minimum value very easy. To find a node that has the smallest value in the tree, we start traversal from the root of the tree and visit the left node each time until we reach the end of the tree. Similarly, we traverse the right subtree recursively until we reach the end to find the node with the biggest value in the tree.

For example, consider *Figure 6.34*, in order to search for the minimum and maximum elements.

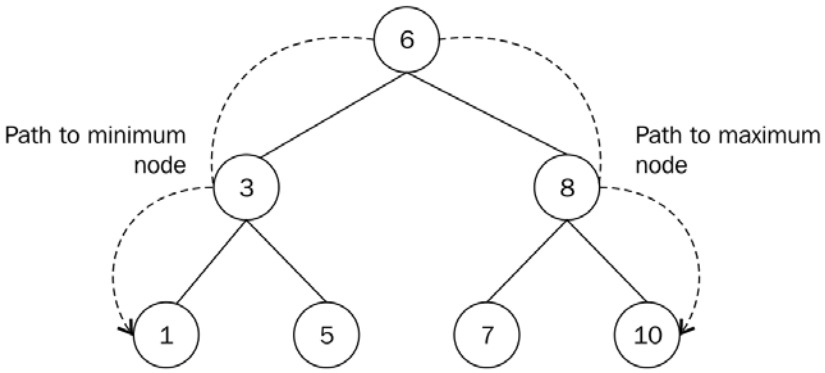

Figure 6.34: Finding the minimum and maximum nodes in a binary search tree

Here, we start by moving down the tree from root node `6` to `3`, and then from node `3` to `1` to find the node with the smallest value. Similarly, to find the maximum value node from the tree, we go down from the root along the right-hand side of the tree, so we go from node `6` to node `8` and then node `8` to node `10` to find the node with the largest value.

The Python implementation of the method that returns the minimum value of any node is as follows:

```
    def find_min(self):
        current = self.root_node
        while current.left_child:
            current = current.left_child

        return current.data
```

The `while` loop continues to get the left node and visits it until the last left node points to `None`. It is a very simple method.

Similarly, the following is the code of the method that returns the maximum node:

```
    def find_max(self):
        current = self.root_node
        while current.right_child:
            current = current.right_child

        return current.data
```

The following code demonstrates how to use the `find_min` and `find_max` methods in the `Tree` class:

```
tree = Tree()
tree.insert(5)
tree.insert(2)
tree.insert(7)
tree.insert(9)
tree.insert(1)
print(tree.find_min())
print(tree.find_max())
```

The output of the above code is as shown below:

```
1
9
```

The output of the above code, `1` and `9`, are the minimum and maximum values. The minimum value in the tree is `1` and the maximum is `9`. The running time complexity to find the minimum or maximum value in a binary search tree is `O(h)`, where `h` is the height of the tree.

Benefits of a binary search tree

A binary search tree is, in general, a better choice compared to arrays and linked lists when we are mostly interested in accessing the elements frequently in any application. A binary search tree is fast for most operations, such as searching, insertion, and deletion, whereas arrays provide fast searching, but are comparatively slow regarding insertion and deletion operations. In a similar fashion, linked lists are efficient in performing insertion and deletion operations, but are slower when performing the search operation. The best-case running time complexity for searching an element from a binary search tree is `O(log n)`, and the worst-case time complexity is `O(n)`, whereas both best-case and worst-case time complexity for searching in lists is `O(n)`.

The following table provides a comparison of the array, linked list, and binary search tree data structures:

Properties	**Array**	**Linked list**	**BST**
Data structure	Linear.	Linear.	Non-linear.
Ease of use	Easy to create and use. Average-case complexity for search, insert, and delete is `O(n)`.	Insertion and deletion are fast, especially with the doubly linked list.	Access of elements, insertion, and deletion is fast with the average-case complexity of `O(log n)`.
Access complexity	Easy to access elements. Complexity is `O(1)`.	Only sequential access is possible, so slow. Average- and worst-case complexity are `O(n)`.	Access is fast, but slow when the tree is unbalanced, with a worst-case complexity of `O(n)`.
Search complexity	Average- and worst-case complexity are `O(n)`.	It is slow due to sequential searching. Average- and worst-case complexity are `O(n)`.	Worst-case complexity for searching is `O(n)`.
Insertion complexity	Insertion is slow. Average- and worst-case complexity are `O(n)`.	Average- and worst-case complexity are `O(1)`.	The worst-case complexity for insertion is `O(n)`.
Deletion complexity	Deletion is slow. Average- and worst-case complexity are `O(n)`.	Average- and worst-case complexity are `O(1)`.	The worst-case complexity for deletion is `O(n)`.

Let's consider an example to understand when the binary search tree is a good choice to store the data. Let's assume that we have the following data nodes—5, 3, 7, 1, 4, 6, and 9, as shown in *Figure 6.35*. If we use a list to store this data, the worst-case scenario will require us to search through the entire list of seven elements to find the item. So, it will require six comparisons to search for item 9 in this data node, as shown in *Figure 6.35*:

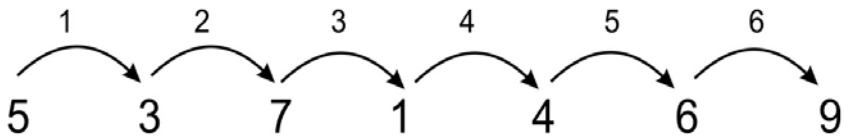

Figure 6.35: An example list of seven elements requires six comparisons if stored in a list

However, if we use a binary search tree to store these values, as shown in the following diagram, in the worst-case scenario, we will require two comparisons to search for item 9, as shown in *Figure 6.36*:

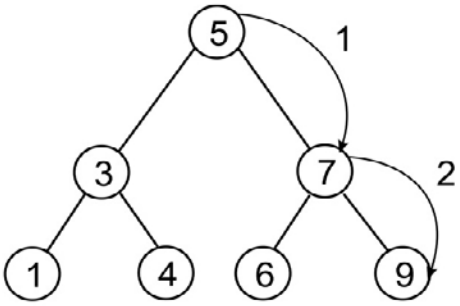

Figure 6.36: An example list of seven elements requires three comparisons if stored in a binary search tree

However, it is important to note that the efficiency of searching also depends on how we built the binary search tree. If the tree hasn't been constructed properly, it can be slow. For example, if we had inserted the elements into the tree in the order 1, 3, 4, 5, 6, 7, 9, as shown in *Figure 6.37*, then the tree would not be more efficient than the list:

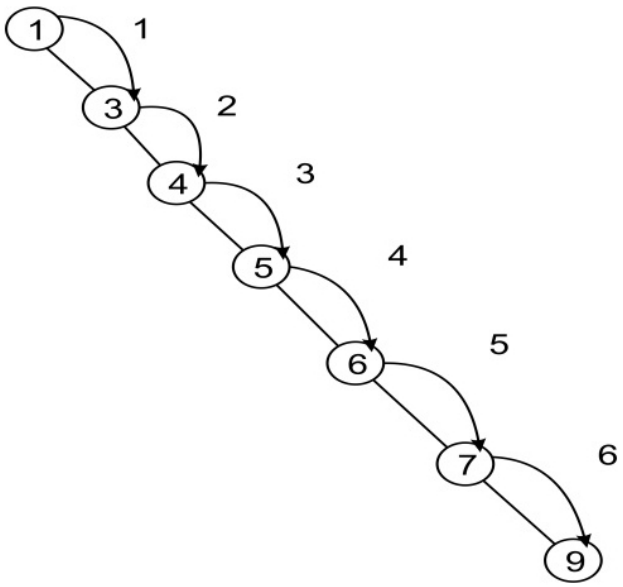

Figure 6.37: A binary search tree constructed with elements in the order 1, 3, 4, 5, 6, 7,9

Depending upon the sequence of the nodes added to the tree, it is possible that we may have a binary tree that is unbalanced. Thus, it is important to use a method that can make the tree a self-balancing tree, which in turn will improve the `search` operation. Therefore, we should note that a binary search tree is a good choice if the binary tree is balanced.

Summary

In this chapter, we discussed an important data structure, i.e. tree data structures. Tree data structures in general provide better performance compared to linear data structures in `search`, `insert`, and `deletion` operations. We have also discussed how to apply various operations to tree data structures. We studied binary trees, which can have a maximum of two children for each node. Further, we learned about binary search trees and discussed how we can apply different operations to them. Binary search trees are very useful when we want to develop a real-world application in which the retrieval or searching of data elements is an important operation. We need to ensure that the tree is balanced for the good performance of binary search tree. We will discuss priority queues and heaps in the next chapter.

Exercises

1. Which of the following is a true about binary trees:

    a. Every binary tree is either complete or full
    b. Every complete binary tree is also a full binary tree
    c. Every full binary tree is also a complete binary tree
    d. No binary tree is both complete and full
    e. None of the above

2. Which of the tree traversal algorithms visit the root node last?

    Consider this binary search tree:

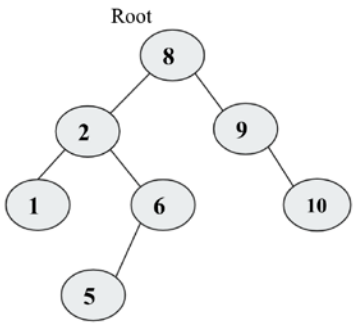

Figure 6.38: Sample binary search tree

3. Suppose we remove the root node 8, and we wish to replace it with any node from the left subtree, then what will be the new root?
4. What will be the `inorder`, `postorder` and `preorder` traversal of the following tree?

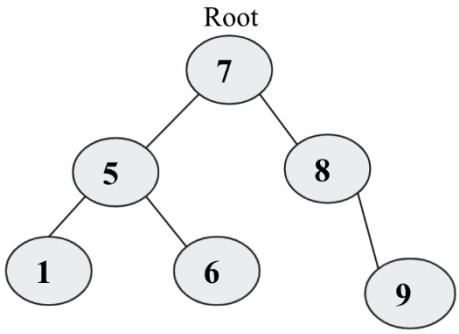

Figure 6.39: Example tree

5. How do you find out if two trees are identical?
6. How many leaves are there in the tree mentioned in *question number 4*?
7. What is the relation between a perfect binary tree's height and the number of nodes in that tree?

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:
`https://packt.link/MEvK4`