2

Good Things Come in Pairs

Modeling hierarchical relationships with binary trees

All the previous data structures that we have explored so far have been *linearly* organized. They are good at keeping a set of elements that we need to access in a certain order. Each element is accessed by its position, either in terms of an index within the data structure, or in terms of its relative position in the top/bottom of a stack, or the head/tail of a queue.

In some applications what we need is a data structure that organizes elements based on their relationship to one another. For example, if we want to represent an organizational chart, we need to keep the information of who reports to whom. To represent a family tree, we need to know who are the parents of whom. To represent an inter-city road network, we need to record which city can be reached from which other city.

A data structure that can model such relationships are called *non-linear* data structures. In this course, we will cover two such structures. In this chapter, we will model *hierarchical* relationships with *trees*, and more specifically *binary trees*. In the next chapter (Chapter 3), we will model non-hierarchical relationships using graphs.

2.1 Tree

Trees are used to represent hierachical relationships. Figure 2.1 shows an organizational chart of a company, which can be modeled as a tree. Each element in a tree is called a **node** (e.g., CEO, VP Operations).

Nodes in a tree are related to one another. A node n_1 (e.g., CEO) is the **parent** of another node n_2 (e.g., VP Operations), if there is an edge between the two nodes, and n_1 is above n_2 in the tree. In this case, n_2 is a **child** of n_1. Every node can have zero or more children, but at most one parent. Nodes with the same parent are **siblings** of one another.

If we extend this relationship through multiple hops, we say a node n_1 is an **ancestor** of another node n_3 (or equivalently n_3 is a **descendant** of n_1), if there exists an unbroken path from n_1, to a child of n_1, to a child of this child, and so on, and finally to n_3. For example, in Figure 2.1, Manager (HR) is a descendant of CEO.

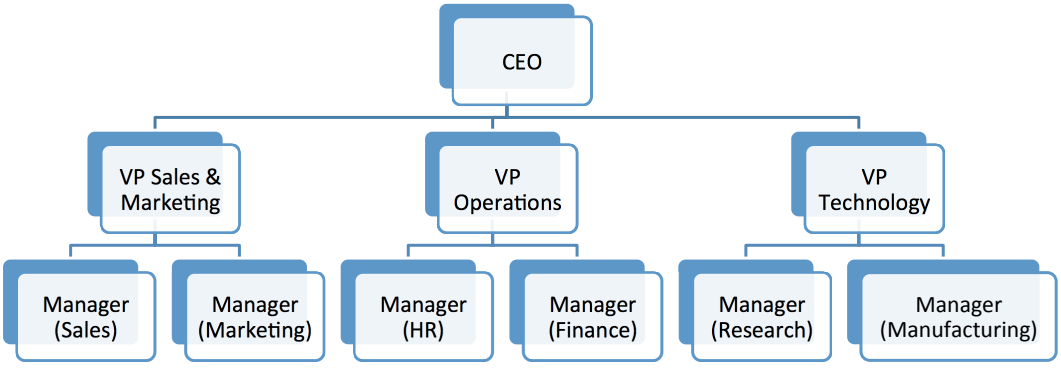


Figure 2.1: *Example of Tree Structure: Organization Chart*

A **tree** consists of a node and all its descendant. The ultimate ancestor node is called the **root**. Nodes without children, at the lowest levels of the tree, are called **leaf** nodes. Note the irony, and contrast to real-life trees, of having a tree with roots at the top, and leaves at the bottom. Figure 2.1 is a tree with CEO as the root node. The managers occupy the leaf nodes of this tree.

The root node is at **level** 1 of the tree. Its children are at level 2, and so on. The **height** of a tree is the maximum level of any leaf node. In Figure 2.1, the height of the tree is 3.

A tree can have many **subtrees**, where again each subtree consists of a node and all its descendants in the original tree. For instance, in Figure 2.1, each VP is the root of its own subtree.

Because each child of the root node is itself the root of its own subtree, we can have the following **recursive definition of a tree**. A tree consists of a root node n, and a number of subtrees rooted at the children of the root node n.

2.2 Binary Tree

In a general tree, a parent can have any number of children. In this section, we focus on a specific type of tree, called *binary tree*, where each parent can have at most two children, which we refer to as the **left child** and the **right child**. Hence, a binary tree is defined recursively as consisting of a root node, a left binary subtree and a right binary subtree.

Figure 2.2 illustrates an example of a binary tree. Mrs. Goodpairs, the CEO, believes that any officer of the company should not have more than two direct reports. She herself has two vice presidents, VP1 and VP2, reporting to her. Each vice president can have up to two managers. In this case, M1 and M2 report to VP1, and M3 reports to VP2.

A binary tree is said to be **full** if all the nodes, except the leaf nodes, have two children each, and all the leaf nodes are at the same level. Figure 2.2 is not a full binary tree.

A binary tree is said to be **complete** if all its nodes at level `h-2` and above have two children each. In addition, when a node at level `h-1` has children, it implies that all the nodes at the same level to its left have two children each. When a node at level `h-1` has only one child, it is a left child. Figure 2.2 is a complete binary tree.

A binary tree supports the following operations:

- `new` creates an empty binary tree with no root

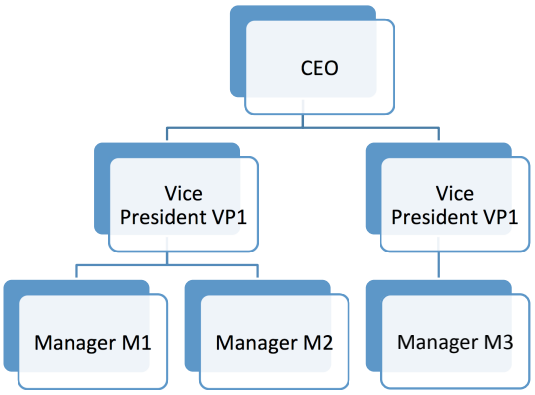


Figure 2.2: *Example of Binary Tree*

- `setRoot(node)` sets a node passed as input to be the root of the tree
- `root` returns the node, which is the root of the tree
- `noOfNodes` returns the number of nodes in the tree
- `heightOfTree` returns the height of the tree
- `isEmpty?` checks whether the tree has any node

Importantly, each node `n` in the tree also supports specific operations:

- `new(value)` creates a new node with a specific value passed in as argument
- `left` returns the left child
- `right` returns the right child
- `setLeft(node)` sets the node passed as input to be the left child
- `setRight(node)` sets the node passed as input to be the right child
- `delLeft` removes the current left child
- `delRight` removes the current right child

Let us now try our hands on a tutorial project on IRB, which will get us familiar with these operations of a binary tree.

Tutorial Project

Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'is103'
=>  true

>>  include TreeLab
=>  Object
```

T1. Make a new node `r` that we will use as the root of our first binary tree:

```
>>  r = Node.new("CEO")
=>  Value: CEO, left child: nil, right child: nil
```

Note that when we create the node `r`, we give it a value "CEO". The output prints out this value. Currently, the left child and the right child are `nil`, because we have not set them yet.

T2. Let's provide `r` with a couple of children nodes `n1` and `n2`, which we will set as the left and right children respectively.

```
>>  n1 = Node.new("VP1")
=>  Value: VP1, left child: nil, right child: nil

>>  r.setLeft(n1)
=>  Value: CEO, left child: VP1, right child: nil

>>  n2 = Node.new("VP2")
=>  Value: VP2, left child: nil, right child: nil

>>  r.setRight(n2)
=>  Value: CEO, left child: VP1, right child: VP2
```

T3. Let us check that the node `r` now has a left child and a right child.

```
>>  r.left
=>  Value: VP1, left child: nil, right child: nil

>>  r.right
=>  Value: VP2, left child: nil, right child: nil
```

T4. We now create a new binary tree `t`, and set `r` as the root node of this tree.

```
>>  t = BinaryTree.new

>>  t.setRoot(r)
=>  Value: CEO, left child: VP1, right child: VP2
```

T5. Let's double check that the root has indeed been set to `r`.

```
>>  t.root
=>  Value: CEO, left child: VP1, right child: VP2
```

T6. To make it easier to understand the current structure of the tree, we can visualize it.

```
>>  view_tree(t)
=>  true
```

T7. Currently, the nodes `n1` and `n2` do not have any child. Let's practice adding a few more nodes into the tree.

```
>>  n3 = Node.new("M1")
=>  Value: M1, left child: nil, right child: nil

>>  n1.setLeft(n3)
=>  Value: VP1, left child: M1, right child: nil

>>  n4 = Node.new("M2")
=>  Value: M2, left child: nil, right child: nil

>>  n1.setRight(n4)
=>  Value: VP1, left child: M1, right child: M2
```

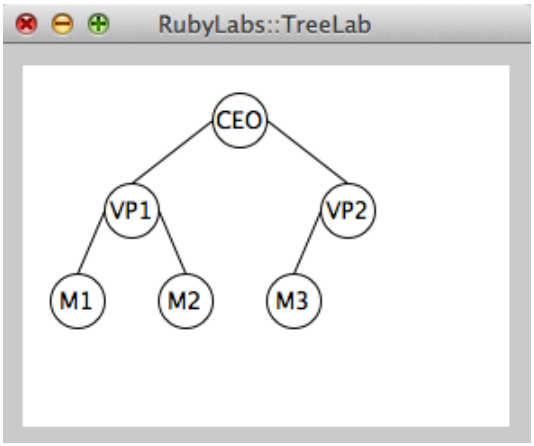


Figure 2.3: *Visualization of tree after the first tutorial project*

```
>>  n5 = Node.new("M3")
=>  Value: M3, left child: nil, right child: nil

>>  n2.setLeft(n5)
=>  Value: VP2, left child: M3, right child: nil

>>  n6 = Node.new("M4")
=>  Value: M4, left child: nil, right child: nil

>>  n2.setRight(n6)
=>  Value: VP2, left child: M3, right child: M4
```

Look at the tree visualization. Is the structure of the tree as expected?

T8. Sometimes, we may need to remove a node from the tree. For example, the company may be downsized, and the manager "M4" may be let go.

```
>>  n2.delRight
=>  true
```

Note that to remove a node, we simply call `delLeft` or `delRight` method from the parent of the node to be removed. In this case, "M4" is node `n6`, which is the right child of node `n2`.

T9. Do you see how the visualization changes to indicate that "M4" is no longer part of the tree?

The visualization of the tree up to this step is shown in Figure 2.3. Does yours look the same?

T10. We can also find out the height of this tree, as well as the number of nodes.

```
>>  t.heightOfTree
=>  3

>>  t.noOfNodes
=>  6
```

T11. Try a few more tree operations on your own.

2.3 Traversals of a Binary Tree

Suppose that we would like to examine each element in the data structure exactly once. This is useful for many applications, such as searching for an element of a specific value. This process is called *traversing* a data structure.

For linear data structures, there is usually only one way, which is to iterate through the index positions of all the elements. However, in a non-linear data structure, we traverse elements by their relationships, and therefore there could be more than one traversal method.

During traversal, each node is visited exactly once. The node is "processed" during the visit. For instance, during a search, visiting a node may mean checking for its value.

For binary trees in particular, there are three common traversal methods: *preorder*, *inorder*, *postorder*, which we will explore in the following.

preorder

In preorder traversal, we visit a node first, followed by visiting its left child, and then its right child.

An implementation of preorder traversal is shown in Figure 2.4(a). It is expressed in terms of a recursive algorithm. The root node is visited *before* the recursive calls to the left and right subtrees.

For an example of preorder traversal, we will traverse the same tree in Figure 2.3 that we created earlier in the previous tutorial project. We call `preorder_traverse` on the root node, which is "CEO". This node is visited first (no. 1). Next, we will visit the left subtree, rooted at "VP1" (no. 2). Recursively, we will visit the left subtree of "VP1", rooted at "M1" (no. 3), followed by the right subtree of "VP1", rooted at "M2" (no. 4). Since all the nodes in the left subtree of "CEO" have been visited, we now turn to the right subtree rooted at "VP2" (no. 5), which recursively calls its own left subtree rooted at "M3" (no. 6).

By now we have visited all the nodes in the following order: "CEO", "VP1", "M1", "M2", "VP2", "M3". The order is indicated in Figure 2.4(a) as sequence numbers below each node.

inorder

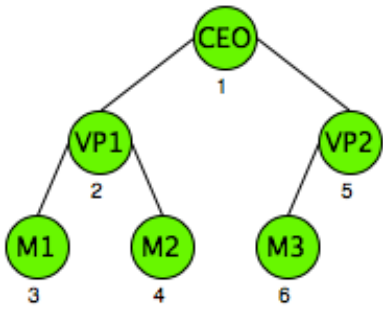
In inorder traversal, we visit the root's left subtree first, then the root, and finally its right subtree.

An implementation of inorder traversal is shown in Figure 2.4(b). It is also expressed in terms of a recursive algorithm. Note that the `visit` is called *in between* the recursive calls to the left subtree and the right subtree.

For example, we will traverse the same tree in Figure 2.3. We call `inorder_traverse` on the root node, which is "CEO". However, we do not visit this root node yet, and instead recursively traverse its left subtree rooted at "VP1". Again recursively, we traverse the left subtree of "VP1" rooted at "M1". Since "M1" has no child, there is no further recursive call, and "M1" is visited (no. 1). Having visited "M1", "VP1" is then visited (no. 2), followed by its right child "M2" (no. 3). The left subtree of "CEO" has been visited, so now we visit "CEO" itself (no. 4). This is then followed by visits to "M3" (no. 5) and "VP2" (no. 6).

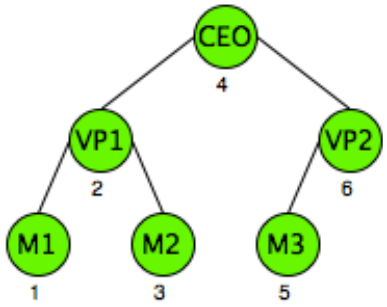
The inorder traversal sequence is "M1", "VP1", "M2", "CEO", "M3", "VP2". The order is indicated in Figure 2.4(b) as sequence numbers below each node.

```
def preorder_traverse(node)
  if (node != nil)
    visit(node)
    preorder_traverse(node.left)
    preorder_traverse(node.right)
  end
end
```



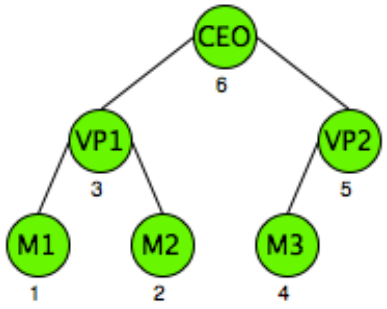
(a) preorder traversal

```
def inorder_traverse(node)
  if (node != nil)
    inorder_traverse(node.left)
    visit(node)
    inorder_traverse(node.right)
  end
end
```



(b) inorder traversal

```
def postorder_traverse(node)
  if (node != nil)
    postorder_traverse(node.left)
    postorder_traverse(node.right)
    visit(node)
  end
end
```



(c) postorder traversal

Figure 2.4: *Various Ways of Traversing a Binary Tree*

postorder

In postorder traversal, we visit the root after visiting its left and right subtrees.

An implementation of postorder traversal is shown in Figure 2.4(c) It is also expressed in terms of a recursive algorithm. Note that the `visit` is called *after* the recursive calls to the left subtree and the right subtree.

We will again traverse the tree in Figure 2.3. We call `postorder_traverse` on the root node, which is "CEO". Immediately, we do a recursive call to its left subtree rooted at "VP1", followed by another recursive call to subtree rooted at "M1". Since "M1" has no child, there is no recursive call to its children. "M1" becomes the first node visited (no. 1). This is followed by "M2" (no. 2), and then "VP1" (no. 3) after its two children have been visited. We now go to the right subtree of "CEO", visiting "M3" (no. 4), "VP2" (no. 5), and finally "CEO" (no. 6).

The postorder traversal sequence is "M1", "M2", "VP1", "M3", "VP2", "CEO". The order is indicated in Figure 2.4(c) as sequence numbers below each node.

Tutorial Project

Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'is103'
=>  true

>>  include TreeLab
=>  Object
```

To do traversal, first we need to create a binary tree. If you still have the `irb` session for the previous tutorial project, we will use that. Otherwise, you may want to re-do the first tutorial project in this chapter to create a tree first.

T12. Traverse the tree `t` using preorder:

```
>>  traversal(t, :preorder)
=>  Visited: CEO, VP1, M1, M2, VP2, M3
```

Do you see the animation in which the nodes are visited?

T13. If the animation is too fast for you, you can specify the time steps in seconds. Suppose the desired time step is 2 seconds, we call the method as follows.

```
>>  traversal(t, :preorder, 2)
=>  Visited: CEO, VP1, M1, M2, VP2, M3
```

The result is the same, but the animation goes slower now, doesn't it?

T14. Traverse the tree `t` using inorder:

```
>>  traversal(t, :inorder)
=>  Visited: M1, VP1, M2, CEO, M3, VP2
```

T15. Traverse the tree `t` using postorder:

```
>>  traversal(t, :postorder)
=>  Visited:  M1, M2, VP1, M3, VP2, CEO
```

T16. Let's create a new tree, this time using numbers as values of nodes.

```
>>  t2 = BinaryTree.new
>>  m1 = Node.new(29)
>>  t2.setRoot(m1)
>>  m2 = Node.new(14)
>>  m1.setLeft(m2)
>>  m3 = Node.new(2)
>>  m2.setLeft(m3)
>>  m4 = Node.new(24)
>>  m2.setRight(m4)
>>  m5 = Node.new(27)
>>  m4.setRight(m5)
>>  m6 = Node.new(50)
>>  m1.setRight(m6)
>>  m7 = Node.new(71)
>>  m6.setRight(m7)
>>  view_tree(t2)
```

Do you get a tree that looks like Figure 2.5?

T17. What will be the sequence of preorder traversal on this tree `t2`? Work it out by hand first before calling the Ruby method to confirm your answer.

T18. What will be the sequence of inorder traversal on this tree `t2`? Work it out by hand first before calling the Ruby method to confirm your answer.

T19. What will be the sequence of postorder traversal on this tree `t2`? Work it out by hand first before calling the Ruby method to confirm your answer.

T20. Let us attach a probe to count the number of nodes visited in any traversal. Since all the traversal methods use the `visit` method, we will attach a counting probe there.

```
>>  Source.probe("visit", 2, :count)
=>  true
```

T21. Let us now count how many nodes are visited by each traversal method.

```
>>  count { traversal(t2, :preorder) }
=>  7

>>  count { traversal(t2, :inorder) }
=>  7

>>  count { traversal(t2, :postorder) }
=>  7
```

All the traversal methods visit 7 nodes, which is the total number of nodes in the binary tree `t2`. This is because traversal means iterating through all the nodes, just in different orders.

T22. Create your own binary trees, and test your understanding of the various orders of traversal.

2.4 Binary Search Tree

Suppose that we are searching for a value in a binary tree. We can use any traversal order introduced in the previous section. In the worst case, we have to traverse all the nodes before we find the node we are looking for, or before we realize no such node exists. The complexity of this approach would be $O(n)$, for a binary tree of n nodes.

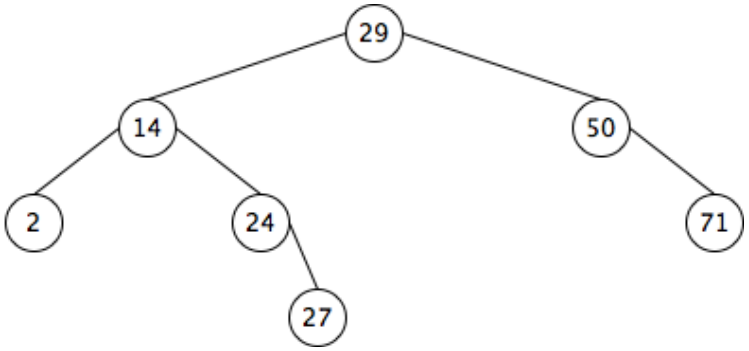


Figure 2.5: *Binary Search Tree*

One lesson that we keep encountering is that search could be much more efficient if the data is organized. For instance, the difference between linear search, with complexity $O(n)$, and binary search with complexity $O(log\ n)$, is that binary search assumes that the underlying array is sorted. Can we apply a similar paradigm of searching an array to searching a binary tree?

The answer is yes, if we keep the nodes in a binary tree in a "sorted" order. What does this mean? See the binary tree in Figure 2.5. Note that for all the nodes in the left subtree of the root node have smaller values than the root node. The nodes in the right subtree have larger values than the root node. Recursively, this applies for any subtree of this binary tree. Searching such a binary tree would be more efficient, because at any branch, we only need to traverse **either** the left subtree **or** the right subtree, but not both.

A **binary search tree** is a special binary tree, where for each node n in the tree, its value is greater than all nodes in its left subtree, and is smaller than all nodes in its right subtree. All subtrees of this tree are themselves also binary search trees.

A node in the tree may be associated with several different attributes. For example, if a node represents a person, we may wish to store the name, as well as other attributes such as profession, date of birth, etc. In this case, we designate one of these attributes to be the *search key*, and this search key is unique.

Searching a Binary Search Tree

In Figure 2.6, we show a recursive algorithm to search a binary tree rooted at `root` for the value `key`. This implementation makes use of preorder traversal. First, we check whether the root node has the value we are searching for. If not found and there is a left subtree, it recursively searches the left subtree first, followed by searching the right subtree. If the key is not found at all, it returns `nil`.

Note that although the order is similar to preorder traversal, it is not necessary to visit all the nodes, which would have been an $O(n)$ operation. `searchbst` finishes as soon as the node with the correct value is found. The method visits at most one node at each level, which means that the complexity of `searchbst` is $O(h)$, where h is the height of the binary search tree. If the binary search tree is full, and n is the number of nodes, then we have $h = log_2\ (n+1)$, which makes the complexity similar to binary search over an array.

```
def searchbst(root, key)
  if root.value == key
    return root
  elsif root.left != nil and root.value > key
    return searchbst(root.left, key)
  elsif root.right != nil and root.value < key
    return searchbst(root.right, key)
  else
    return nil
  end
end
```

Figure 2.6: *A Recursive Method to Search a Binary Search Tree*

```
def insertbst(root, key)
  if root.value == key
    return root
  elsif root.value > key
    if root.left != nil
      return insertbst(root.left, key)
    else
      node = Node.new(key)
      root.setLeft(node)
      return node
    end
  else
    if root.right != nil
      return insertbst(root.right, key)
    else
      node = Node.new(key)
      root.setRight(node)
      return node
    end
  end
end
```

Figure 2.7: *A Recursive Method to Insert a new Value to a Binary Search Tree*

Inserting a New Node to a Binary Search Tree

Because a binary search tree keeps its nodes in a certain order, we have to be careful when inserting a node so that that order is preserved.

Suppose we want to insert new node with value 34 to the binary search tree in Figure 2.5. We first need to find the right parent for this new node. From visual inspection, we can figure out that 34 should be the left child of 50, because it is larger than 29, but smaller than 50.

Algorithmically, how do we find the "correct" position to insert the new node? Note that once the new node has been inserted, calling `searchbst` to search for this node will lead us to this "correct" position. Therefore, we should place the new node in a position where `searchbst` expects to find it in the first place. That would be where `searchbst` would have failed to find the node before it was inserted.

Figure 2.7 shows a recursive algorithm to insert a new node with value `key` to a binary search tree rooted at `root`. Note the similarity in structure to `searchbst`.

- It first inspects whether the current node already has the value `key`. If yes, no new node is inserted, and the current node is returned.
- If the `key` is less than the current node's value, we inspect the left subtree. If there is a left subtree, it recursively tries to insert the new value to the left subtree. If there is no left subtree, a new node with value `key` is inserted as the left child of the current node.
- Otherwise, we inspect the right subtree, either to make another recursive call, or to insert the new node as the right child of the current node.

Challenge

Can you think of how to remove a node from a binary search tree, such that after the removal the nodes in the tree will still be ordered?

Hint: There are three possible scenarios, which need to be handled differentlly. In the first scenario, the node to be removed is a leaf node. In the second scenario, the node to be removed has only one child. In the last scenario, the node to be removed has two children.

Tutorial Project

Start an IRB session and load the module that will be used in this chapter.

```
>>  require 'is103'
=>  true

>>  include TreeLab
=>  Object
```

T23. Let us create a new binary search tree object, and visualize it.

```
>>  bst = BinarySearchTree.new

>>  view_tree(bst)
=>  true
```

This binary search tree is currently empty.

T24. Our first exercise is to re-create the binary search tree in Figure 2.5. To do this, we begin by setting the root node to a new node with value 29.

```
>>  bst.setRoot(Node.new(29))
=>  Value: 29, left child: nil, right child: nil
```

T25. Because a binary search tree is a special form of binary tree, it has all the binary tree operations. In addition, it also allows us to insert and search for a specific key value. To insert a new node, we specify the value that the new node will have.

```
>>  bst.insert(14)
=>  Value: 14, left child: nil, right child: nil
```

The operation `insert` will call the recursive helper method `insertbst` shown in Figure 2.7. It works by first searching for the "correct" position to insert the new value, and then creating a new node in that position. In this case, because 14 is smaller than 29, it is inserted as the left child of the root node.

T26. Insert the other nodes as well. Observe the visualization window. Note how for each insertion, we first search for the right "position" to place the new node. This results in one or more nodes being visited during the search, which are indicated by the animation in the visualization window.

```
>>  bst.insert(2)
=>  Value: 2, left child: nil, right child: nil

>>  bst.insert(24)
=>  Value: 24, left child: nil, right child: nil

>>  bst.insert(27)
=>  Value: 27, left child: nil, right child: nil

>>  bst.insert(50)
=>  Value: 50, left child: nil, right child: nil

>>  bst.insert(71)
=>  Value: 71, left child: nil, right child: nil
```

T27. Do you think the structure of the tree will be the same if we insert the nodes in a different order? Why?

T28. We will now search for value 50 in this tree.

```
>>  bst.search(50)
=>  Value: 50, left child: nil, right child: 71
```

The operation `search` will call the recursive helper method `searchbst` shown in Figure 2.6. It first visits the root node, and because the key (50) is greater than the value of the root node (29), it goes to the right subtree and finds the node with value 50 there.

T29. We will now try searching for a non-existent value.

```
>>  bst.search(51)
=>  nil
```

As expected, it returns `nil`, after navigating to the leaf level and still not finding the key.

T30. Let's attach a counting probe to see how many nodes are visited in order to search for a particular value.

```
>>  Source.probe("searchbst", 2, :count)
=>  true

>>  count { bst.search(50) }
=>  2
```

We only need to visit two nodes. The first one is the root node. The second one is the node with value 50.

T31. Let's search for a value that does not exist.

```
>>  count { bst.search(90) }
=>  3
```

In this case, it goes to root node, the right child, and finally the leaf node of the right subtree, and still does not find anything.

T32. Let's search for a non-existent value that would have been found in the left subtree.

```
>>  count { bst.search(25) }
=>  4
```

Do you know why the answer is 4? What is the maximum number of nodes visited in this binary search tree?

T33. We will now experiment with creating a new binary search tree with random values. Start by setting a root node to this new tree.

```
>>  bst2 = BinarySearchTree.new

>>  view_tree(bst2)

>>  bst2.setRoot(Node.new(rand(100)))
```

The call `rand(100)` generates a random number between 1 and 100. So each time you run this method, it will give you a different number.

T34. Experiment by repeatedly inserting a new node with random values.

```
>>  bst2.insert(rand(100))
```

T35. Try to search for different values in the tree you have just created. Anticipate which nodes will be visited in the search.

2.5 Summary

In this chapter, we learn about a new data structure, tree, used to store hierarchical relationships among data elements. In particular, we focus on binary trees, where each parent node can have at most two children nodes.

There are several ways to traverse a tree to visit all the nodes. Preorder traversal visits the current node *before*, inorder traversal does so *in between*, and postorder traversal does so *after* visiting the left and right children.

We also learn about binary search tree, which is a special type of binary tree with its elements always in order. Searching a binary search tree is more efficient than traversing all the nodes. However, inserting a new node has to be done carefully so as to preserve the ordering among data elements.

Advanced Material (Optional)

For students interested in additional reading, including removal of nodes from a binary search tree, as well as implementations of binary tree and binary search tree in Java, please refer to Chapter 11 of the supplementary textbook [Prichard and Carrano(2011)].