Hierarchical Structures

So far, we have studied sequential structures which assigns each of their elements an index in the structure. This imposes an ordering relationship between elements based on this index. Now we will examine more complex relationships that we can impose between elements in a structure. The first of these is the hierarchy which establishes child-parent relationships between values.

Hierarchical Structures

Hierarchies occur naturally in all sorts of data. For example, consider the following domains:

1. The reporting structure in a large corporation.
2. Classification of animals.
3. HTML documents, i.e., web pages.

All of these domains possess a hierarchical structure. A corporation’s reporting structure may look like this:

```
CEO
   
Vice President (Finance)  Vice President (Engineering)
   
Manager  Manager
   
Analyst  Analyst  Accountant  Accountant
```

The various individual contributors to a company (e.g., analysts, accountants, and programmers) report to their managers. The various managers within a division report to the vice president of the division. Finally, the various vice presidents report to the CEO who sits at the top of the reporting structure.

Living organisms are classified as follows:

```
Living Things
   Bacteria  Protozoa  Chromista  Plantae  Fungi  Animalia
   
   ...  ...  ...  ...  ...  Chordata  ...
   
   ...  ...
```

Biologists divide up living things into a hierarchy of classes: kingdoms, phylums within those kingdoms, classes within those phylums, and so forth.

Finally, HTML documents have a hierarchical structure as well. If you are reading this webpage in Microsoft Edge or Chrome, you can view its source in a tree-like format (e.g., in
Chrome: Options \(\rightarrow\) More Tools \(\rightarrow\) Development Tools). Every HTML document begins with an outer `<html> ... </html>` tag. Inside this tag are the elements of the webpage. For example `<head> ... </head>` contains header information about the page and `<body> ... </body>` contains the actual page content. Here is a barebones HTML document with the hierarchy of tags made explicit:

```
<html>
  <head>
    <title>My Title</title>
  </head>
  <body>
    <h1>Section Header</h1>
    <p>Paragraph</p>
  </body>
</html>
```

The structure imposed by the hierarchies in all three examples is essential. For example, imagine if we wrote down the corporation reporting structure down as a list:

CEO, Vice President (Finance), Manager, Analyst, Analyst, Manager, Account, Account, Vice President (Engineering).

From the list, it isn’t clear who reports to who in the company! Our list representation of the company has lost the reporting structure that the hierarchy captured.

**The Tree Abstract Data Type**

From our discussion above, it is clear that a list is insufficient for representing these sorts of data that have hierarchical relationships. We instead use a different abstract data type, the tree, to capture these relationships.

A tree is a data type that encodes a hierarchical structure among its elements. The meaning of these relationships is domain-specific. As a result, trees typically don’t have a fixed interface like a list—the operations we’d like to perform depend heavily on what the tree is used for. Nevertheless, we’ll study some essential core operations and programming techniques over trees that you can adapt to a variety of situations.

To begin, we’ll study the simplest form of a tree to understand these basics. Trees have an elegant recursive definition: a tree is either:

- An **empty leaf** or
- A **node** consisting of an element, a left sub-tree, and a right subtree.

We can visualize the second case below:
We typically denote the left and right sub-trees, \textit{left} and \textit{right}, respectively. Note that these sub-trees are simply recursive occurrences of our tree definition—they'll either be empty or be a tree itself.

As a concrete example, consider the following tree of integers:

```
  5
 /|
/  |
3  9
/  \
1  2 7 11
```

The leaves of the tree are denoted by single dots (·). The top-most element of the tree is called its \textit{root}—here the element 5 is the root. The root has two sub-trees. The left sub-tree contains the elements 1, 2, and 3. The right sub-tree contains the elements, 6, 7, 9, and 11. We can identify any of the sub-trees by its root, \textit{e.g.}, the sub-tree \textit{rooted at} 3 contains itself, 1, and 2. The sub-tree rooted at 7 contains itself and 6. As a degenerate case, the sub-tree rooted at 11 only contains itself, but it is still a tree, nevertheless.

For any two elements in the tree we can talk about the relationship that the tree induces between them. For example, 7 appears as the root of the left sub-tree rooted at 9. Therefore, we say that 7 is a \textit{child} of 9; conversely, 9 is a \textit{parent} of 7. We’ll use all sorts of similar terminology to denote these parent-child relations as is appropriate for the domain, \textit{e.g.}, subordinate and boss for the corporate domain or subclass and superclass for the living organism domain.

Drawing out the empty leaves is usually unnecessary. Therefore, we frequently leave them out to simplify the diagram:

```
  5
 /|
/  |
3  9
/  \
1  2 7 11
```

With this diagram, the elements 1, 2, 6, and 7 are the \textit{leaves} of the tree since they have no children. The remaining elements form the \textit{interior nodes} of the tree.

\textbf{Placement of Data}

Our initial definition of a tree places the data at the \textit{interior nodes} of the tree—\textit{i.e.}, the non-leaf nodes of the tree. However, there is nothing essential about this choice. Indeed, we may give an
alternative definition of a tree that places the data at the leaves rather than the nodes: A tree is either:

- A leaf containing a value or
- A node containing a left and right sub-tree.

With this definition, our sample tree above may be represented as follows:

```
  1
 / \
3   6
 / \ / \ / \
2  7 9 11
```

While this tree is structurally distinct from our original tree, they both encode the same information. Choosing one representation over another is simply a matter of choosing the representation that best fits the domain the tree is being used in.

**Tree Representation and Operations**

In essence, a tree is similar to a linked list except that instead of a single `next` field that contains the “rest of the list”, it has two fields `left` and `right` that contain the “rest of the tree”. Therefore, we adopt a similar strategy to represent a tree in Java—a class to represent nodes of a tree and a class to represent the tree itself:

```
// In Node.java
public class Node<T> {
    public T value;
    public Node<T> left;
    public Node<T> right;
    public Node(T value, Node<T> left, Node<T> right) {
        this.value = value;
        this.left = left;
        this.right = right;
    }
    public Node(T value) { this(value, null, null); }
}

// In Tree.java
public class Tree<T> {
    private Node<T> root;
    public Tree() { root = null; }
    // Tree operations...
}
```
Here, the leaves of the tree are represented with a `null Node<T>` value rather than some other class.

Because our tree contains data, we ought to perform similar sorts of operations that we can perform over lists, e.g., adding elements, querying for an element, or checking the size of the container. Imagine implementing this last operation, `size()` for a linked list. We would maintain a `cur` reference to the current node in the list we are examining and increment a counter for each one. Let’s try the same approach for our tree:

```java
// In the Tree class...
public int size() {
    Node<T> cur = root;
    int size = 0;
    while (cur != null) {
        size += 1;
        cur = cur.left;
        // But what about cur.right...?
    }
    return size;
}
```

However, we run into a problem if we try to apply our linked list traversal techniques to trees. This attempt at `size()` traverses the left-hand nodes of the tree but doesn’t visit the right-hand nodes. But once we leave a node, we have no way of coming back to it to visit those nodes.

To remember these nodes, we need to appeal to an auxiliary data structure, e.g., a list.

```java
// In the Tree class...
public int size() {
    List<T> pending = new LinkedList<T>();
    pending.add(root);
    int size = 0;
    // Loop invariant:
    // pending contains the current frontier of nodes
    // that we still need to visit.
    while (pending.size() != null) {
        Node<T> cur = pending.remove(0);
        if (cur.left != null) { pending.add(cur.left); }
        if (cur.right != null) { pending.add(cur.right); }
        size += 1;
    }
    return size;
}
```

Here, we use a list essentially like a `queue`, adding nodes to be explored to the end of the list and then removing nodes to visit from the front.
This approach works, however, the use of an auxiliary data structure is somewhat undesirable. Furthermore, our solution does not reflect the recursive definition of the tree. Because of this, we’ll pursue a recursive definition of the size() operation for trees, mirroring this definition.

In the absence of a particular programming language, we can define the size operation in pseudocode to mirror the definition of a tree:

- The size of a leaf is 0.
- The size of a node is one plus the size of its left and right subtrees.

There are several ways to reflect this definition in Java with varying trade-offs of complexity, elegance, and handling of corner cases. Here, we present a particular style that allows our code to reflect this definition directly:

```java
// In the Tree class...
private static int sizeH(Node<T> cur) {
    if (cur == null) {
        return 0;
    } else {
        return 1 + sizeH(cur.left) + sizeH(cur.right);
    }
}

public int size() { return sizeH(root); }
```

We establish a static helper function, sizeH, that computes the size of a tree rooted at a given Node<T> object. The method proceeds by case analysis on the shape of that Node<T>—it is null or not. In this manner, the helper function mirrors rather closely the pseudocode definition given above. Finally, we define the actual size method to simply call this helper method starting with the root of the tree.

**Binary Search Trees**

Before we discuss other tree operations, it is worthwhile to narrow our domain of interest to take advantage of the hierarchical relationships that the tree offers. Recall that linear search over an unsorted sequential structure has $O(n)$ time complexity. However, if the structure is already sorted then we can employ binary search which has $O(\log n)$ time complexity instead. The catch is that we must now keep the structure sorted which requires some additional work on top of the sequential operations we’ve discussed previously.

A binary search tree is a tree-based structure that maintains a sortedness property among its elements. It does this by way of an invariant that is baked into the definition of a binary search tree. A binary search tree is a tree consisting of either:

- An empty leaf.
- A node consisting of a value and left- and right-subtrees with the property that all the elements in the left subtree is less than the value and all the elements in the right subtree are greater than or equal to the value.
We can visualize this binary search tree invariant as follows:

\[
\begin{align*}
 & & v \\
 & & \{< v\} \quad \{\geq v\}
\end{align*}
\]

This invariant gives us guidance as to where to place elements in the tree. For example, consider starting out with an empty binary tree and then adding the elements 3, 5, 2, 6, and 4. Here is the evolution of our tree after each insertion.

\[
\begin{array}{c}
3 \\
\downarrow \\
5 \\
\downarrow \\
2 \\
\downarrow \\
6 \\
\downarrow \\
\end{array}
\begin{array}{c}
3 \\
\downarrow \\
2 \quad 5 \\
\downarrow \\
2 \quad 5 \\
\downarrow \\
\end{array}
\begin{array}{c}
3 \\
\downarrow \\
2 \quad 5 \quad 6 \\
\downarrow \\
2 \quad 5 \\
\downarrow \\
\end{array}
\]

In general, our insertion strategy is to traverse the tree according to the binary search tree invariant to find a leaf. We then replace the leaf with a node containing the value to be inserted. In the above example:

1. Initially, we replace the single leaf of the empty tree with a node containing the value 3.
2. To insert 5, we note that 5 is greater than 3, so we recursively dive into the right-hand subtree, find that it is a leaf, and replace it with a node containing 5.
3. To insert 2, we note that 2 is less than 5, so we recursively dive into the left-hand subtree, find that it is a leaf, and replace it with a node containing 2.
4. To insert 6, we note that 6 is greater than 3 and 5, so it goes into the right-most subtree.
5. Finally, to insert 4, we note than 4 is greater than 3 but less than 5, so goes into the left subtree of 5.

We can generalize these examples into a procedure for inserting elements into a binary search tree. When inserting a value \(v\) into a binary search tree:

- If you are inserting into a leaf, then replace that leaf with a node containing \(v\) and no left or right subtrees.
- If you are inserting into a node that contains some value \(v'\), then recursively insert into the left subtree if \(v < v'\), otherwise recursively insert into the right subtree.

We may realize this in Java as follows:

```java
public class BinarySearchTree<T extends Comparable<T>> {  
    // Node class same as before...
    private Node<T> root;
    public BinarySearchTree() { root = null; }
}
```
/** @return the updated tree after inserting h into the given tree */
private static Node<T> insertH(T v, Node<T> cur) {
    if (cur == null) {
        return new Node<>(v);
    } else {
        if (v.compareTo(cur.value) < 0) {
            cur.left = insertH(v, cur.left);
        } else {
            cur.right = insertH(v, cur.right);
        }
        return cur;
    }
}
public void insert(T v) { root = insertH(v, root); }

The definition of the BinarySearchTree<T> class is identical to our regular Tree class. The exception is that in order to maintain the binary search tree invariant, we must be able to compare elements contained within the Tree. This means that we must constraint the generic type T to be any type that implements the Comparable<T> interface, i.e., T defines how to compare elements against itself.

The definition of insert follows the skeleton we established for size above. However, unlike size, insert modifies the underlying tree once it finds a leaf—a node that is null. To avoid having to write lots of null checks for each of the subtrees of a node, we employ a recursive design pattern called the update pattern. Our recursive method, insertH takes the Node<T> that is the root of the tree as well as the element to insert as input. The method also returns a value—the updated root—as output. In the case where we insert into a leaf, the root is null, so the method returns a new node. In the case where we insert into a node, the root is non-null, so the method simply returns the node that was passed to it. However, along the way, insertH modifies this node with an updated subtree.

We can think of insertH as returning an updated version of its input Node<T>. This is why the public version of insert has the following form:

root = insertH(v, root);

We have updated the root of the tree with the result of inserting v into the tree.

Tree Traversals

Next let’s revisit traversal of a tree. size() is a simple example of a tree traversal method. However, the order we visit the elements of the tree is irrelevant in calculating the size of the tree. In contrast, imagine a method toString that prints the elements of the tree. Here, the order in which we visit the elements does matter.

Consider the following sample tree:
And the following pseudocode description of `toString`:

- If the tree is a leaf, print nothing.
- If the tree is a node, print the value at the node, recursively print the left-hand tree, and recursively print the right-hand tree.

This version of `toString` first prints the value at a node before recursively descending. This results in the following output for the sample tree:

\[ [5, 2, 1, 3, 8, 7, 6, 9, 10] \]

This is an example of a **pre-order traversal** of the tree where we "visit" the value at the node first, then the left-hand subtree, and the right-hand subtree.

We can exchange this order to obtain two other traversal strategies:

- **In-order traversal**: Recursively process the left-hand subtree, "visit" the value at the node, recursively process the right-hand subtree.
- **Post-order traversal**: Recursively process the left-hand subtree, recursively process the right-hand subtree, and "visit" the value at the node.

An in-order traversal of the sample tree yields the list \([1, 2, 3, 5, 6, 7, 8, 9, 10]\). The post-order traversal of the sample tree yields the list \([1, 3, 2, 6, 7, 10, 9, 8, 5]\).

Each traversal order has its use cases. In particular, an in-order traversal of a binary search tree such as the sample tree yields the elements of the tree in sorted order. Pre-order traversal provides a convenient way for **serializing** a tree into a linear form appropriate for a file that can be used to recreate the tree later. If we interpret the interior nodes of the tree as operators and the leaves as values, post-order traversal yields **postfix notation** or **reverse polish notation (RPN)** which does not require expressions to be parenthesized. For example, the mathematical expression written in traditional infix style \(3 \times (4 + 5)\) has the unambiguous representation in RPN: \(3 4 5 + \times\).

**Complexity Analysis**

Finally, let’s consider the time complexity of the various tree operations we’ve discussed in this chapter. The various traversals, like their sequential counterparts, visit every element of the structure; they therefore all take \(O(N)\) time where \(N\) is the number of elements in the tree.

More interesting is the cost of lookup and insertion into a binary search tree. In the worst case of lookup, we search one path from the root of the tree to one of its leaves. For example, in the following binary search tree:
If we look for the value 4, we’ll visit the nodes 3, 5, and 4 during the search process. Thus, the runtime of lookup is dependent on the length of such a path.

Let’s consider a degenerate example of a binary search tree.

This tree is a binary search tree, however, it is far from an ideal one. It is essentially a linked list! Searching this binary search tree takes $O(N)$ time in the worst case, the same as linked list search.

Now let’s consider an ideal binary search tree:

This binary search tree has three levels of nodes. The first level contains the element 5. The second level contains the elements 3 and 7. The third level contains the elements 1, 2, 6, and 8. Each of these levels are full, that is, they have the maximum number of possible elements. We call such a tree perfect—all interior nodes have two children and all leaves exist at the same level.

To assess the length of a path in this perfect tree from root to leaf, we must consider the number of nodes at each level of a perfect binary tree. The first level contains 1 element, the second contains 2, the third contains 4, the fourth level contains 8, the fifth level contains 16, and so forth. It is reasonable to hypothesize that the number of nodes at level $i$ is $2^i$. This turns out to be true and provable with a quick proof by mathematical induction:

**Claim 1.** The number of nodes at level $i$ of a perfect binary search tree is $2^i$. 
Proof. Proof by induction on the level $i$.

- $i = 0$: at level 0 (the first level), there is 1 node, the root, and $2^0 = 1$.

- $i = k + 1$: by our inductive hypothesis, level $k$ contains $2^k$ nodes. Because each node of level $k$ contributes two nodes, a left and right child, to level $k + 1$, then the number of nodes at level $k + 1$ is $(2^k) \cdot 2 = 2^{k+1}$.

The total number of nodes in a perfect binary search tree of height $h$ is therefore given by summing up the nodes at each level:

$$N = \sum_{i=0}^{h} 2^i = 2^0 + 2^1 + 2^2 \cdots + 2^h$$

This sum relates the total number of nodes of the tree with its height. It has a closed form solution:

$$N = \sum_{i=0}^{h} 2^i = 1 - \frac{2^{h+1} - 1}{2 - 1} = -(1 - 2^h) = 2^{h+1} - 1.$$

A perfect binary tree of height $h$ has $2^{h+1} - 1$ nodes. From this, we can solve for $h$ in terms of $N$.

$$N = 2^{h+1} - 1$$

$$N + 1 = 2^{h+1}$$

$$\log_2 N + 1 = \log_2 2^{h+1}$$

$$\log_2 N + 1 = h + 1$$

$$\log_2 N + 1 - 1 = h$$

Thus, the height is bounded by $\log N$. When the tree is perfect, lookup has worst-case time complexity $O(\log N)$. Note that insertion into a binary search tree operates identically, so it too has worst-case $O(\log N)$ time complexity in this situation.

However, what is the appropriate average case time complexity? This turns out to be difficult to analyze precisely—what is the layout of the average tree? This depends on the effects of the insertion and deletion operations performed on the tree. In particular, deletion favors the rotation of one side of the nodes, so we might expect that the tree becomes more unbalanced with repeated deletions.

To make progress, we can restrict our question to layout of the average tree created by only a chain of insertion. It turns out that, on average, the height of such a tree is $O(\log N)$, i.e., the average height is within some constant factor of the optimal height. Therefore, in this situation, the average case of insertion is $O(\log N)$. However, even when we consider only insertions, we can still obtain the degenerate binary search tree if we insert elements in-order.

In general, our current insertion policy does not allow us to maintain a balanced tree shape, one that looks roughly like a perfect tree. To get around this problem, we employ various balancing techniques to maintain a balanced tree while maintaining good performance of our fundamental tree operations. Examples of trees employing balancing techniques include AVL trees, red-black
trees, and B-trees. All of these structures place additional constraints or invariants on the structure of the tree that ensure that it remains well-balanced.