Trees in different programming languages

In this reading, you will learn about some of the differences in the implementation of trees in different programming languages.

A tree is an abstract data type (ADT) that is present in many languages. As discussed in other readings, an ADT is a blueprint for how a data structure will manifest. It relates to the restraints and requirements placed on a data structure to ensure that it will always operate in the same way. This can be very useful for programmers who are switching between multiple programming languages as it provides clear expectations of how a data structure will operate.

The key fundamental principles of a tree are that it contains a number of nodes, a root node and a selection of leaf nodes. Leaf nodes are unconnected nodes at the base of a tree. A root node is always at the top, and every value descends from this one node. A tree will always be arranged in some form of hierarchy.

Implementing a tree

There are many types of trees; perhaps the most straightforward and common one is a binary tree. A binary tree has the following properties:

1. Every node has a maximum of two child nodes.
2. Every node must have a key so that it can be easily identified.
3. Values found to be less than the node are placed in the left child node, and values that are greater are placed in the right child node.

To create a tree, one must ensure that there is a root (starting node) and a method for determining the subsequent nodes. Each node must contain a reference to the left and right nodes so that the tree can be traversed. This can be achieved by creating a class with these three attributes (key, location of left node, location of right node). This class will need three additional methods so that it can function as a tree. These include:

1. A lookup method: It is important that the tree can be queried for the existence or absence of information.
2. Insertion method: As has already been noted, inserting a node on a tree involves finding out where it should go and placing it on the left side of the nearest higher value.
3. Removal method: This method will need to remove an item from a tree. This operation poses additional challenges when applied to a tree due to the connected nature of a tree. Consider that a tree comprises a series of connected nodes. So, removing one carelessly can result in destroying the connections in a tree. Therefore, when implementing this method, in addition to removing the node, it is necessary to check all the children nodes and ensure that a new connection is made with the node of the next highest value.

Searching in trees

The approach outlined above for coding a tree is applicable to all the languages covered in this course. There are no concrete implementations of trees, which means that to use them, you are required to implement the code yourself. Another feature that is worth bearing in mind is how a search is performed on a tree. One can code a solution that applies a depth-first search or a breadth-first search. To better understand this concept, consider how the tree is organized. Each level has a parent node that connects to two child nodes. These child nodes, in turn, have two of their own child nodes. A breadth-first search is one that will examine each node on the same level before stepping into a deeper level. This can be pictured as scanning all nodes horizontally before checking the next level. A depth-first search will examine each node on a branch until the end node is reached before checking the adjacent branch. This can be pictured as a vertical scanning of the tree.

Conclusion

This reading covered the topic of trees and how they appear in different programming languages. Because trees are an abstract data type, there is no built-in method for the languages used in this course. To implement a tree, it is crucial to keep in mind the important characteristics of a tree so that you can correctly implement them.