Abtstract Data Types and Trees

Abhinav Ashar
CS 61B: Data Structures and Algorithms

March 25, 2019

1 Abstract Data Types

What is an Abstract Data Type?

Abstract data types are a concept in Java that allows for the flexible use of objects. Different pieces of data work in different ways, and it is impossible to anticipate all the different possible functionalities. Thus, programmers use abstract data types to define the general structure of a class, but not specify how exactly it should work. In essence, it ties together data and functionality.

Why do we use an Abstract Data Type?

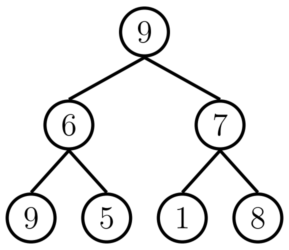
Abstract Data Types create a layer of abstraction. This is because we don't always know the type of data we are working with. Is it an List of Strings or ints? It is a Set of Doubles or chars? Because we don't know the possible specific implementations and we don't want be redundant by providing all the different implementations, we provide a generic type and don't specify how it should work.

How to use an Abstract Data Type?

Use the <> symbol with parameters that will be the different, unspecified types

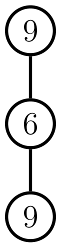
2 Trees

Binary Trees
As the name may suggest, binary trees are trees where each node has up to 2 children.



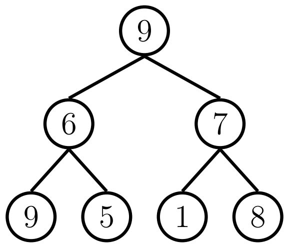
One of the ways we can represent a BinaryTree class is to have a Node subclass since every binary tree is made up of nodes. Each Node would have a value (generic type) and two Node pointers, $left$ and *right*. This allows us to go from one node to its children, down a path of the tree. If a node does not have one or both children, then either $left$, *right*, or both are equal to *null*.

Balanced Binary Trees
When working with a binary tree, there are many ways in which the data can be distributed. In one case, the tree can end up looking like a single linear path.



This is called a *spindly* tree. When it comes to the different types of trees, this tends to be the most inefficient version because it has a height of n. This is important because many operations (add, contains etc.) in the worst case require us to go down to the leaf, which will force us to traverse n nodes in this case.

On the other hand, there could be a tree where each node has both a $left$ and *right*, except for leaves.



This is a *balanced* tree, one of the best versions of a tree that has a height of approximately $\log n$. The proof for why this is true is not necessary to know, but if you really want to know it, here is a good explanation. Since it has a height of about $\log n$, going down to the leaf for some operations will only require us to traverse $\log n$ nodes, which is much more efficient than n.