SIENAcollege
Computer Science

Computer Science 210
Data Structures
Siena College
Fall 2020

Topic Notes: Trees

We've spent a lot of time looking at a variety of structures where there is a natural linear ordering of the elements in arrays, vectors, linked lists. We then put some restrictions on those structures, looking at stacks, queues, deques, and ordered structures.

Just like we can write programs that can branch into a number of directions, we can design structures that have branches.

Today, we'll start looking at our first more complicated structure: *trees*.

In the structures we have studied so far, elements are very specifically ordered. In our list structures, we can refer to elements by an index. One way to think of this is that every element has *unique successor*.

In a trees, an element may have *multiple successors*.

We usually draw trees vertically, but upside-down, in computer science.

You won't see trees in nature that grow with their roots at the top (but you can see some at Mass MoCA over in North Adams).

Examples of Trees

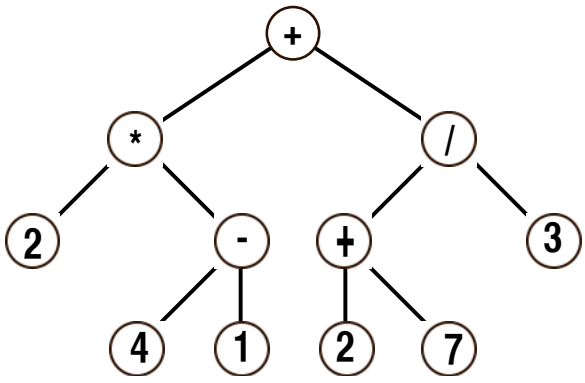
Expression trees

One example of a tree is an *expression tree*:

The expression

```
(2*(4-1))+((2+7)/3)
```

can be represented as

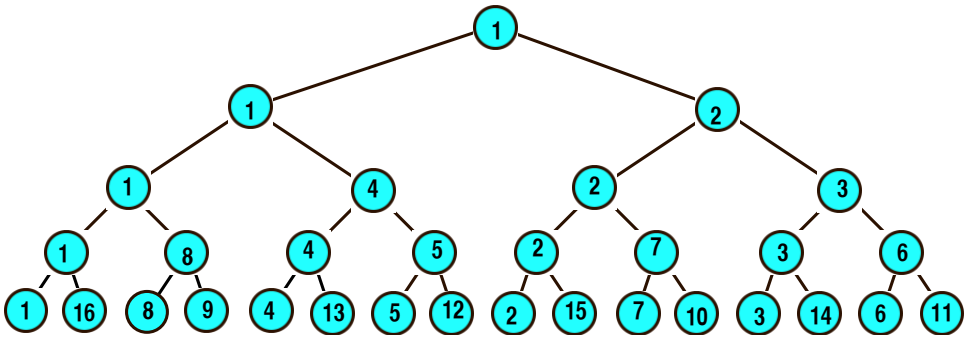


Once we have an expression tree, how can we evaluate it?

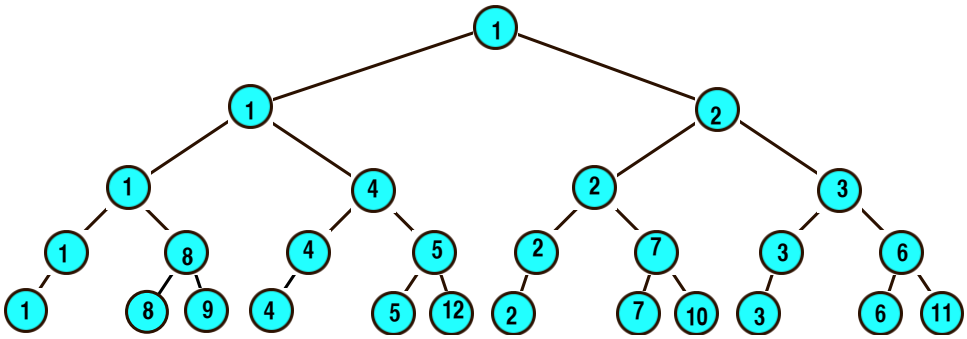
We evaluate left subtree, then evaluate right subtree, then perform the operation at root. The evaluation of subtrees is recursive.

Tournament Brackets

Another example is a tree representing a tournament bracket:



or



Tree of Descendants

One popular use of a tree is a pedigree chart – looking at a person's ancestors. Instead, let's look at a person's descendants. (Example drawn in class).

Tree Definitions and Terminology

There are a lot of terms we will likely encounter when dealing with tree structures:

A tree is either *empty* or consists of a *node*, called the *root node*, together with a collection of (disjoint) trees, called its *subtrees*.

- An *edge* connects a node to its subtrees
- The roots of the subtrees of a node are said to be the *children* of the node.

- There may be many nodes without any children: These are called *leaves* or *leaf nodes*. The others are called *interior nodes*.
- All nodes except root have unique *parent*.
- A collection of trees is called a *forest*.

Other terms are borrowed from the family tree analogy:

- *sibling* – nodes sharing the same parent
- *ancestor* – a node's parent, parent's parent, *etc.*
- *descendant* – a node's child, child's child, *etc.*

Some other terms we'll use:

- A *simple path* is series of distinct nodes such that there is an edge between each pair of successive nodes.
- The *path length* is the number of edges traversed in a path (equal to the number of nodes on the path - 1)
- The *height of a node* is length of the longest path from that node to a leaf.
- The *height of the tree* is the height of its root node.
- The *depth of a node* is the length of the path from the root to that node.
- The *degree of a node* is number of its direct descendents.
- The idea of the *level of a node* is defined recursively:
  - The root is at level 0.
  - The level of any other node is one greater than the level of its parent.

  Equivalently, the level of a node is the length of a path from the root to that node.

We often encounter *binary trees* – trees whose nodes are all have degree ≤ 2.

We will also orient the trees: each subtree of a node is defined as being either the *left* or *right*.

For binary trees, there are additional properties to consider, though not all terms are universally accepted to have the meanings below.

- a *full* binary tree is one in which every node has either 0 or 2 children (*i.e.*, no only children allowed)

- a *complete* binary tree is one where each level contains all possible nodes exept the last, and any missing nodes must be all the way to the right
- a *perfect* binary tree is one where all levels in existence contain all possible nodes (some texts use this as the definition of a full tree)

A Binary Tree for Arithmetic Expressions

Much like we did for examples of other structures, we will consider a specialized binary tree as our first actual tree stucture. This one will store and evaluation binary expression trees.

`https://github.com/SienaCSISDataStructuresJDT/binaryexprtree-example`

Tree Traversals

Iterating over all values in our linear structures is usually fairly easy. Moreover, one or two orderings of the elements are the obvious choices for our iterations. Some structures, like an array or an `ArrayList`, allow us to traverse from the start to the end or from the end back to the start very easily. A singly-linked list, however, is most efficiently traversed only from the start to the end.

For trees, there is no single obvious ordering. Do we visit the root first, then go down through the subtrees to the leaves? Do we visit one or both subtrees before visiting the root?

Think about the orderings we used for the expression tree: one was used to evaluate (where we needed to do something with both children before we could apply an operator) and a different one was used to print it (where we needed to print the operator between the left and right child printouts).

We will consider 4 standard *tree traversals* for our binary trees:

1. *preorder*: visit the root, then visit the left subtree, then visit the right subtree.
2. *in-order* visit the left subtree, then visit the root, then visit the right subtree.
3. *postorder*: visit the left subtree, then visit the right subtree, then visit the root.
4. *level-order*: visit the node at level 0 (the root), then visit all nodes at level 1, then all nodes at level 2, etc.

For example, consider the preorder, in-order, and postorder traversals of the expression tree we looked at in the example code:

- preorder leads to prefix notation:
  / * + 4 3 - 10 5 2

- in-order leads to infix notation:
  4 + 3 * 10 - 5 / 2
- postorder leads to postfix notation:
  4 3 + 10 5 - * 2 /

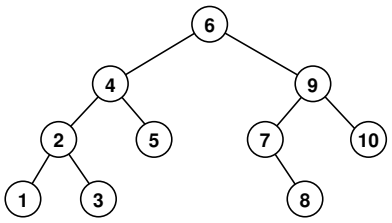
The iterator concept fits nicely with tree traversals, but the code for tree iterators tends to be somewhat complex, so we will first consider traversals without iterators.

The `BinaryExprTree` class has methods to perform these traversals. Three of the traversals are naturally recursive, so we implement them recursively. The level order traversal is not recursive, but demonstrates a good use of a queue data structure!

Binary Search Trees

Possibly the most common usage of a binary tree is to store data for quick retrieval.

Definition: A binary tree is a *binary search tree* (BST) iff it is empty or if the value of every node is both greater than or equal to every value in its left subtree and less than or equal to every value in its right subtree.

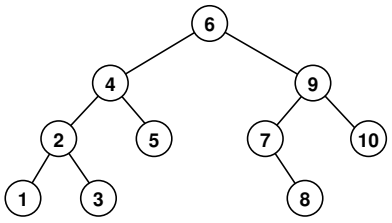


Operations on the BST structure (add, remove, contains) can take advantage of the fact that the values stored in the nodes have this ordering property to be much more efficient than similar operations on other structures.

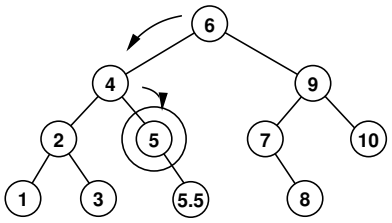
For example, let think about an `add` method. We will be creating a new node with the given value, and setting parent and child links in the tree to place this new value appropriately.

We need to consider several cases:

1. We are adding to an empty binary tree, in which case we just make the new node the root of the BST.
2. We add the new value as a new leaf at the appropriate place in the tree as the child of a current leaf. For example, add 5.5 to this tree:

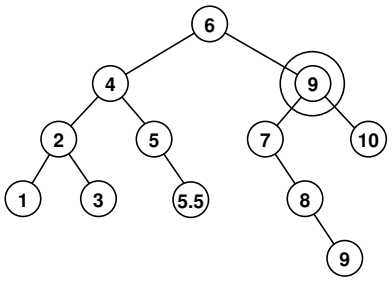


to get



3. If the value is one that is already in the tree, the search may stop early, in which case we need to add the new item as a child of the predecessor of the node found.

   For example, adding 9 to the tree above:



   We can choose to add duplicate values as either left or right children as long as we're consistent. Our implementation chooses to put them at the left.

In `BinarySearchTree.java` in `https://github.com/SienaCSISDataStructuresJDT/bstintro-examples` has the basics of the code you will be working with in lab next week.

For now, let's work this example, tracing the code for at least the first few values added:

Create a BST by inserting the numbers 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31 from left to right. What is the height of the tree?

`Comparable` `Object`**s**

What if we want to make our BST generic? We have seen how to do this with other structures by using a type parameter. We could introduce a type parameter `E`, and use it in place of the primitive type `double` in instance variables, parameters, and return values.

But...if we are going to deal with generic `Objects` for a binary search tree, we need a way to compare them. In the `recursiveAdd` method (and in other methods we'd want in our BSTs), you will see there are comparisons of `double` values using the standard operators like <= and >. Those can't be used to compare our generic objects in any meaningful way.

What we want is to write a method that compares an `Object` to another, like the `compareTo()` method of `String`s. However, there is no `compareTo` method in `Object`, and it's the methods provided by the `Object` class that are the only ones we can assume exist for any possible type we'd want to store in our BST.

Fortunately, Java provides an interface that does exactly this, the `Comparable` interface. Any object that implements `Comparable` will have a `compareTo` method, so if we write our BST to operate on objects that implement `Comparable`, we know they'll have a `compareTo` method that we can use any time we need to compare two of our objects in the methods of the BST.

So that's good, but we need to tell Java that objects allowed in our BST can be of any type that implements the `Comparable` interface. The syntax for this is a bit messy but not too bad once you get used to it. Here's a version of the code above that uses `Comparables` instead of `doubles`.

This is in `BinarySearchTreeT.java` in `https://github.com/SienaCSISDataStructuresJDT/bstintro-examples`.

This allows us to use any class that implements `Comparable` to be used as the values of this tree implementation. Many Java API classes, including `String`, `Integer`, and `Double`, implement `Comparable`. And we can make our own custom classes implement `Comparable`, providing an appropriate `compareTo` method, and use those as the values of this tree implementation as well.