Binary Trees - the Beginning

Robin Dawes

February 17, 2021

Begins our deep and (hopefully) rewarding study of the properties and applications of binary trees - a topic of essential importance in computing.

.

Planting the Seeds

In this introduction to binary trees I will assume you are familiar with trees as defined in CISC-203.

Definition: a rooted tree is a tree in which one vertex is identified as the root. The edges may or may not be directed. If so, either all edges are oriented with their heads pointing towards the root, or all edges are oriented with their heads pointing away from the root.

When the edges are oriented away from the root, the set of vertices $\{y \mid \exists \text{ edge } \overrightarrow{(x,y)}\}$ are called the children of x. We call x the parent of its children. Vertices with the same parent are called siblings.

By convention, an empty tree is considered to be a rooted tree.

This may seem silly but it actually simplifies some of our later definitions.

Definition: A Binary Tree is a rooted tree in which each vertex has at most two children. The children of a vertex (if any) are labelled as the left child and the right child. If a vertex has only one child, that child can be either the left child or the right child.

Binary trees can also be defined recursively:

Definition: A rooted tree T is a binary tree if:

T is an empty tree, or

T consists of a root vertex with a left subtree and a right subtree, each of which is a binary tree

This recursive definition prefigures the pattern of most algorithms we use on this data structure, as we will see as we go forward.

There are at least two options for implementing binary trees. For now we will focus on the obvious method: objects with pointers.

We can use two classes to define binary trees: Binary_Tree_Vertex and Binary_Tree.

It's actually possible to everything with just one class ... but I've always done it with two! YMMV

A Binary_Tree_Vertex object needs the following attributes:

- **value** ... which could be a single value, a collection or list of information, or a key value and associated data, etc.
- **left_child** ... in a typed language, this is a pointer to a Binary_Tree_Vertex object
- **right_child** ... same

and may also have pointers to the vertex's siblings, its parent, the root of the tree, etc.

A Binary_Tree object needs just one attribute:

- **root** ... in a typed language, this is a pointer to a Binary_Tree_Vertex object

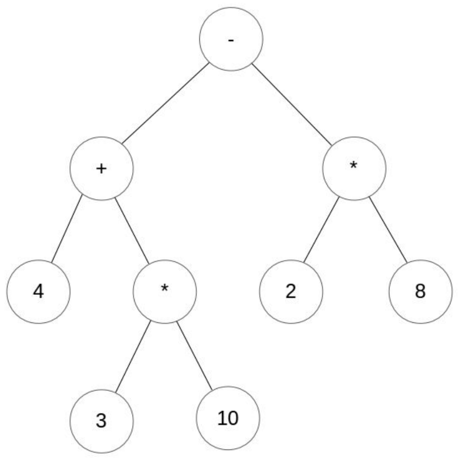
and may also have attributes such as "height" and "number_of_vertices"

We will adopt the common "<object>.<attribute>" notation ... so if T is a Binary_Tree object, we will refer to T's root as T.root

Similarly if v is a Binary_Tree_Vertex object, we will refer to v.value, v.left_child and v.right_child.

Binary Tree Traversals

One of the things we do frequently with binary trees is traverse them, which means "visit each vertex of the tree". There are four popular methods for traversing binary trees. We will illustrate them on this tree, which has a token stored in each vertex.



In-Order Traversal

The basic idea of In-Order Traversal is to explore the left subtree, then look at the current vertex, then explore the right subtree. We can write this recursively:

```
In_Order(v):        # v is a vertex in a binary tree
    if v == nil:
        return
    else:
        In_Order(v.left_child)
        print v.value
        In_Order(v.right_child)
```

If we apply this to the tree shown above, the result is

$$4 \; + \; 3 \; * \; 10 \; - \; 2 \; * \; 8$$

Well that's interesting - this creates an arithmetic expression in standard infix form!

Pre-Order Traversal

The basic idea here is to look at the current vertex, then explore its left subtree, then explore its right subtree. In pseudo-code, the recursive form of this is:

```
Pre_Order(v):          # v is a vertex in a binary tree
    if v == nil:
        return
    else:
        print v.value
        Pre_Order(v.left_child)
        Pre_Order(v.right_child)
```

If we apply this to the tree shown above, the result is

$$- \; + \; 4 \; * \; 3 \; 10 \; * \; 2 \; 8$$

In the notes about Stacks we looked at POSTFIX NOTATION but we didn't spend any time talking about PREFIX NOTATION for arithmetic expressions ... but it's not complicated. In postfix notation each operator follows its operands ... but in prefix notation each operator *precedes* its operands. The expression shown above is correct prefix notation for the expression we are working on. It would be interpreted (by a talking computer) as ... "Oh a minus sign. I need two numbers. Now I have a plus sign - I need two numbers for that. There's a 4 - that's one number for the addition. Now I have a multiplication sign - I need two numbers for that. There's a 3. There's a 10. I have the two numbers for the multiplication: 3*10 = 30. Now I have the second number for the addition: 4 + 30 = 34. I still need a second number for the subtraction. I see a multiplication - I need two numbers. There's a 2. There's an 8. Now I can compute 2*8 = 16. 16 is the second number I need for the subtraction so I can compute 34 - 16 = 18. Now I need a cool refreshing beverage."

We could also process prefix expressions in a right-to-left order - this would be completely analogous to processing a postfix expression from left-to-right: we would encounter the operands and then the operator.

We talked previously about how postfix notation is deeply related to the way expressions are actually evaluated at the assembly language level in a computer (first we load the values into registers, then we apply the operation to them). By contrast, prefix notation is closely related to the way we express method calls in high level programming. For example we might write something like

```
compute_triangle_area(x, power(a,max(b,c)), sqrt(z))
```

where the three arguments are the lengths of the sides of a triangle. It is reasonable to call this prefix notation because we name each function and then list the values to which it is being applied (some of which are the result of other function calls).

Post-Order Traversal

Having seen In-Order and Pre-Order it will be no surprise that the third traversal algorithm is called Post-Order Traversal. As you can probably guess, the idea here is to explore the left subtree, then the right subtree, then the current vertex. As a recursive method it looks like this:

```
Post_Order(v):          # v is a vertex in a binary tree
    if v == nil:
        return
    else:
        Post_Order(v.left_child)
        Post_Order(v.right_child)
        print v.value
```

If we apply this to the tree shown above, the result is

$$4\ 3\ 10\ *\ +\ 2\ 8\ *\ -$$

which we can see is a correct postfix version of the arithmetic expression we are working with.

Now this is pretty amazing! We were able to store the expression in a simple data structure that let us extract all three ways of writing the expression (infix, prefix and postfix) using simple traversal algorithms.

You might want to think about how to implement these binary tree traversal algorithms non-recursively.

Here's a hint: one method is to use a stack as well as the tree.

Breadth-First Traversal

The fourth traversal algorithm that is widely used is called Breadth-First Traversal - we will look at it in some detail in other notes, but for now we can give an explanation of the idea: explore the tree one level at a time - so first we visit the root, then its children, then their children, then theirs, and so on down to the bottom of the tree.

Applying this to our tree *might* give

$$- \quad + \quad * \; 4 \; * \; 2 \; 8 \; 3 \; 10$$

The ambiguity comes from the lack of any rule about the order in which the vertices on each level are to be visited.

This is not as useful in terms of evaluating the expression because it is difficult to match the operations up with the operands - but breadth-first traversal has many other applications.

Complexity of In-Order, Pre-Order and Post-Order Traversals

Let's consider the complexity of In-Order, Pre-Order and Post-Order. If we let n be the number of vertices in the binary tree, you can see that in each of the three algorithms each vertex gets visited exactly once. Furthermore the event that brings us to a vertex, (ie. executing a recursive call in any one of the three algorithms), is exactly equivalent to following an edge of the tree. Since we know there are $n-1$ edges in a tree with n vertices (refer to the CISC-203 notes), the number of such operations is $n-1$. Referring to the algorithms given above we see that visiting a vertex takes $O(1)$ time. Thus we see that no matter what the actual structure of the tree (ie. whether it has many levels or few levels), these algorithms all take $O(n)$ time.

In case you don't remember (or didn't like) the proof from CISC-203 that the tree has $n-1$ edges, here is a different one:

Theorem: A binary tree with n vertices has $n-1$ edges.

Proof: Recall that in a rooted tree, every edge joins a parent to a child, and every vertex except the root has one edge that connects it to its parent. Thus there are $n-1$ edges joining vertices to their parents, and there aren't any other edges ... so the number of edges is $n-1$.