20200123

Binary Trees

Binary Tree: a rooted tree in which each vertex has at most two children. The children (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:

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 below.

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

A **Binary_Tree_Vertex** object needs:
- **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 siblings, parent, root, etc.

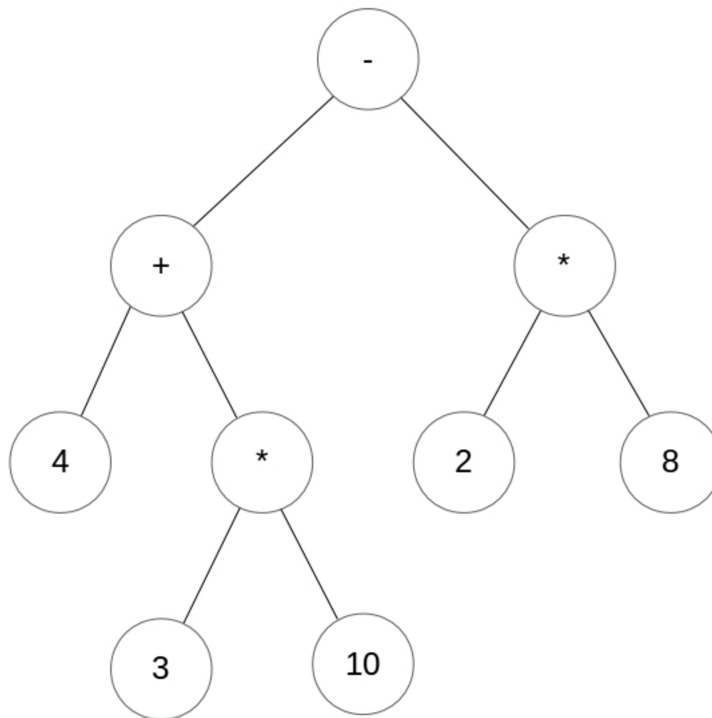
A **Binary_Tree** object needs:
- **root** (in a typed language, this is a pointer to a **Binary_Tree_Vertex** object)

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

Traversals of Binary Trees

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 class we looked at the traversal algorithms in a different order than I am presenting them in these notes. That order seemed to make sense then ... this order seems to make sense now. In CISC-235 we live in the moment!

The first tree traversal algorithm we will look at is called **In-Order Traversal**. The basic idea is to explore the left subtree, then look at the current vertex, then look at 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.

The next traversal algorithm to look at is **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

We didn't spend much time talking about "prefix notation" for arithmetic expressions but it's not complicated. In postfix notation each operator follows its operands ... so 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 method calls).

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 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 impressive! 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.