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. The tokens should look familiar – they are the tokens in the arithmetic expression we looked at earlier when we were discussing stacks:
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

\[ 4 + 3 \times 10 - 2 \times 8 \]

Well that’s interesting – this recreates the arithmetic expression in exactly its original form. This is not a coincidence – you might want to think about how I arranged the binary tree to make this happen.

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

\[ - + 4 \times 3 10 \times 2 8 \]

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 talked 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, b), 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 next 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 have seen before as 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: use a stack as well as the tree.
The fourth traversal algorithm that is widely used is called **Breadth-First Search** - we will look at it in some detail later, 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 gives

```
- + * 4 * 2 8 3 10
```

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 search has many other applications.

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 (we proved this in CISC-203), the number of such operations is \( n-1 \). Thus we see that no matter what the actual structure of the tree (i.e. 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:

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 \( n-1 \).

Now we turn to the most popular application of binary trees ... one that is found throughout computing.
Binary Search Trees
aka
Lexically Ordered Binary Trees

Suppose we have a collection $S$ of values and we want to perform the “search” operation. It comes in two flavours:

- Given $x$, is $x$ in $S$?
- Given $x$, what is the location of $x$ in $S$?

As always in this course, our concern is choosing the best structure in which to store $S$ to facilitate answering these questions.

Most often we are interested in the second question because we want to do something with $x$, such as access or modify information associated with $x$. If we are really only interested in the first question, there are structures that are particularly suited to that … as we will see later (dramatic foreshadowing).

We are already familiar with a structure that lets us search for $x$ very efficiently – our old friend, the lowly one-dimensional array. If we store $S$ in sorted order in an array, we can search $S$ in $\Theta(\log n)$ time, using binary (or even trinary) search.

So … end of story? Not quite.

If our set $S$ is fixed and unchanging, a sorted array is perfectly fine. But if we ever have to add or delete a value, the array is not a good choice at all: inserting or deleting values in a sorted array takes $\Omega(n)$ time.

The question then becomes: is there a data structure that allows searching, adding and deleting to all be completed in $\Theta(\log n)$ time?

The answer is yes, and of course since this discussion is lodged in the “Binary Tree” section of the course you will have guessed that this is the structure. But to facilitate the search operation we need to be more precise about how the values in $S$ will be stored in a binary tree.
When we store information in a binary tree there is no rule that says the information must be stored according to a specific pattern or rule. However, in order to use a binary tree to address the “search” problem we enforce a simple rule for the placement of the values in the tree: small values go to the left and large values go to the right. We can formalize this as follows:

**Binary Search Tree (BST):** a binary search tree is a binary tree in which each vertex contains an element of the data set. The vertices are arranged to satisfy the following additional property: at each vertex, all values in the left subtree are ≤ the value stored at the vertex, and all values stored in the right subtree, are > the value stored in the vertex. Note that we use "≤" for the left subtree to accommodate the possibility of having duplicate values in the tree.

In order to make a case for using a BST we need to determine the complexity of algorithms for the search, insert and delete operations, and then argue that they are superior to the algorithms for the same operations on an array or list.

**BST Searches**

Because of the ordering of the values in the vertices, searching a BST works just like binary search on a sorted array. We start at the root - if it contains the value we want, we are done. If not, we go to the left child or right child as appropriate.

Our design goal for implementing this data structure (and all subsequent ones) is that the user - in this case, the program which is calling the search function - should not need to know any details about the implementation of the structure. For example, the user should not need to know that the root of the tree is identified by an attribute called "root". For this reason, the function header is just `BST_Search(T,x)` where T is the tree to be searched and x is the search value. Of course if this function has been implemented as a method belonging to the tree, the function call would probably look like `T.BST_search(x)`.

We need to decide which flavour of search we are going to implement ("if x is there, return True" versus "if x is there, return its location"). We will opt for the latter, since it is neither easier nor more difficult with the BST structure. If x is in T, we return a pointer to the vertex containing it. If x is not in T, we return a null pointer.
BST_Search(T,x):
    current = T.root
    while current != nil:
        if current.value == x:
            return current
        elif current.value < x:
            current = current.right_child
        else:
            current = current.left_child
    return nil    # x is not in the tree

We can also implement the search algorithm recursively. We can use a "wrapper" function so that the interface does not change (this is better than the way I did it in class – the user should not need to know whether our algorithm is iterative or recursive):

BST_Search(T,x):
    return rec_BST_Search(T.root,x)

rec_BST_Search(current,x):
    if current == nil:
        return nil
    elif current.value == x:
        return current
    elif current.value > x:
        return rec_BST_Search(current.left_child,x)
    else:
        return rec_BST_Search(current.right_child,x)

You should convince yourself that these algorithms do indeed achieve the same result. It is easy to see that they have the same complexity since they visit exactly the same sequence of vertices.

Which of the two is better? The recursive version is marginally more elegant, but the iterative version is probably a bit more efficient - this is because a function call, in real life, takes longer to execute than an iteration of a loop. This means that even though the two algorithms have the same complexity, the constant multiplier for the iterative version may be smaller than the constant multiplier for the recursive version.

(At least, this is the conventional wisdom. As I discovered by experimenting in Python with recursive versus iterative implementations of Quicksort, it seems that recursive implementations of some algorithms may be faster than iterative implementations of the same algorithms. I encourage you to conduct some experiments to explore this question for yourself.)
Regardless of the difference in speed, I prefer the recursive version. As we will see when we look at more sophisticated algorithms for BSTs, there are times when using recursion is much, much cleaner than using iteration. Thinking about trees as recursive objects is a valuable exercise. Sometimes, even if the eventual goal is an iterative algorithm the best way to get there is to start by constructing a recursive algorithm, then convert the recursive calls into loops.

We can think of BST_Search() – either the recursive or the iterative version – as a modification of one of the three traversal algorithms we explored earlier. Which one?

We spent a bit of time in class looking at the process of building a BST from a set of numbers. We discovered that to find the proper place to insert a new value, we use the same sequence of branching (ie compare the new value to the root and then go either left or right as appropriate) and we eventually get to a point where the new value can be added as a new leaf of the tree. Using this method, each new value is placed exactly where it needs to be for the search algorithm to successfully find it. This means that the actual BST that we end up with is completely determined by the order in which we insert the values.

This means that we can write the insert algorithm as a simple modification of the search algorithm ... and that is what we will do on Monday.