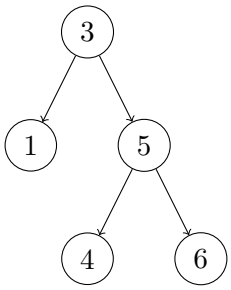
3.8 Traversals/Searches

There are various traversals that you can do on a binary tree.

3.8.1 Breadth First Traversal/Search (BFS)

The idea of a BFS is to travel along each level of the tree in order starting from its root. Consider the following tree:



What we want to do is traverse this tree in an order such that the output will be $3, 1, 5, 4, 6$. One way to do this is to rely on a queue and follow the algorithm:

```
Q <- new Queue
enqueue root
while Q is not empty:
    node <- dequeue from Q
    enqueue node.left
    enqueue node.right
    print node
```

Let's try this on the above tree. I'll just list the queue contents and what is printed after each iteration of the loop.

$$Q = [3] \qquad P = [] \tag{3.1}$$
$$Q = [1, 5] \qquad P = [3] \tag{3.2}$$
$$Q = [5] \qquad P = [3, 1] \tag{3.3}$$
$$Q = [4, 6] \qquad P = [3, 1, 5] \tag{3.4}$$
$$Q = [6] \qquad P = [3, 1, 5, 4] \tag{3.5}$$
$$Q = [] \qquad P = [3, 1, 5, 4, 6] \tag{3.6}$$
$$\tag{3.7}$$

3.8.2 Depth First Traversal/Search (DFS)

The idea of a DFS is to go as deep as possible into the tree as possible while searching. One way to achieve this is to replace the queue in the previous algorithm with a stack and enqueue/dequeue with push/pop respectively. Let's see how that goes on the same tree.

$$S = [3] \qquad P = [] \tag{3.8}$$
$$S = [1, 5] \qquad P = [3] \tag{3.9}$$
$$S = [1, 4, 6] \qquad P = [3, 5] \tag{3.10}$$
$$S = [1, 4] \qquad P = [3, 5, 6] \tag{3.11}$$
$$S = [1] \qquad P = [3, 5, 6, 4] \tag{3.12}$$
$$S = [] \qquad P = [3, 5, 6, 4, 1] \tag{3.13}$$
$$\tag{3.14}$$

An alternate way to perform a DFS is to take advantage of recursion to emulate the stack. It's as simple as

```
function dfs(node)
    dfs(node.right)
    dfs(node.left)
    print node
```

Notice that the output is actually different, however the idea was still the same; travel as deep into the tree as possible before doing anything.

3.8.3 Pre/In/Postorder Traversals

These traversals are pretty simple to remember if you memorize the format

```
function traverse(node)
    // pre
    traverse(node.left)
    // in
    traverse(node.right)
    // post
```

A preordering of a tree processes nodes in topological order. This means that the parent is processed before the children. This suggests that you should print the node before visiting the left and right children.

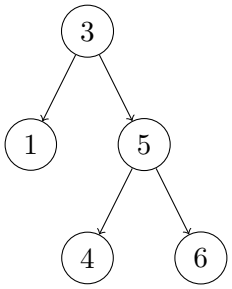
An inorder traversal actually gives the contents of the binary search tree well, in order. This is done by printing the node after visiting the left subtree but before visiting the right, or as I prefer to say: in between the recursive calls.

A postorder traversal is the final kind of tree traversal and can be executed by printing the node after traversing both left and right subtrees.

If we were to perform these three traversals on the above tree, we would get

$$preorder = [3, 1, 5, 4, 6] \tag{3.15}$$
$$inorder = [1, 3, 4, 5, 6] \tag{3.16}$$
$$postorder = [1, 4, 6, 5, 3] \tag{3.17}$$

However, there is a simpler way to perform these traversals without stepping through the algorithm by hand. Let's look back at our tree:



We notice that our left right recursive calls more or less traverse the tree in a fashion that outlines the structure of the tree starting from the left of the root; $[3, 1, 5, 4, 6]$. All we have to do to mimic when we print the node is to mark the node at a spot that represents when it is printed i.e. these 3 spots.

I'll leave it to you to figure out which spot is for which traversal.