Tree traversal

Often we wish to iterate through or "traverse" all the nodes of the tree. We generally use the term *tree traversal* for this. There are two aspects to traversing a tree. One is that we need to follow references from parent to child, or child to its sibling. The second is that we may need to do something at each node. I will use the term "visit" for the latter. Visiting a node means doing some computation at that node. We will see some examples later.

Depth first traversal

The first two traversals that we consider are called "depth first". In these traversals, a node and all its descendents are visited before the next sibling is visited. There are two ways to do depth-first-traversal of a tree, depending on whether you visit a node before its descendents or after its descendents. In a *pre-order* traversal, you visit a node, and then visit all its children. In a *post-order* traversal, you visit the children of the node (and their children, recursively) and then visit the node.

```
depthfirst_Preorder(root){
  if (root is not empty){
    visit root
    for each child of root
      depthfirst_Preorder(child)
  }
}
```

An example is illustrated in the lectures slides. Suppose we have a file system. The directories and files define a tree whose internal nodes are directories and whose leaves are either empty directories or files. We first wish to print out the root directories, namely list the subdirectories and files in the root directory. For each subdirectory, we also print its subdirectory and files, and so on. This is all done using a pre-order traversal. The visit would be the print statement. Here is a example of what the output of the print might look like. (This is similar to what you get on Windows when browsing files in the Folders panel.)

```
My Documents                    (directory)
   My Music                     (directory)
      Raffi                     (directory)
         Shake My Sillies Out   (file)
         Baby Beluga            (file)
      Eminem                    (directory)
         Lose Yourself          (file)
   My Videos                    (directory)
      :                         (file)
   Work                         (directory)
      COMP250                   (directory)
          :
```

For a postorder traversal, one visits a node after having visited all the children of the node.

```
depthfirst_Postorder(root){
   if (root is not empty){
      for each child of root
         depthfirst_Postorder(child){
      visit root
   }
}
```

Let's look at an example of post-order traversal. Suppose we want to calculate how many bytes are stored in all the files within some directory including all its sub-directories. The reason this is post-order is that, in order to know the total bytes in some directory, we first need to know the total number of bytes in all the subdirectories Hence, we need to visit the subdirectories first. Here is an algorithm for computing the number of bytes. It traverses the tree in postorder in the sense that it computes the sum of bytes in each subdirectory by summing the bytes at each child node of that directory. (Frankly, in this example it is a big vague what I mean by "visit", since computing `sum` doesn't just happen after visiting the children but rather involves steps that occur before, during, and after visiting the children. The way I think of this is that if we were to store `sum` as a field in the node, then we could only do this after visiting the children.)

```
numBytes(root){
  if root is a leaf
     return number of bytes at root
  else{
     sum = 0    // local variable
     for each child of root{
        sum += numBytes(child)}
     return sum
  }
}
```

Depth first traversal without recursion

As we have discussed already in this course, recursive algorithms are implemented using a call stack which keep track of information needed in each call. (Recall the stack lecture.) You can sometimes avoid recursion by using an explicit stack instead. Here is an algorithm for doing a pre-order depth first traversal which uses a stack rather than recursion. As you can see by running an example (see lecture slides), this algorithm visits the list of children of a node in the opposite order to that defined by the `for` loop.

At first glance, this algorithm seems to be pre-order, not post-order since the `visit cur` statement occurs prior to the `for` loop. However, this is not the essential reason why the algorithm is pre-order: even if we were to move the `visit cur` statement to be after the `for` loop, the algorithm would still be pre-order. The essential reason the traversal is pre-order is that a node is visited before its children are visited.

```
treeTraversalUsingStack(root){
  s.push(root)
  while !s.isEmpty(){
    cur = s.pop()
    visit cur
    for each child of cur
       s.push(child)
  }
}
```

Breadth first traversal

What happens if we use a queue instead of a stack in the previous algorithm?

```
treeTraversalUsingQueue(root){
   q = empty queue
   q.enqueue(root)
   while !q.isEmpty() {
     cur = q.dequeue()
     visit cur
     for each child of cur
        q.enqueue(child)
}
```

As shown in the example in the lecture, this algorithm visits all the nodes at each depth, before proceeding to the next depth. This is called *breadth first* traversal. The queue-based algorithm effectively does the following:

```
for i = 0 to height
    visit all nodes at level i
```

You should work through the example in slide 30 of the PDF and make sure you understand why using queue here is diffrent from using a stack.

A note about implementation

Recall first-child/next-sibling data structure for representing a tree, which we saw last lecture. Using this implementation, you can replace the line

```
for each child of cur
   ...
```

with the following

```
child = child.firstChild
while (child != null){
   ...                          //  maybe do something at that child
  child = child.nextSibling
}
```

Here we are iterating through a (singly linked) list of children.