

Tree traversal

Often we wish to examine (or visit) all the nodes of the tree. This is called *tree traversal*, or *traversing* a tree. 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 need to “visit” the node. By “visit”, I mean doing some computation, e.g. getting or setting a field of an element referenced by that node.

There are several different ways in which we can traverse a tree. They differ in the order in which the nodes are visited.

Depth first traversal

The first two traversals we look at 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.

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

postOrderTraversal(root){
  if (root is not empty){
    for each child of root
      postorderTraversal(root.child){
        visit root
      }
  }
}
```

Two examples are 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 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 subdirector 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 XP 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)
Kids skiing at Tremblant	(file)
Work	(directory)
COMP250	(directory)
etc	

Next let's look at an example of post-order traversal. Suppose we want to calculate how many bytes there are stored in all the files within some directory (including sub-directories, etc). The reason this is post-order is that, to know the total bytes in some directory, I first need to get the bytes in all the subdirectories. Hence, I need to visit these nodes first. Here is an algorithm for computing the number of bytes. Note that 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.

```
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

Recursion uses an (implicit) stack to keep track of where to return from a method call. You can often avoid recursion without much trouble by using an explicit stack instead. Here is an algorithm for doing a pre-order depth first traversal which uses a stack but does not use recursion. As you can see by running an example (see lecture slides), this algorithm visits the children of a node in the opposite order to that defined by the "for each" statement. The algorithm is still pre-order though, since it visits each node before visiting its children.

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

Breadth first traversal

The opposite of a depth first traversal is a breadth first traversal. (Note that it is not recursive.)

```
for each level i in the tree // level is the same as depth
  visit all nodes at level i
```

Here is a more detailed sketch of how to implement this algorithm, by using a queue instead of a stack.

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

To be more specific, you can replace the last two lines of the above algorithm as follows. Recall first-child/next-sibling data structure for representing a tree, which we saw last lecture.

Then

```
//   for each child of cur           //   REPLACE THESE TWO LINES
//       s.enqueue(cur.child)       //
//
cur = cur.firstChild
while (cur != null){
  q.enqueue(cur)
  cur = cur.nextSibling }
```

Why would you want to do a breadth first traversal? Let's consider an example of a two player game, such as chess. Let the root node be the current state of the game, and suppose it is your turn to move. You have several possible moves you could make. Each of these moves defines a child node of the root, which would be the next state of the game if you were to make that move. Since it would then be your opponent's move, each of these child nodes would define another set of states (the child node's children) which the game would be in after your opponent has made its next move, etc. One way to think of a chess strategy is to consider each of your next moves, and then consider your opponent's moves, and then your moves that would follow. You can quickly see the number of states to be searched would rise very fast with the number of levels of your breadth first search. (And those of you who play chess might realize that there are many ways to "prune" the branches of these searches to save time in planning. If you want to learn more about these types of search problems, then take COMP 424 Intro to AI.)

Example

See the slides and exercises.