COMP 250
Lecture 18

tree traversal

Oct. 21, 2016
How to enumerate/iterate through/traverse/visit/... the nodes of a tree?
```java
depthfirst (root){
    // “preorder”
    if (root is not empty){
        visit root
        for each child of root
            depthfirst( child )
    }
}
```
depthfirst (root) {
    if (root is not empty) {
        visit root
        for each child of root
            depthfirst(child)
    }
}
depthfirst (root) { // "preorder"
    if (root is not empty) {
        visit root
        for each child of root
            depthfirst (child)
    }
}
“Visit” implies that you do something at that node.

Analogy: you aren’t visiting London UK if you are just flying through Heathrow.
e.g. Printing a file hierarchy
depthfirst (root) 
   if (root is not empty) 
      for each child of root 
         depthfirst( child ) 
   visit root
depthfirst (root) {
    if (root is not empty) {
        for each child of root
            depthfirst ( child )
        visit root
    }
}
depthfirst (root) {
    if (root is not empty) {
        for each child of root
            depthfirst( child )
    visit root
}
}
depthfirst (root){
    if (root is not empty){
        for each child of root
            depthfirst( child )
    
    visit root
}
}
depthfirst (root){
    if (root is not empty){
        for each child of root
            depthfirst( child )
    visit root
}
}
e.g. What is the total number of bytes in all files?
numBytes(root) {
  if root is a leaf
    return number of bytes at root
  else {
    sum = 0
    for each child of root{
      sum += numBytes(child)
    }
    return sum
  }
}

Q: What do we mean by visit here?
A: Determining the number of bytes for a node, e.g. If we were to store ‘sum’ at the node.
Call stack for depthfirst()

Letters are names of nodes (happen to be ordered by calls).

Same call sequence occurs for preorder vs postorder.
Tree traversal

Recursive
  • depth first (pre- versus post-order)

Non-Recursive
  • using a stack
  • using a queue
```java
void treeTraversalUsingStack(TreeNode root) {  
    initialize empty stack s  
    s.push(root)  
    while s is not empty {  
        cur = s.pop()  
        visit cur  
        // moving ‘visit cur’ to be  
        for each child of cur  
        s.push(child)  
        // after for loop  
        // changes nothing  
    }  
}
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    } 
}
treeTraversalUsingStack(root) {
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}

Here I am using same label nodes as in the pre-order recursive depthfirst() example.
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }
}
```
Stack based method is depth first but visits children from right to left

```
recursive
```

```
non-recursive (stack)
```

```plaintext
recursive  abcdefghijk
non-recursive (stack)  ahkjigbdfe}
```
What if we use a queue instead?

def treeTraversalUsingStack(root):
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        for each child of cur
            s.push(child)
    }

def treeTraversalUsingQueue(root):
    initialize empty queue q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
treeTraversalUsingQueue(root){
    initialize empty queue q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
treeTraversalUsingQueue(root){
    initialize empty queue q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}

Queue state at start of the while loop

a
bcd
```python
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
```

Queue state at start of the while loop:

```
    a
   / \
  b   c
 / \ / \ \
e  f  g  h
 /   /   / \
 J   k   i
```

Children of 'a':
- b
- c
- d

Children of 'd':
- g
- h
- i

Children of 'b':
- e
- f

Children of 'f':
- j
- k
treeTraversalUsingQueue(root){
    initialize empty queue q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}

Queue state at start of the while loop

```
a
bcd
cdef
def
```
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}

Queue state at start of the while loop

a
bcd
cdef
def
efghi
treeTraversalUsingQueue(root){
    initialize empty queue  q
    q.enqueue(root)
    while q is not empty {
        cur = q.dequeue()
        visit cur
        for each child of cur
            q.enqueue(child)
    }
}
breadth first traversal

for each level i
visit all nodes at level i

order visited:  abcdefghijk
Tree traversal

Recursive
- depth first (pre- versus post-order)

Non-Recursive
- depth first (uses a stack)
- breadth first (uses a queue)
Implementation Details

Recall: ‘first child, next sibling’

class TreeNode<T>{
    T element;
    TreeNode<T> firstChild;
    TreeNode<T> nextSibling;
    ...
}

class Tree<T>{
    TreeNode<T> root;
    ...
}
Recall: ‘first child, next sibling’

for each child{
    ...
}

This means:

child = cur.firstChild
while (child != null){
    .....  
    child = child.nextsibling
}