COMP 250

Lecture 20

tree traversal

Oct. 25/26, 2017
Tree Traversal

How to visit (enumerate, iterate through, traverse...) all the nodes of a tree?
depthfirst (root) {
    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)
    }
}

// “preorder”
depthfirst (root) {
    if (root is not empty) {
        visit root
        for each child of root
            depthfirst (child)
    }
}
def depthfirst (root):
    if (root is not empty):
        visit root
        for each child of root
            depthfirst( child )
    // “preorder”
Preorder Traversal
e.g. Printing a directory \((\text{visit} = \text{print})\)
“Visit” implies that you do something at that node.

Analogy: you aren’t visiting London UK if you just fly through Heathrow.
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) {
    // “postorder”
    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
    }
}

“postorder”
Example 1 postorder: recall last lecture

height(v) {
  if (v is a leaf)
    return 0
  else {
    h = 0
    for each child w of v
      h = max(h, height(w))
    return 1 + h
  }
}

visit = return value of height
Example 2  Postorder: What is the total number of bytes in all files in a directory?
```c
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
    }
}
```

By ‘visit’ here, we mean determining the number of bytes for a node, e.g. If we were to store ‘sum’ at the node.
NOTE: Same call sequence occurs for preorder vs postorder.

Letter order corresponds to `depthfirst()` call order

```
        a
       / \
      b   g
     / \  / \  
    c   d i   j
    /   /   /  
   e   f   k   
```
Call stack for `depthfirst()`
Call stack for `depthfirst()`
Call stack for depthfirst()
Call stack for `depthfirst()`
Tree traversal

Recursive

- depth first (pre- versus post-order)

Non-Recursive

- using a stack
- using a queue
treeTraversalUsingStack(root){
    initialize empty stack s
    s.push(root)
    while s is not empty {
        cur = s.pop()
        visit cur
        // moving 'visit cur' to be after for loop
        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
    }
}
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)
    }
}
What is the order of nodes visited?
treeTraversalUsingStack\(\text{root}\)\{
initialize empty stack \(s\)
\(s\).push(\text{root})
while \(s\) is not empty {
  \text{cur} = \text{\(s\).pop()}\)
  visit \text{cur}
  for each child of \text{cur}
    \(s\).push(\text{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)
  }
}
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)
  }
}
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)
    }
}
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)
    }
}
```plaintext
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: abcdefghijk

Non-recursive (stack): ahkjjigbdfec
Pre- or post order?

```java
void treeTraversalUsingStack(TreeNode 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)
        visit cur
    }
}
```

Moving the visit does not make it post order. Why not?
What if we use a queue instead?

treeTraversal UsingStack (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)
    }
}

treeTraversal UsingQueue (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)
    }
}
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)
    }
}
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:  \text{abcdefghijk}
Implementation Details

Recall: ‘first child, next sibling’

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

class Tree<T> {
    TreeNode<T>    root;
    ...
    ...
}
for each child{
    ...
}

means:

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