COMP 250

Lecture 24

tree traversal

March 9, 2022
Tree Traversal

How to “visit” / “traverse” / “iterate through” the nodes in a tree?
Tree Traversal

How to “visit” / “traverse” / “iterate through” the nodes in a tree?
Recall – depth of a node

The depth or level of a node is the length of the path from the root to the node.

```
depth (level)
0
1
2
3
```
Tree Traversal

• Depth first traversal  (also called depth first search)

  \textit{Start from root and follow paths to the leaves, backtracking only when a leaf is found}

• Breadth first traversal  (also called breadth first search)

  \textit{Start from root and visit all nodes at depth k before any nodes at depth k+1.}
“Visit” a node implies that you do something at that node.

We will see some examples later.
depthfirst (root) {
    visit root
    for each child of root
        depthfirst (child)
}

```
```
```plaintext
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}
```

"preorder" traversal: visit the root before the children

Visit the root first.
**depthfirst** (root) {
  visit root
  for each child of root
    **depthfirst** (child)
}

Assume we visit children from left to right.
depthfirst (root){
    visit root
    for each child of root
        depthfirst( child )
}

This node has no children.
depthfirst (root) {
    visit root
    for each child of root
        depthfirst (child)
}

[Diagram of a tree with nodes labeled 1, 2, 3, and 4, illustrating the depth-first traversal order.]
depthfirst (root) {
    visit root
    for each child of root
        depthfirst( child )
}
depthfirst (root) {
    visit root
    for each child of root
        depthfirst( child )
}
depthfirst (root) {
  visit root
  for each child of root
    depthfirst( child )
}
Implementation details

Recall the “first child, next sibling” implementation

Then when we write

```java
for each child of root {
  
}
```

we would mean:

```java
child = root.firstChild
while (child !=null) {
  
  child = child.nextSibling
}
```
Example of Preorder Traversal: printing a hierarchical file system
(visit = print directory or file name)
depthfirst (root) {
    for each child of root
        depthfirst (child)
    visit root
}
depthfirst (root) {
    for each child of root
    depthfirst (child)
    visit root
}

“postorder” traversal: visit the root after the children

Q: Which node is visited first?
depthfirst (root) {
  for each child of root
    depthfirst( child )
  visit root
}

“postorder” traversal: visit the root after the children

Q: Which node is visited second?
depthfirst (root) {
    for each child of root {
        depthfirst (child)
    }
    visit root
}

“postorder” traversal: visit the root after the children

Q: Which node is visited 3rd and 4th?
\texttt{depthfirst}(\texttt{root})\
\hspace{1em} \text{if (root is not empty)\{\
\hspace{2em} \text{for each child of root}\
\hspace{3em} \texttt{depthfirst}(\texttt{child})\
\hspace{2em} \text{visit root}\
\hspace{1em} \}}\

"postorder" traversal: visit the root after the children
depthfirst (root) {
    for each child of root
        depthfirst(child)
    visit root
}
`visit` a node here means `determine height of` the node
Example 2 Postorder:
total number of bytes (in a file or in some directory including subdirectories)
numBytes(root) {
    if root has no children
        return number of bytes at root  // 0, if root is directory
    else {
        sum = 0
        for each child of root
            sum += numBytes(child)
        return sum
    }
}

‘visit’ here means determine the number of bytes in subtree that is rooted at that node.
“preorder” traversal

\[
\text{depthfirst} \ (\text{root})\{
    \text{visit root}
    \text{for each child of root}
      \text{depthfirst}( \text{child} )
\}
\]

“postorder” traversal

\[
\text{depthfirst} \ (\text{root})\{
    \text{for each child of root}
      \text{depthfirst}( \text{child} )
    \text{visit root}
\}
\]
The same \texttt{depthfirst(root)} \textit{call sequence} occurs for preorder and postorder. Only the visiting order changes.

In example below, the letter order corresponds to the \texttt{depthfirst(root)} \textit{call order}. Let’s next examine the call stack.
Call stack for `depthfirst(root)`

The “call stack” figure below only shows the `root` parameter.
Call stack for *depthfirst(root)*

The “call stack” figure below only shows the root parameter.
Call stack for `depthfirst(root)`

The “call stack” figure below only shows the `root` parameter.
Call stack for `depthfirst(root)`
Call stack for $\text{depthfirst}(\text{root})$
Call stack for `depthfirst(root)`
Call stack for depthfirst(root)
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)

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 by `treeTraversalUsingStack()`?
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)
    }
}
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)
    }
}
The stack based method is also depth first, but we are visiting children from right to left.

recursive preorder : abcdefghijk
recursive postorder : cefdbgijkha

non-recursive (stack) : ahkjigbdfec
What if we use a queue instead?

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)
  }
}

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)
    }
}
treeTraversal\textbf{UsingQueue}(root)\
\quad \text{initialize empty queue } q \\
\quad q\text{.enqueue}(root) \\
\bullet \text{ while q is not empty } \{ \\
\quad \text{cur} = q\text{.dequeue}() \\
\quad \text{visit cur} \\
\quad \text{for each child of cur} \\
\quad \quad q\text{.enqueue}(child) \\
\} \\
\}
\textbf{treeTraversalUsingQueue}(root)\
\hspace{1cm} initialize empty queue \ q
\hspace{1cm} \text{q.enqueue}(\text{root})
\hline
\begin{itemize}
\item while \ q \ is \ not \ empty 
\hspace{1cm} \text{cur} = \text{q.dequeue}()
\hspace{1cm} \text{visit cur}
\hspace{1cm} \text{for each child of cur}
\hspace{1cm} \text{q.enqueue(child)}
\end{itemize}
\hline

Queue state at start of the while loop

\hspace{1cm} a
\hspace{1cm} b c d
\hspace{1cm} c d e f
\hspace{1cm} d e f
```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 d
c d e f
d e f
e f g h i
```
treeTraversa**lUsingQueue**(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 d
  e f g h i
```

```
  a
  b c d
  e f g h i
  j k
```
treeTraversal\textbf{UsingQueue}(root)\
    initialize empty queue \ space \ q\
    q.\texttt{enqueue}(root)\
    \begin{itemize}
        \item while q is not empty {\
            \begin{itemize}
                \item cur = q.\texttt{dequeue}()\
                \item visit cur\
                \item for each child of cur \space q.\texttt{enqueue}(child)
            \end{itemize}
        }
    \end{itemize}
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
Coming up...

**Lectures**

- Fri. March 11  (lecture 25)
  Expression Trees

- Mon. March 14
  Binary Search Trees

- Wed & Fri. March 16 & 18
  Heaps

**Tutorial + Assessments**

- **Tutorial for Assignment 3**
  today at 6 pm

- Assignment 3
  due  Wed. March 16

- Quiz 4  (lectures 20-25)
  Fri. March 18