ECE430.217 Data Structures

Tree Traversals

Weiss Book Chapter 4.1

Byoungyoung Lee
https://compsec.snu.ac.kr
byoungyoung@snu.ac.kr

Lecture slides were prepared based on Douglas W. Harder's notes (**dwharder@alumni.uwaterloo.ca).**

Outline

This topic will cover tree traversals:

- A means of visiting all the objects in a tree data structure
- We will look at
  - Breadth-first traversals
  - Depth-first traversals
- Applications
- General guidelines

Background

All the objects stored in an array or linked list can be accessed sequentially

Question: how can we iterate through all the objects in a tree in a predictable and efficient manner

– Requirements: $\Theta(n)$ run time and $o(n)$ memory

Types of Traversals

We have already seen one traversal:

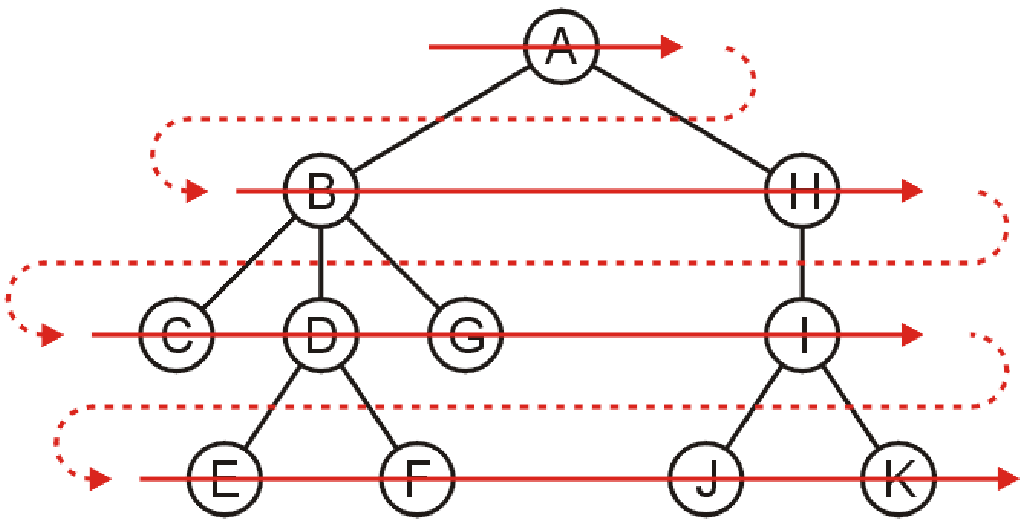
- The **breadth-first traversal** visits all nodes at depth k before proceeding onto depth $k + 1$
- Easy to implement using a queue

Another approach is to visit always go as deep as possible before visiting other siblings: ***depth-first traversals***

Breadth-First Traversal

Breadth-first traversals visit all nodes at a given depth

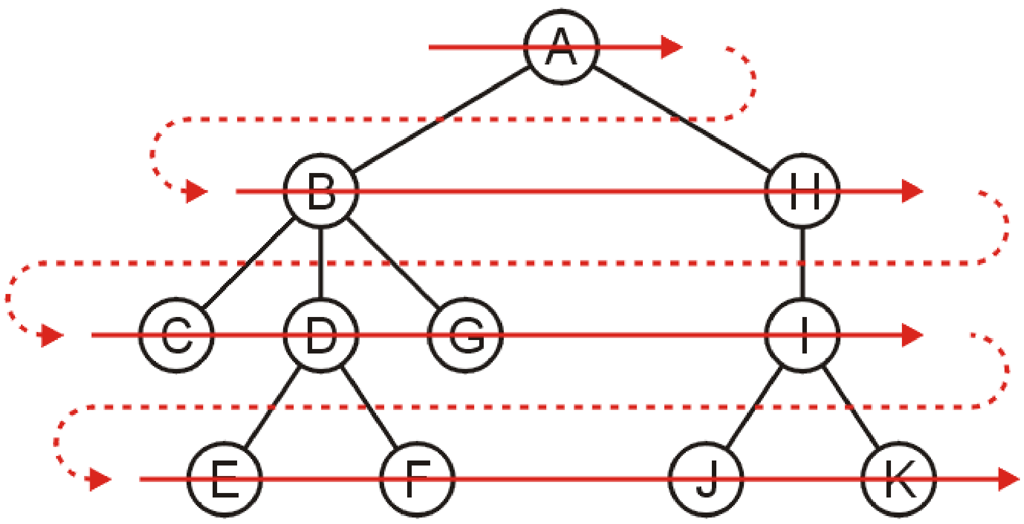
- Can be implemented using a queue
- Run time is $\Theta(n)$
- Memory is potentially expensive: maximum nodes at a given depth
- Order: A B H C D G I E F J K



Breadth-First Traversal

The implementation was already discussed:

- Create **a queue** and push the root node onto the queue
- While the queue is not empty:
  - Push all of its children of the front node onto the queue
  - Pop the front node

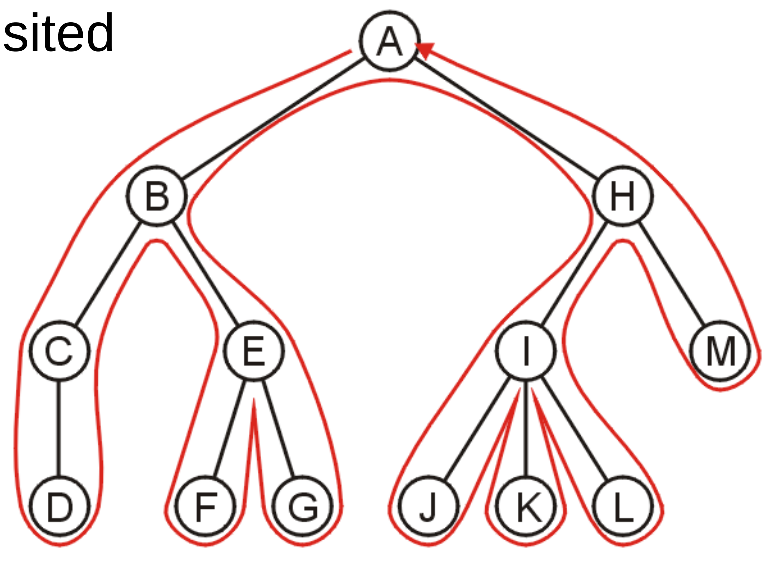


Backtracking

To discuss depth-first traversals, we will define **a backtracking algorithm** for stepping through a tree:

- At any node, we proceed to the first child that has not yet been visited
- Or, if we have visited all the children, we backtrack to the parent and repeat this decision making process

We end once all the children of the root are visited

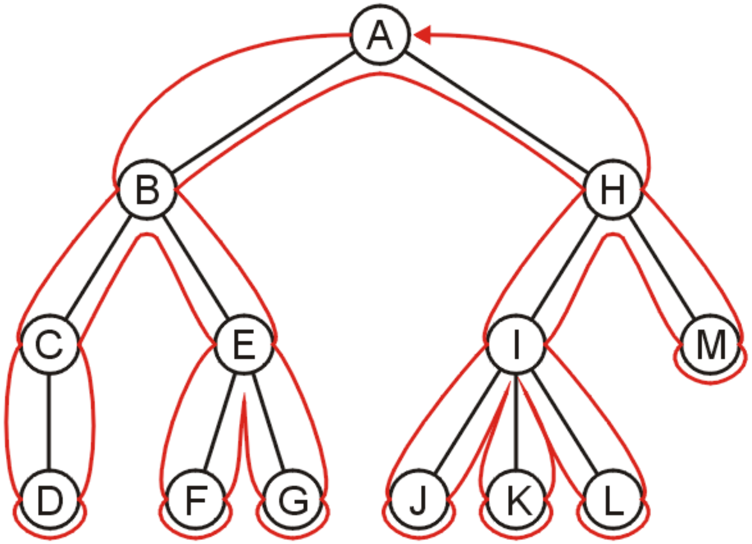


Depth-first Traversal

We define such a path as a ***depth-first traversal***

We note that each node **could be visited twice** in such a scheme

- The first time the node is approached (before any children)
- The last time it is approached (after all children)



Implementing depth-first traversals

Depth-first traversals can be implemented with recursion:

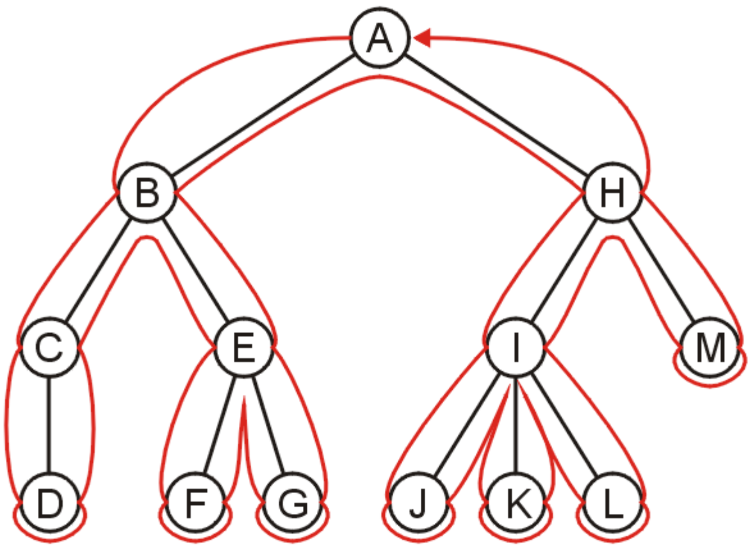
```
template <typename Type>
void print_dfs_tree(SimpleTree<Type> *tree, int depth=0) {
    // Perform pre-visit operations on the value
    std::cout << std::string(depth*4, ' ') << "<" << tree->get_value() << ">" << std::endl;

    // Perform a depth-first traversal on each of the children
    for (int i=0; i<tree->get_degree(); i++) {
        auto ctree = tree->get_child(i);
        print_dfs_tree(ctree, depth+1);
    }

    // Perform post-visit operations on the value
    std::cout << std::string(depth*4, ' ') << "</" << tree->get_value() << ">" << std::endl;
}
```

Implementing depth-first traversals

Performed on this tree, the output would be




```
<A>
    <B>
        <C>
            <D>
            </D>
        </C>
        <E>
            <F>
            </F>
            <G>
            </G>
        </E>
    </B>
    <H>
        <I>
            <J>
            </J>
            <K>
            </K>
            <L>
            </L>
        </I>
        <M>
        </M>
    </H>
</A>
```

Implementing depth-first traversals

Alternatively, we can use a stack:

- Create a stack and push the root node onto the stack
- While the stack is not empty:
  - Pop the top node
  - Push all of the children of that node to the top of the stack in reverse order
- Run time is $\Theta(n)$
- The objects on the stack are all unvisited siblings from the root to the current node
  - If each node has a maximum of two children, the memory required is $\Theta(h)$: the height of the tree

With the recursive implementation, the memory is $\Theta(h)$

Applications

Tree application: displaying information about directory structures and the files contained within

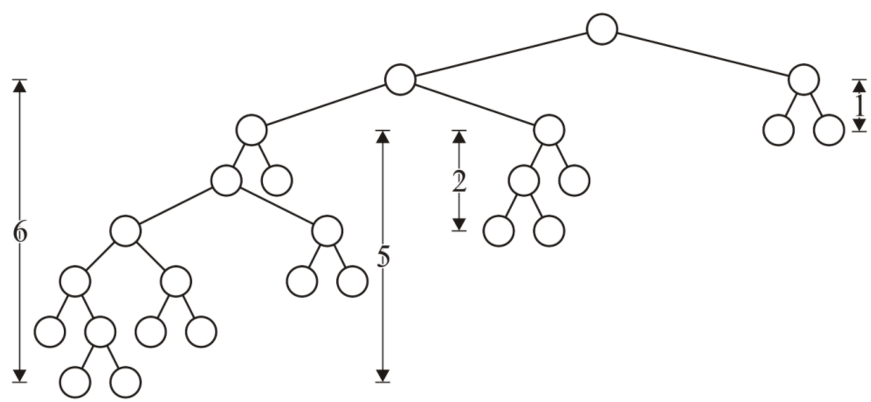
- Finding the height of a tree
- Printing a hierarchical structure

Height

The `int height() const` function is recursive in nature:

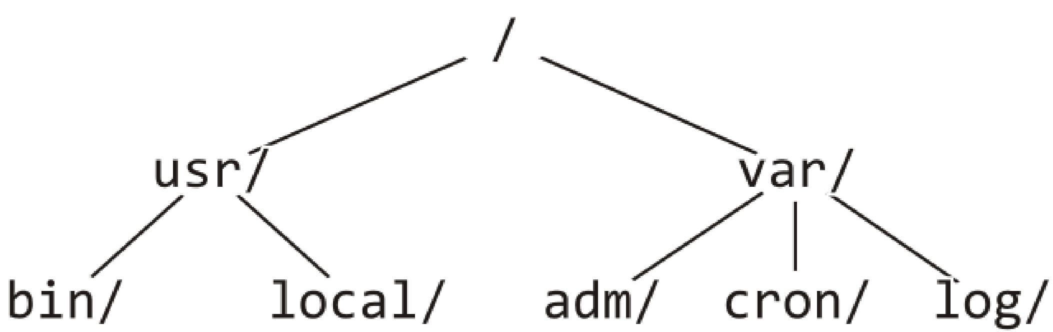
1. Before the children are traversed, we assume that the node has no children and we set the height to zero: $h_{\text{current}} = 0$
2. In recursively traversing the children, each child returns its height h and we update the height as $\max(1 + h,\, h_{\text{current}})$
3. Once all children have been traversed, we return h_{current}

When the root returns a value, that is the height of the tree



Printing a Hierarchy

Consider the directory structure presented on the left—how do we display this in the format on the right?




```
/
    usr/
        bin/
        local/
    var/
        adm/
        cron/
        log/
```

What do we do at each step?

Printing a Hierarchy

For a directory, we initialize a tab level at the root to 0

We then do:

1. Before the children are traversed, we must:
   a) Indent an appropriate number of tabs, and
   b) Print the name of the directory followed by a `'/'`
2. In recursively traversing the children:
   a) A value of one plus the current tab level must be passed to the children, and
   b) No information are passed back
3. Once all children have been traversed, we are finished

Summary

This topic covered two types of traversals:

- Breadth-first traversals
- Depth-first traversals
- Applications
- Determination of how to structure a depth-first traversal