CSE 203: Tree Traversal

Dr. Mohammed Eunus Ali

Professor

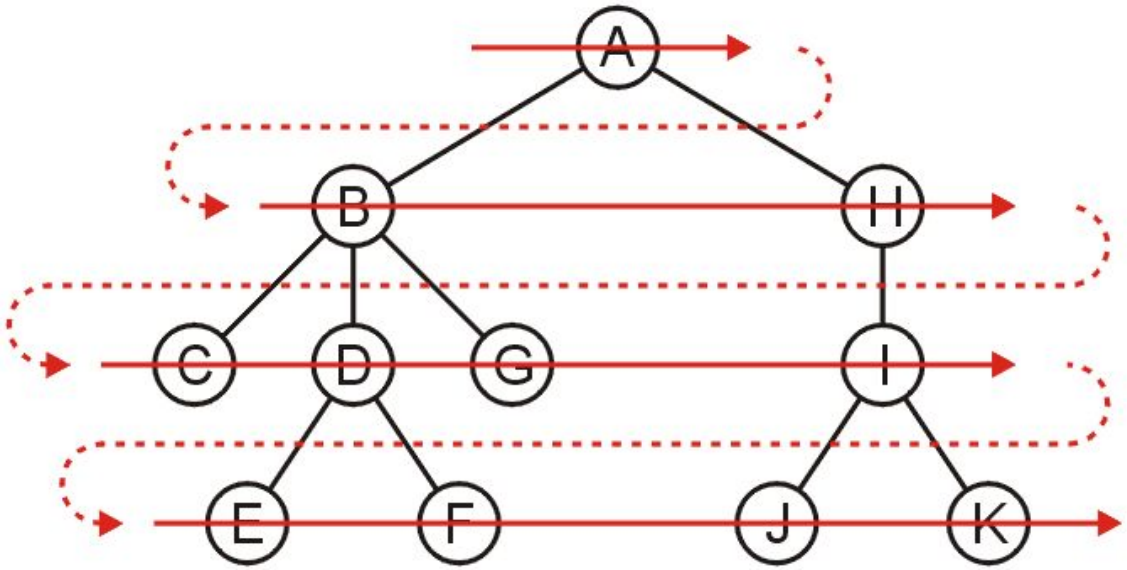
CSE, BUET

Slides are from Douglas Wilhelm Harder, dwharder@alumni.uwaterloo.ca

Breadth-First Traversal

Breadth-first traversals visit all nodes at a given depth

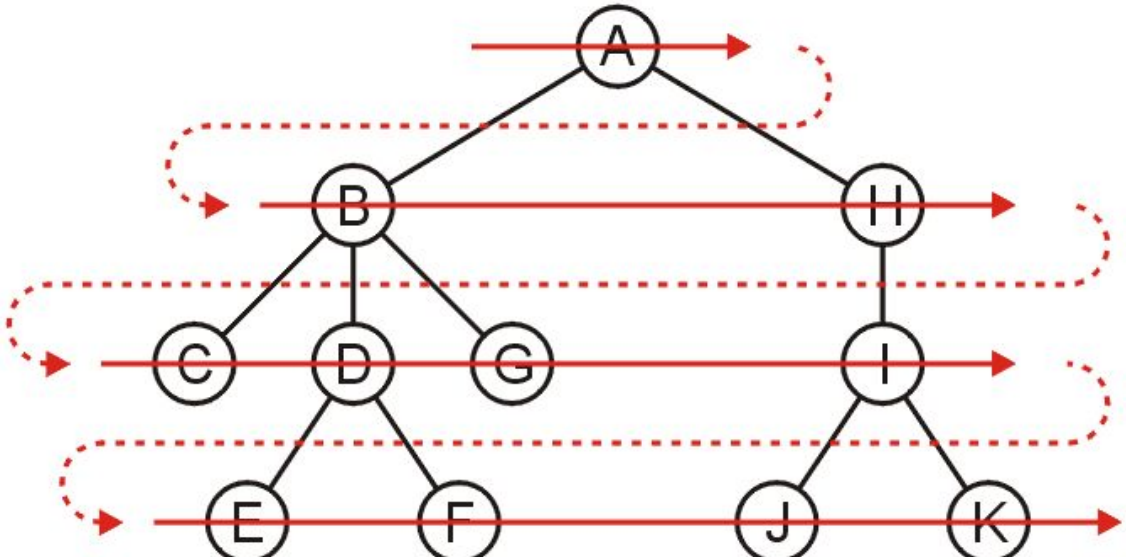
- Can be implemented using a queue
- Run time is $\Theta(n)$
- Memory is potentially expensive: maximum nodes at a given depth
- Order: A B H C D G



Breadth-First Traversal

The implementation:

- 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

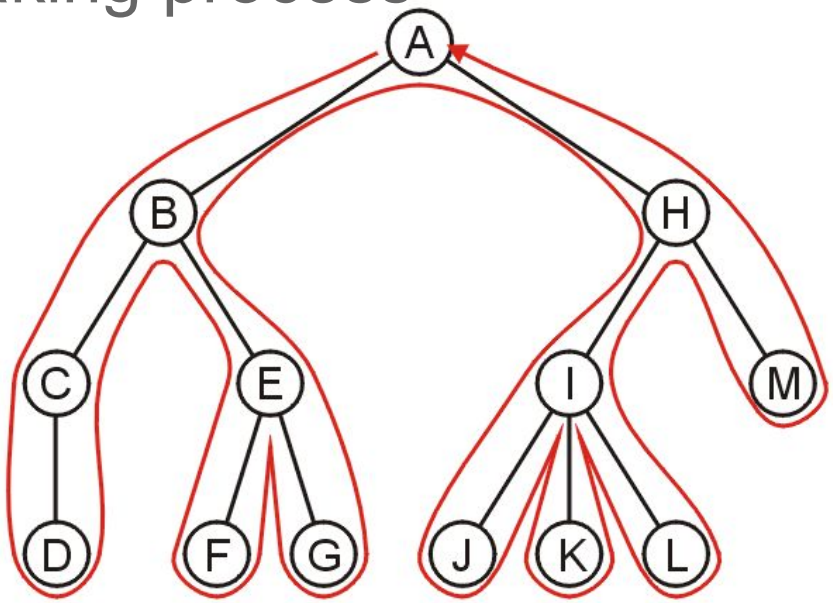


Backtracking

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 (of which a leaf node is a special case), we backtrack to the parent and repeat this decision making process

We end once all the children of the root are visited

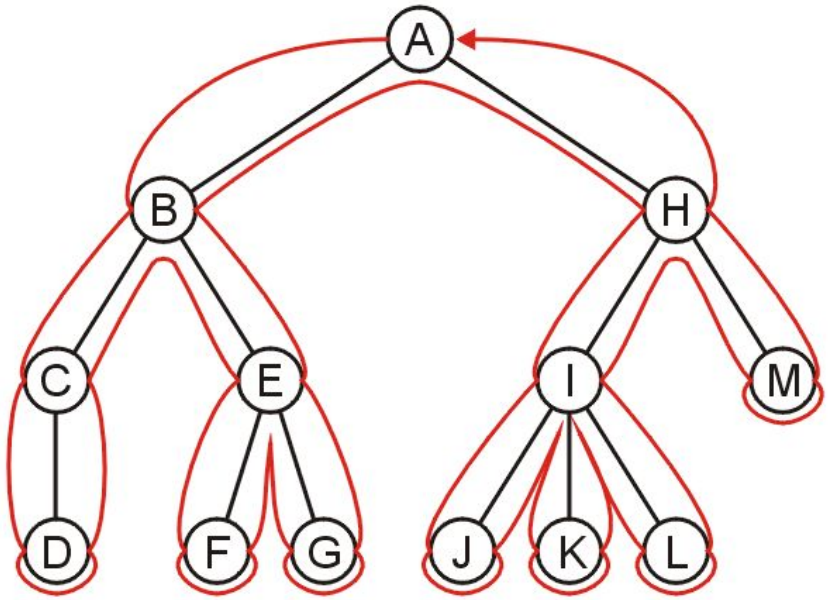


Depth-first Traversal

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

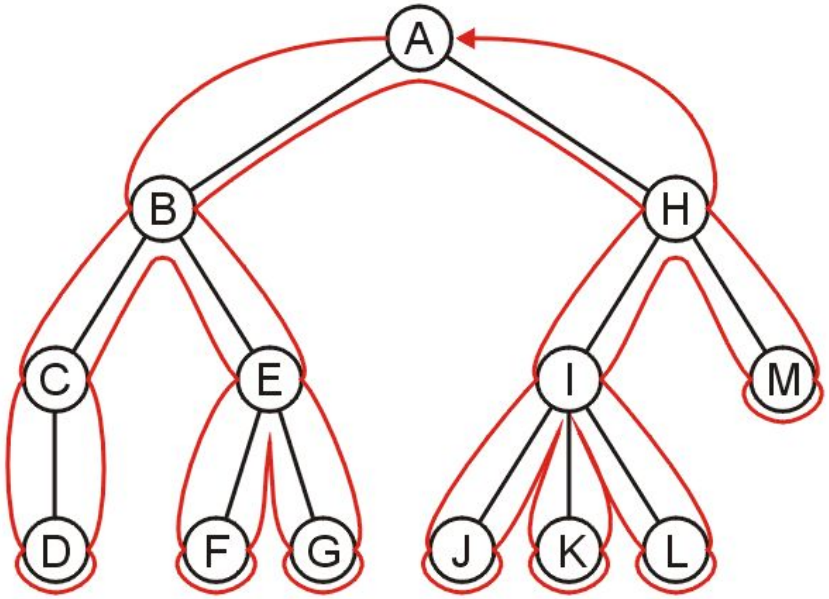
Depth-first traversals can be implemented with recursion:

```
 template <typename Type>
depth_first_traversal() const {
    // Perform pre-order visit operations on the value
    std::cout << "<" << node_value << ">";

    // Perform a depth-first traversal on each of the children
    for ( auto *child = children.head(); child != children.end();
          child = ptr->next() ) {
        child->value()->depth_first_traversal();
    }

    // Perform post-order visit operations on the value
    std::cout << "</" << node_value << ">";
}
```

Implementing depth-first traversals

Performed on this tree, the ouput 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 $\Theta(h)$: recursion just hides the memory

Guidelines

Depth-first traversals are used whenever:

- The **parent** needs information about all its children or **descendants**, or
- The **children** require information about all its parent or **ancestors**

In designing a depth-first traversal, it is necessary to consider:

1. Before the children are traversed, what initializations, operations and calculations must be performed?
2. In recursively traversing the children:
   a) What information must be passed to the children during the recursive call?
   b) What information must the children pass back, and how must this information be collated?
3. Once all children have been traversed, what operations and calculations depend on information collated during the recursive traversals?
4. What information must be passed back to the parent?

Traversals

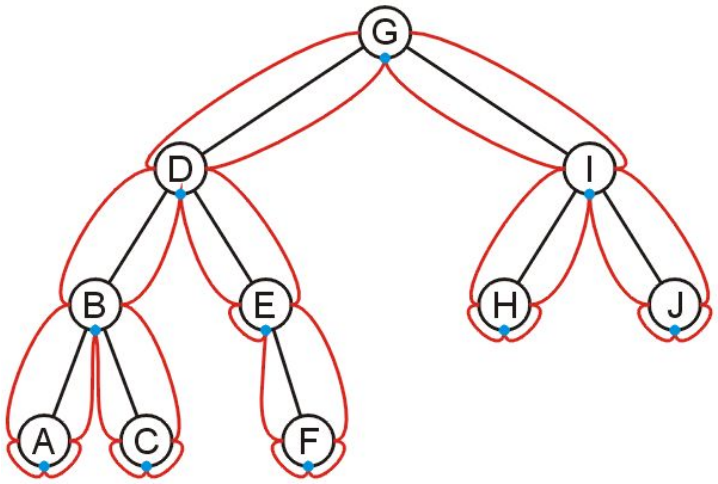
We've seen two depth-first traversals:

- Pre-order
- Post-order

In-order Traversals

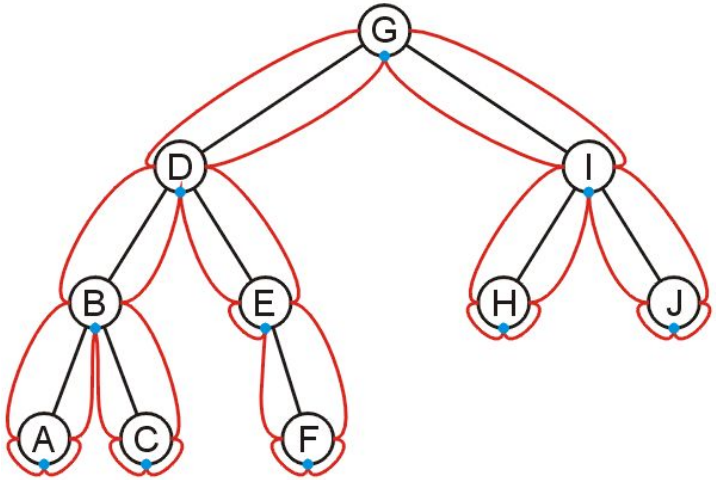
For binary trees, there is a third intermediate visit

- An *in-order* depth-first traversal



In-order Traversals

This visits a binary search tree in order



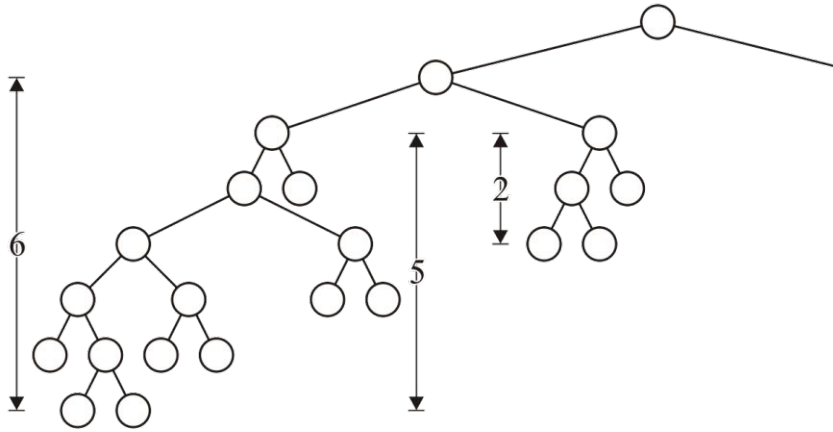
A, B, C, D, E, F, G, H, I, J

Applications of depth-first

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:

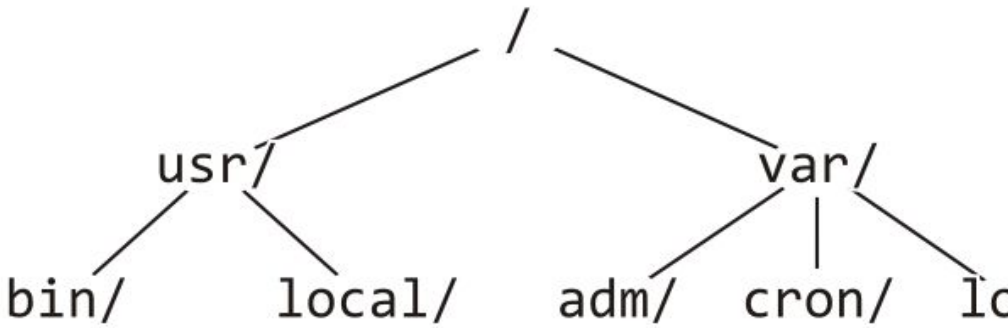
1. Before the children are traversed, we assume that the node has no children and we set the height to zero: $h_{\text{current}} = 0$
2. In recursively traversing the children, each child returns its height h and we update the height if $1 + h > h_{\text{current}}$
3. Once all children have been traversed,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

```
/
    usr/
        bin
        loc
    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 must be passed back
3. Once all children have been traversed, we are finished

Printing a Hierarchy

Assume the function `void print_tabs( int n )` prints n tabs

```
template <typename Type>
void Simple_tree<Type>::print( int depth ) const {
    print_tabs( depth );
    std::cout << value()->directory_name() << '/' << std::endl;

    for ( auto *child = children.head(); child != children.end();
          child = ptr->next() ) {
       child->value()->print( depth + 1 );
    }
}
```

Summary

This topic covered two types of traversals:

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