COMP251: DATA STRUCTURES & ALGORITHMS

Instructor: Maryam Siahbani

Computer Information System
University of Fraser Valley

* Some slides from “Algorithms and Data Structures”
by Douglas Wilhelm Harder

Abstract Trees

Outline

This topic discusses the concept of an abstract tree:

- Description of an Abstract Tree
- Applications
- Implementation

Outline

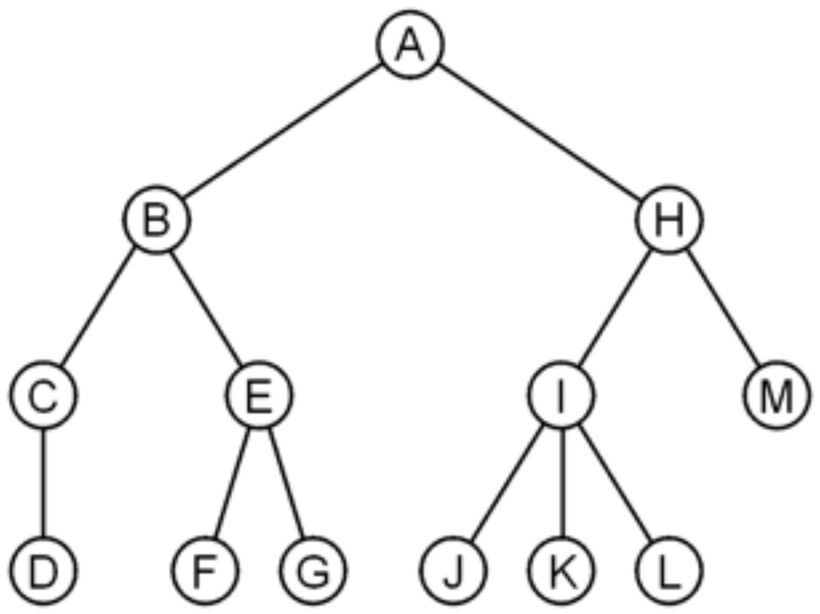
- A hierarchical ordering of a finite number of objects may be stored in a tree data structure
- Operations on a hierarchically stored container include:
  - Accessing the root:
  - Given an object in the container:
    - Access the parent of the current object
    - Find the degree of the current object
    - Get a reference to a child,
    - Attach a new sub-tree to the current object
    - Detach this tree from its parent

General Trees

An abstract tree does not restrict the number of nodes

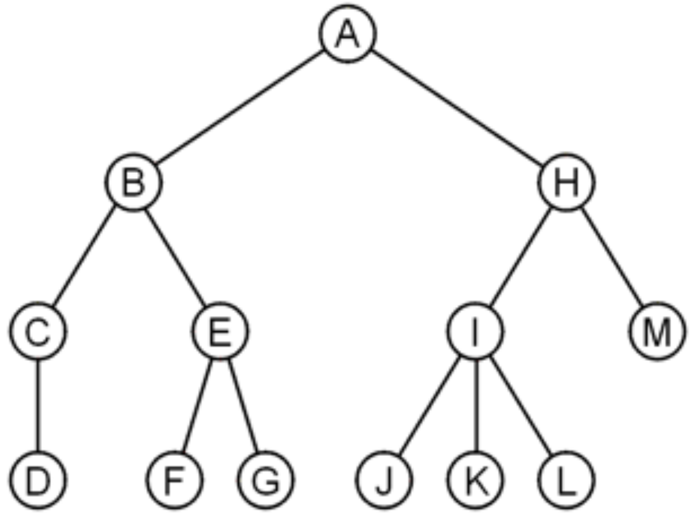
- In this tree, the degrees vary:

Degree	**Nodes**
0	D, F, G, J, K, L, M
1	C
2	B, E, H, A
3	I



General Trees: Design

- Each node in a tree can have an arbitrary number of children (and that number is not known in advance)
- Many possible implementation, we discuss one approach here:
- Implement a general tree by using a class which stores:
  - Data of an element
  - List of children (in a linked-list)
  - A reference to parent (optional)



Implementation

The class definition would be:

```
public static class treeNode<AnyType> {
    private AnyType data;
    private treeNode<AnyType> parent;
    private List<treeNode<AnyType>> children;
}

public class Tree<AnyType> {

    private treeNode<AnyType> root;

    public Tree(AnyType rootData) {
        root = new treeNode<AnyType>();
        root.data = rootData;
        root.children = new LinkedList<treeNode<AnyType>>();
    }
}
```

Applications

Trees have many applications:

- Representing family genealogies
- To model file systems.
- To represent priority queues (a special kind of tree called a heap)
- To provide fast access to information in a database (a special kind of tree called a b-tree)

Tree Traversals

It is often useful to iterate through the nodes in a tree:

- To print all values
- To determine if there is a node with some property
- To make a copy

Tree Traversals

- When we iterate through a List
  - We start with the first node and visited each node in turn. Since each node is visited, the best possible complexity is O(N).
  - For a tree with N nodes, traversal methods should be O(N).

Tree Traversals

- There are many different orders to visit the nodes in trees.
- Three common traversal orders for general trees (one more for binary trees):
  - preorder
  - postorder
  - level order
  - in-order (binary trees)

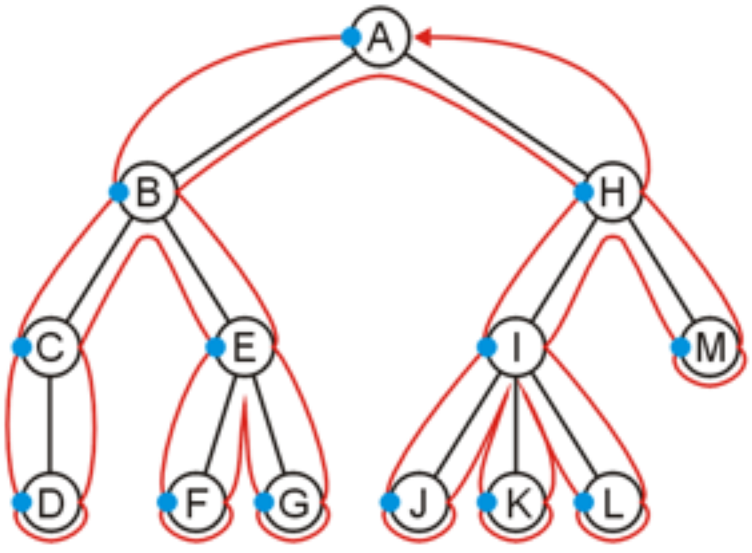
Preorder

- Definition: Read the parent before its children
- A recursive algorithm for preorder traversal:
    - Visit the root
    - Perform a preorder traversal of the first subtree of the root
    - Perform a preorder traversal of the second subtree of the root
    - etc. for all the subtrees of the root

Preorder

Visiting each node first (before its children) results in the sequence

A, B, C, D, E, F, G, H, I, J, K, L, M



Preorder

Example: print preorder of the tree nodes.
We define a method printPreorder in class Tree:

```
public void printPreorder(){
    if(root != null) {
        root.printPreOrder();
}
```

Recursive helper method which is defined in class treeNode

```
public void printPreorder(){
        //Visit the node by Printing the node data
        System.out.print(this.data+" ");
        for (treeNode<AnyType> child : this.children)
            child.printPreorder();
    }
```

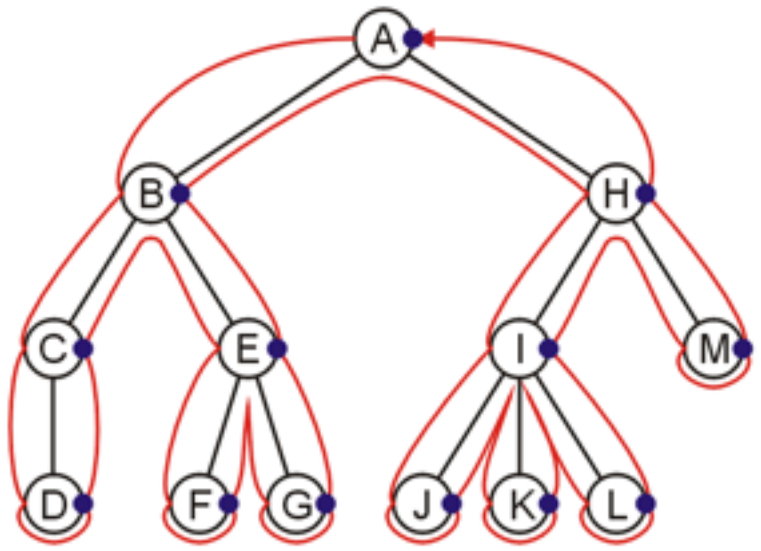
Postorder

- Definition: Read the parent after all children
- A recursive algorithm for postorder traversal:
  - Perform a postorder traversal of the first subtree of the root
  - Perform a postorder traversal of the second subtree of the root
  - etc. for all the subtrees of the root
  - Visit the root

Postorder

Visiting the nodes with their last visit:

D, C, F, G, E, B, J, K, L, I, M, H, A



Postorder

Example: print postorder of the tree nodes.
We define a method printPostorder in class Tree:

```
public void printPostorder(){
   if(root != null) {
       root.printPostorder();
}
```

Recursive helper method which is defined in class treeNode

```
public void printPostorder(){
      for (treeNode<AnyType> child : this.children)
           child.printPostorder();
      //Visit the node by Printing the node data
       System.out.print(this.data+" ");
}
```

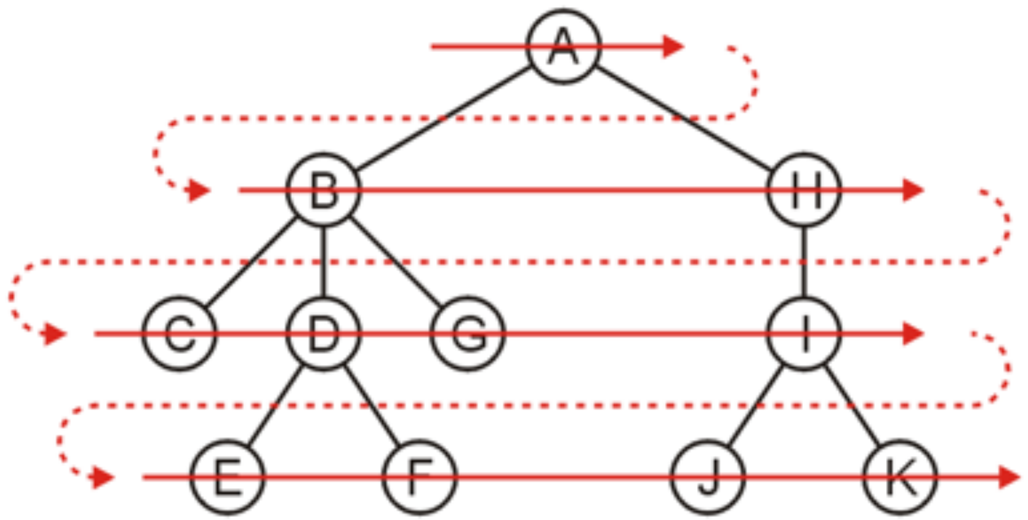
Level Order

- Definition: Visit all nodes of a given level
    - Visit the root
    - Then visit all nodes at level 1(left to right),
    - Then visit all nodes at level 2
    - etc.

Level Order

Visiting the nodes level by level (depth 1 to…)

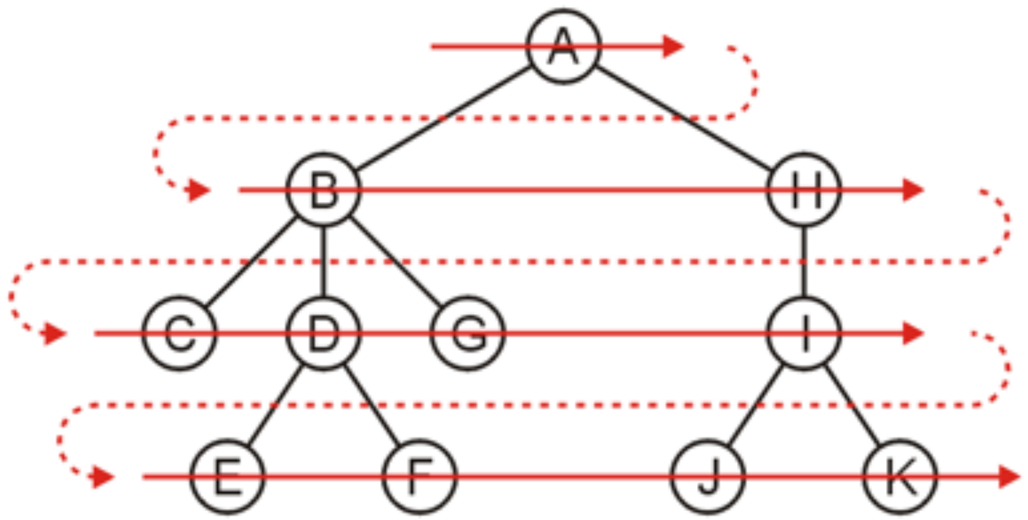
A B H C D G I E F J K



Level Order

Visiting the nodes level by level (depth 1 to…)

A B H C D G I E F J K



Level Order

The easiest implementation is to use a queue:

- Visit root node (e.g. print the data)
- Place the root into a queue
- While the queue is not empty:
  - Dequeue the node at the front of the queue
  - Visit the node (e.g. print the data)
  - Enqueue all of its children into the queue

This is also called breadth-first order (We will see breadth-first-search in the graphs again)

Analysis

Time complexity

- Preorder and Postorder: $\Theta(n)$
  - n is the number of nodes
  - visit each node once!
- Try to implement them iteratively (using a stack), time complexity is the same:
  - 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
- Level Order : $\Theta(n)$
  - visit each node once!

Analysis

Space complexity

- Iterative algorithm for Preorder and Postorder: $\Theta(hd)$
  - h is the height of the tree,
  - d is maximum degree
  - In the iterative algorithm, the nodes on the stack are all unvisited siblings from the root to the current node
  - If each node has a fixed number of 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

- We need a stack frame for each method call

Guidelines

Postorder traversal is used whenever:

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

Preorder traversal is used whenever:

- –The **children** require information about all its parent or **ancestors**

Applications

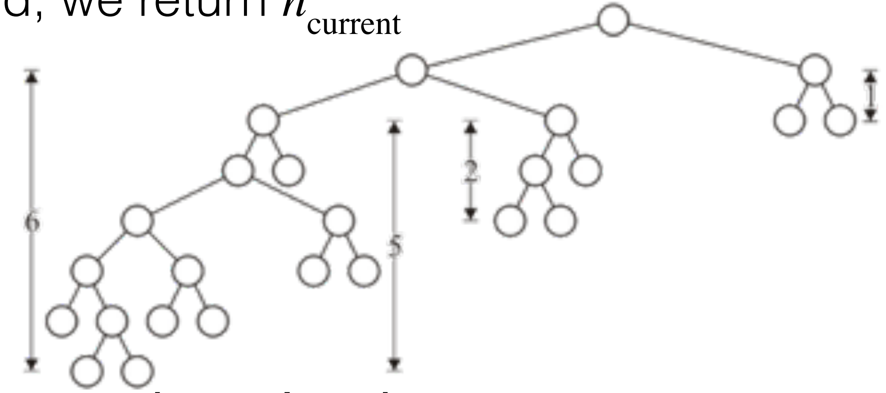
Tree application: displaying information about directory structures and the files contained within

- Finding the height of a tree
- Printing a hierarchical structure
- Determining memory usage

Height

We define a function **`int height()`** 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

Height

Sample Implementation:
We define a method height in class Tree, which calls a recursive height() method on root.

```
public int height() {
    return root.height();
}
```

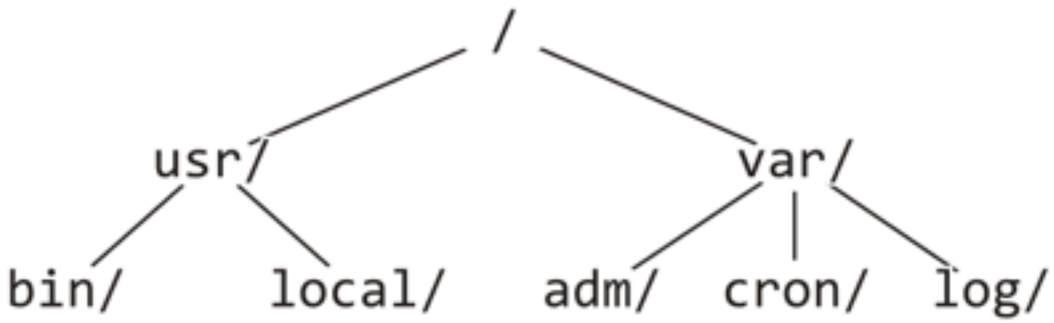
The helper method which is defined in class treeNode.

```
    public int height(){

        int h = 0;
        for (treeNode<AnyType> child : this.children) {
            h = max(h, 1 + child.height());
        }
        return h;
    }
```

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 (to maintain the appropriate number of tabs for indentation)
3. Once all children have been traversed, we are finished

Printing a Hierarchy

Sample Implementation: (Assume the function `void print_tabs(int n)` prints *n* tabs)

- We set depth to 0 when we call the method on the root of the tree.

```
void print(int depth) {
    print_tabs( depth );
    System.out.println(this.name() + "/");

    for (Dir child : this.children){
        child.print(depth + 1 );
    }
}
```

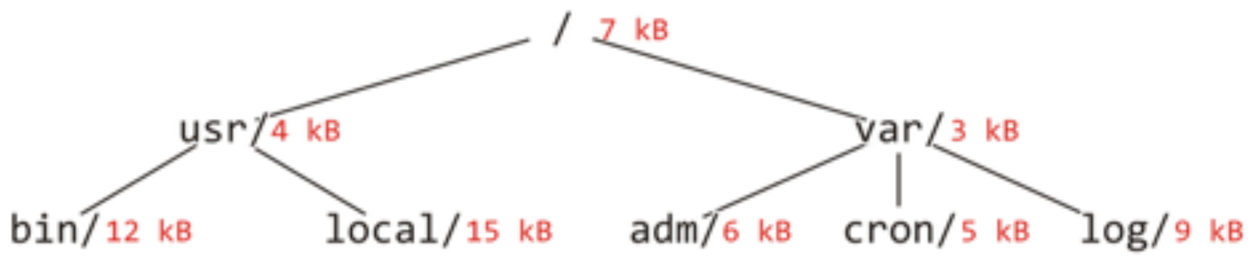
Determining Memory Usage

Suppose we need to determine the memory usage of a directory and all its subdirectories:

- We must determine and print the memory usage of all subdirectories before we can determine the memory usage of the current directory

Determining Memory Usage

Suppose we are printing the directory usage of this tree:



output:

```
        bin/ 12
        local/ 15
    usr/ 31
        adm/ 6
        cron/ 5
        log/ 9
    var/ 23
/ 61
```

Determining Memory Usage

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) Initialize the memory usage to that in the current directory.
2. In recursively traversing the children:
    a) A value of one plus the current tab level must be passed to the children, and
    b) Each child will return the memory used within its directories and this must be added to the current memory usage.
3. Once all children have been traversed, we must:
    a) Print the appropriate number of tabs,
    b) Print the name of the directory followed by a `"/   "`, and
    c) Print the memory used by this directory and its descendants

Determining Memory Usage

Sample Implementation (assume memory() and name() has been defined to compute memory and return name)

```
int du(int depth) {
    int usage = this.memory();
    //memory(): a predefined function to compute the memory

    for (Dir child : this.children) {
            usage +=  child.du( depth + 1 );
    }

    print_tabs( depth );
    System.out.println(this.name() + "/ " + usage);

     return usage;
}
```

Extra Slides

Guidelines

In designing a (pre or post order) 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?