

# Trees

Lecturer: Arran Stewart

# Outline

We discuss:

- ▶ What is a tree?
- ▶ What are they used for?
- ▶ How can they be implemented?

# Trees

- ▶ *Trees* are a data structure that occurs in many real-world situations.
- ▶ We have ...

---

genealogical trees	organisational trees
biological hierarchy trees	evolutionary trees
population trees	book classification trees
probability trees	decision trees
induction trees	design trees
graph spanning trees	search trees
planning trees	encoding trees
compression trees	program dependency trees
expression/syntax trees	fruit trees
⋮	⋮

---

- ▶ Additionally, many other data structures are based on trees

## Example – a family tree

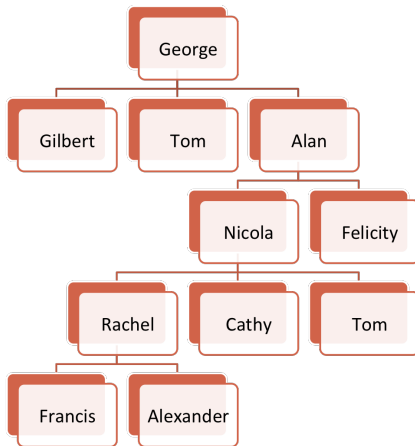
- ▶ How could you represent your family members in a computer?

## Example – a family tree

- ▶ How could you represent your family members in a computer?
- ▶ We could possibly use a list – but that doesn't represent very well the *hierarchy* in a tree.

## Example – a family tree

If we were drawing a family tree, we might use a representation like this:



- Note that not all family information is shown in this tree. For example, we only show one of two parents.

## Example – a family tree

How is this better than a list representation?



## Example – a family tree

How is this better than a list representation?



- ▶ One reason is that the hierarchical structure contains more information than a list
- ▶ We know family relationships between everyone by examining the tree.
- ▶ Additionally – though it's not obvious in this case – storing data in trees can make it much faster to search the data.



# Tree terminology

- ▶ Each person is represented by a **node** in the tree.
- ▶ Nodes can have **child** nodes as well as a **parent** node.
  - ▶ We use the terms “child” and “parent”, even when using trees for things besides families.
- ▶ A node with no parent, the topmost node, is called the **root** of the tree.
- ▶ A **leaf** node is one that does not have any children.
  - ▶ Francis, Felicity, and Gilbert are all leaf nodes.

## Tree terminology, cont'd

- ▶ The **depth** of a node is the length of the path from the root to the node.
- ▶ The **height** of a node is the length of the path from the node to the deepest leaf.
  - ▶ Thus, the height of a tree is the number of “levels”.
- ▶ Nodes with no children (leaves) are sometimes called **external nodes**;
- ▶ and nodes with one or more children are **internal nodes**.

# Binary trees

- ▶ A **binary tree** is a specific type of tree, in which each node has at most 2 children.
- ▶ Binary trees an important base for other data structures such as binary search trees and priority queues.

# Binary tree nodes

How can we implement a binary tree?

Tree nodes can be implemented in a similar way to nodes in a linked list.

Each node has some data it stores, and two pointers to other nodes.

The code below is a simplified version of the code in `BinaryTreeNode.java`.

```
public class BinaryTreeNode {  
    int value;           // data value associated with a node  
    BinaryTreeNode left; // reference for the left child  
    BinaryTreeNode right; // reference for the right child  
    // constructor and other methods go here  
    // ...  
}
```

## Binary tree nodes

- ▶ In the family tree we saw, the associated data was each person's name.
  - ▶ Names could be represented with a `String`.
- ▶ But we would not normally use a *binary* tree to represent a family tree, as people may have more than two children.

# Tree Traversal

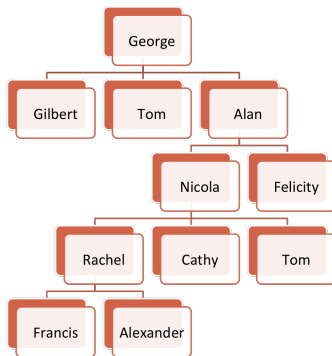
Suppose we wanted to visit all the elements of a binary tree.

- ▶ This is called **traversing** the tree.
- ▶ Tree traversal is used to search for an element, to print out all elements, and so on.

There are different ways of doing tree traversal.

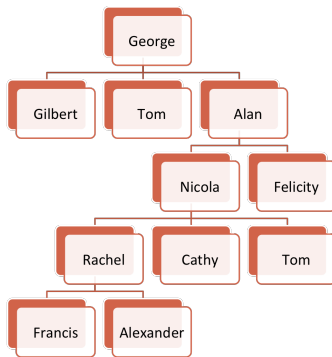
## “Pre-order” traversal (NLR)

- ▶ One possible order would be:
  - ▶ Visit a node
  - ▶ Then, for each of its child nodes, do the same.
- ▶ This would give us the order:  
George, Gilbert, Tom, Alan, Nicola, Rachel, Francis, Alexander ...



## “Pre-order” traversal (NLR)

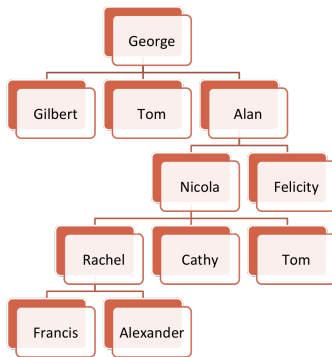
- ▶ This is called “pre-order” because we visit each node *before* we visit its children.
- ▶ A way of remembering this is to think of the mnemonic “**NLR**” – we visit the **N**ode, then its **L**eft child, then the **R**ight child.





## “Pre-order” traversal (NLR)

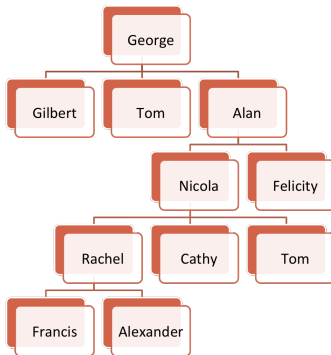
- ▶ This order can also be described as being *depth-first* – if there is a long path “down” the tree, we will travel all the way down it before moving to branches further to the right.



## “Pre-order” traversal (NLR)

- ▶ Tree traversals are most straightforwardly implemented using *recursive* methods.
- ▶ Pseudocode for doing a pre-order traversal would be:

```
method preorder(aNode):  
    visit aNode  
    preorder(aNode.left)  
    preorder(aNode.right)
```



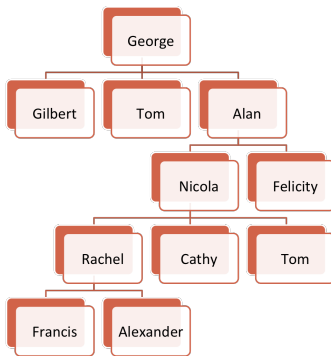
## Example of pre-order traversal

In the `BinaryTreeNode` class, you can see a method which uses pre-order traversal to print nodes:

```
public void printPreOrder () {  
    if (this != null)  
        System.out.println(value.toString()); //visit root  
    if (left != null)  
        left.printPreOrder(); //visit left  
    if (right != null)  
        right.printPreOrder(); //visit right  
}
```

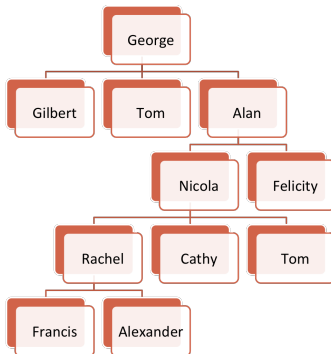
## “Post-order” traversal (LRN)

- ▶ To do *post-order* traversal, given some node, we
  - ▶ do post-order traversal of its left child
  - ▶ do post-order traversal of its right child
  - ▶ visit the node
- ▶ This would give us the order:  
Gilbert, Tom, Francis, Alexander, Rachel ...



# “Post-order” traversal (LRN)

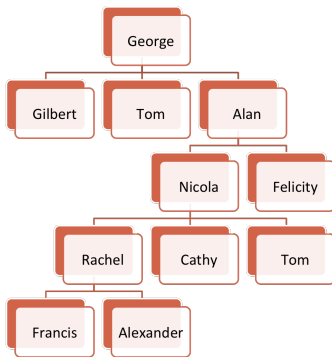
- ▶ This is called “post-order” because we visit each node *after* we visit its children.
- ▶ A way of remembering this is to think of the mnemonic “**LRN**” – we process the node’s **L**eft child, then the **R**ight child, and then we visit the **N**ode.



## “Post-order” traversal (LRN)

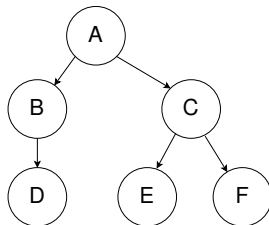
Pseudocode for post-order traversal would be:

```
method postorder(aNode):  
    postorder(aNode.left)  
    postorder(aNode.right)  
    visit aNode
```



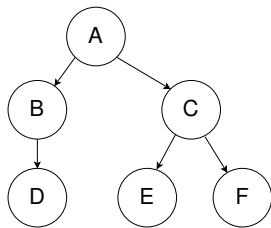
## “In-order” traversal (LNR)

- ▶ For “in-order” traversal, we
  - ▶ process the **l**eft child
  - ▶ then process the **r**ight child
  - ▶ then visit the **n**ode
- ▶ Hence the mnemonic **LRN** can be used to remember it.
- ▶ For the tree on the right, this would give the order:  
D, B, A, E, C, F



## “Level order” traversal

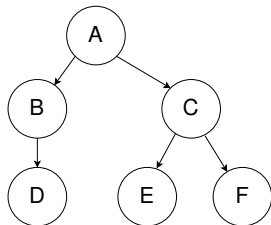
- ▶ Finally, we can do *breadth-first* or *level order* traversal.
- ▶ In this form of traversal, we visit –
  - ▶ the root
  - ▶ then all nodes at level 1 (children of the root)
  - ▶ then all nodes at level 2 (grand-children nodes)
- ▶ That is, starting at root, visit nodes level by level (left to right).
- ▶ This type of traversal does not suit a recursive approach because you have to “jump” from subtree to subtree.





## “Level order” traversal

- ▶ The order of visiting nodes if we did level order traversal would be:  
A, B, C, D, E, F



## Further reading

You should see the textbook, and the UWA lecture notes file `Topic12-Traversals.pdf` for more details on tree traversal.