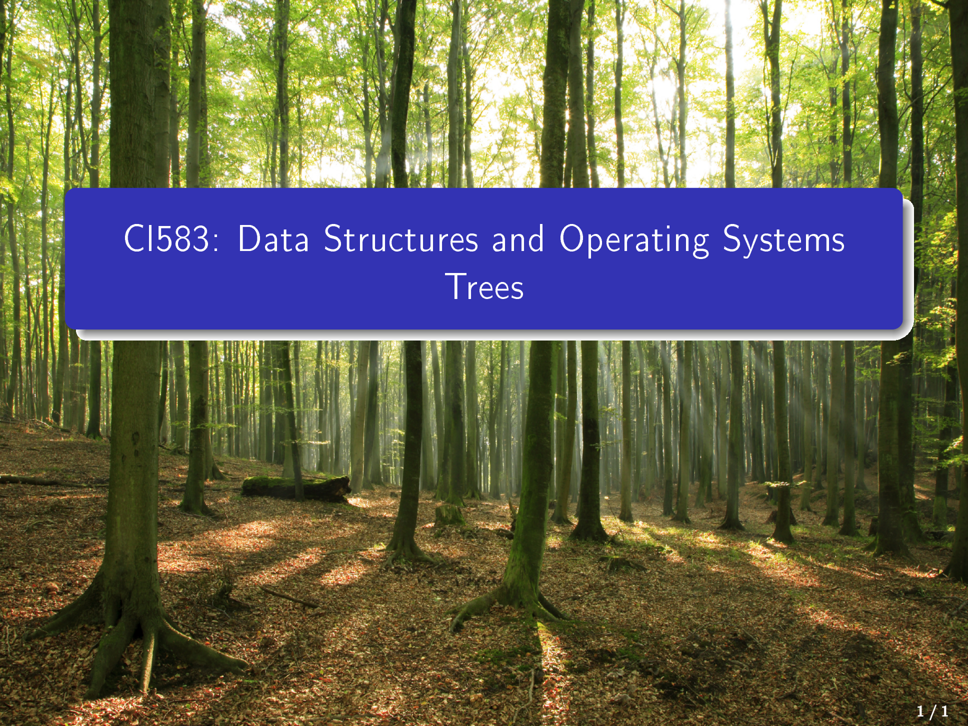


CI583: Data Structures and Operating Systems
Trees

Outline

Trees

One of the most important sort of data types is the tree. There are many varieties of tree, and they can provide very efficient ways to store and retrieve data.

It could be that the data we are storing has a naturally “tree-like” hierarchical structure, such as an organisation chart. In this case, the structure of the tree represents the *manages* relationship:

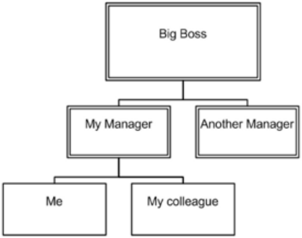
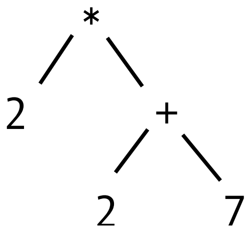


Image © http://www.drawmeanidea.com/

Trees

More generally, there is no requirement that trees contain hierarchical data or that they are arranged in an ordered way, though we will often want to do this.

Trees are a fundamental data structure that find many uses. Tree structures are widely used in compiler construction, in which expressions from a programming language or arithmetic expressions are arranged into parse trees and abstract syntax trees:

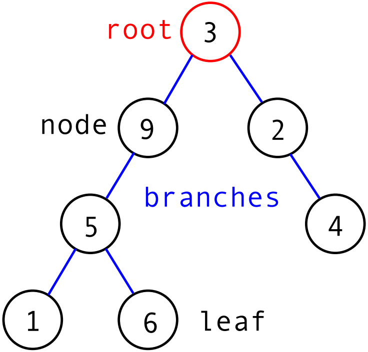


Trees

Some terminology:

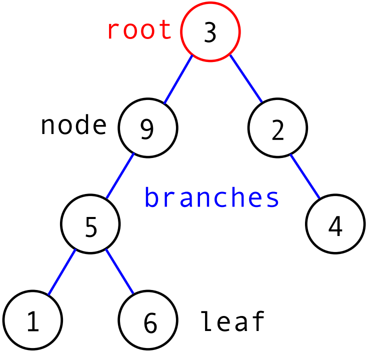
A node is a structure within the tree that may contain data (called the key or label).

Each node has zero or more child nodes and each node has one parent node except for a single node called the root, which has no parent node.



Trees

The connections between nodes are called branches, edges or links. Those nodes with no children are called leaf nodes.

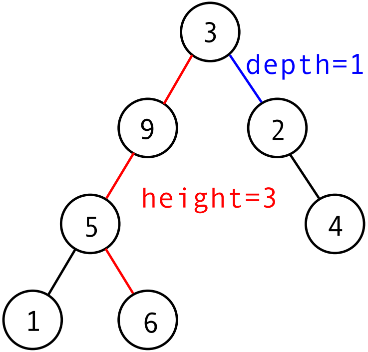


Trees

The height of a tree is the maximum number of edges from the root to a leaf. The depth of a node is the number of edges from the root to that node. Nodes with the same depth are at the same level.

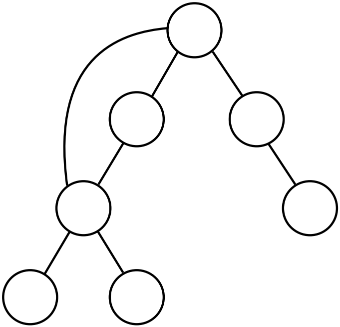
A subtree is formed by taking a node together with its children.

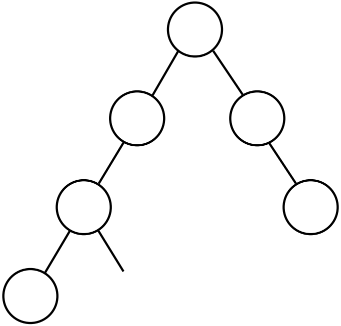
A binary tree is one in which each node has at most two children (often called the left and right child).



Two non-trees

One of these structures is not a tree because there is more than one path from the root to a leaf node (it's a graph), and the other one is just badly formed.





Programming with trees

Algorithms that operate on trees normally start with the root node then traverse the tree, either depth-first (keep following edges until we reach a leaf) or breadth-first (examine all children before going any deeper).

Programming with trees

```
abstract class Tree {
  int label;
  abstract int countNodes();
  abstract int height();
}
class BranchNode extends Tree {
  Tree left;
  Tree right;
  //...
}
class LeafNode extends Tree {
  //...
}
```

Programming with trees

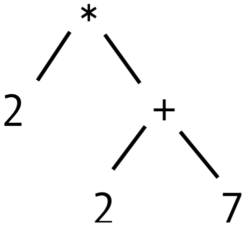
```
class BranchNode extends Tree {
  int countNodes() {
    return 1 + left.countNodes() + right.countNodes();
  }
}
class LeafNode extends Tree {
  int countNodes() {
    return 1;
  }
}
```

Programming with trees

```java
class BranchNode extends Tree {
  int height() {
    int lh = (left == null) ? 0 : left.height(); // tertiary if statement
    int rh = (right == null) ? 0 : right.height();
    return 1 + max(lh, rh);
  }
}
class LeafNode extends Tree {
  int height() {
    return 0;
  }
}
```

Traversal

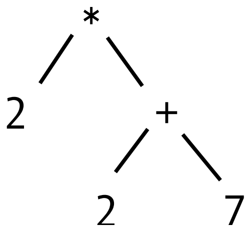
The `countNodes` method is a traversal of the tree, in which each node is visited. Visiting the node could mean adding the label to an array, printing the label, etc. To return to the parse tree example, in order to evaluate the expression that this tree represents $(2 \times (2 + 7))$, we can traverse it to collect the keys.



Traversal

There are three ways we might traverse a tree:

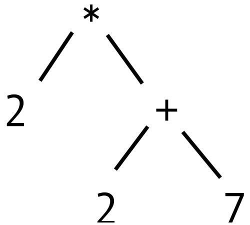
1. **inorder**: visiting the nodes in the order of their labels,
2. **preorder**: visit a node, then traverse the left sub-tree then traverse the right sub-tree, and
3. **postorder**: traverse the left and right sub-trees then visit the node.



Traversal

Thus, we can retrieve three different expression from the tree:

1. **inorder** : $2 * 2 + 7$, the *infix* expression,
2. **preorder**: $* 2 + 2 7$, the *prefix* expression,
3. **postorder**: $2 2 7 + *$, the *postfix* expression.



Traversal

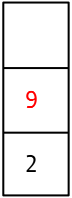
The postfix expression is unambiguous and can be evaluated conveniently using a stack. Read the expression; when we encounter an *operand*, push it onto a stack. When we encounter an *operator*, take two values from the stack, apply the operator and push the result.

2 2 7 + *

7
2
2

Traversal

2 2 7 + *



Traversal

2 2 7 + *