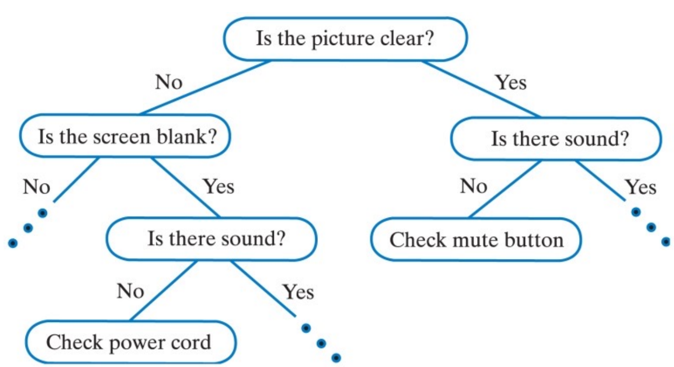
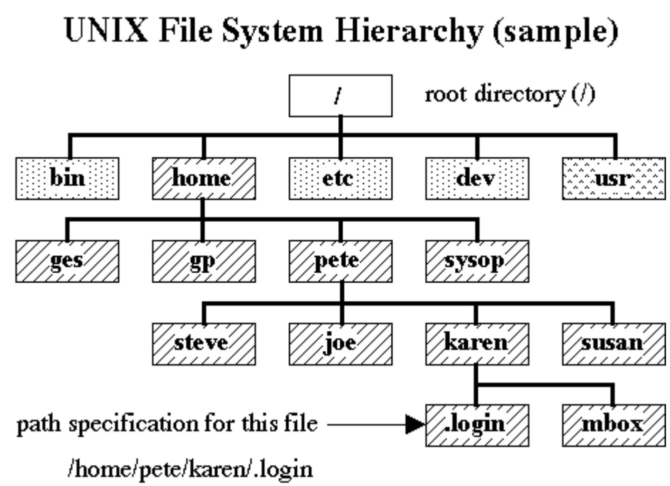
Binary Trees

Trees
Binary Trees
Binary Tree Nodes
Recursive tree algorithms

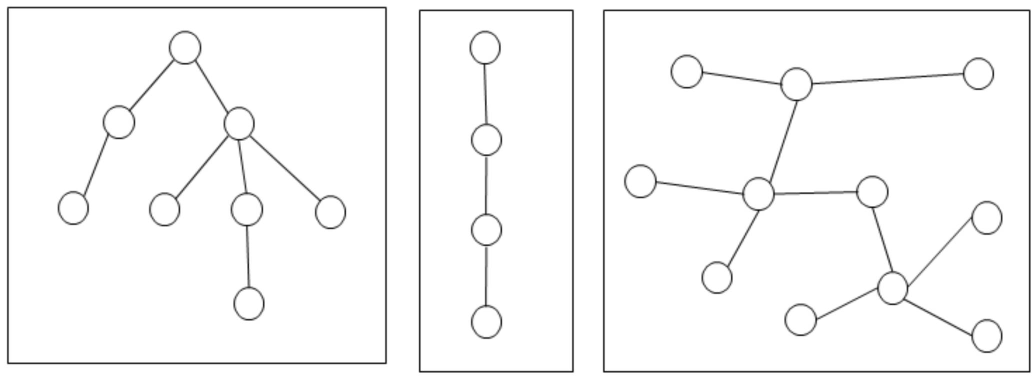




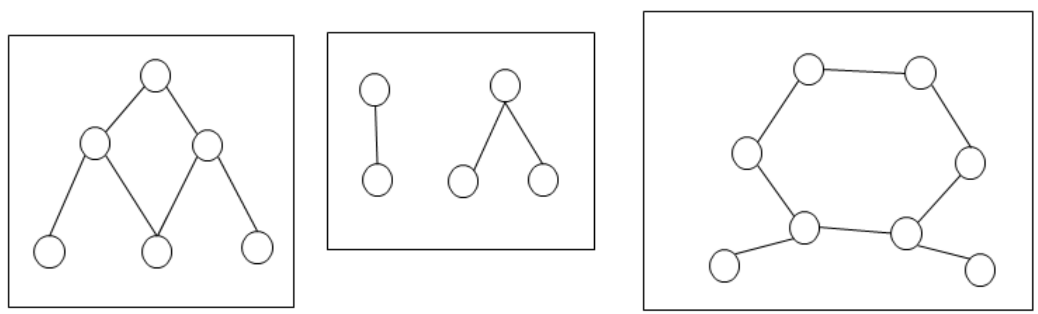
Trees

A tree is a connected undirected simple graph with no cycles

trees:



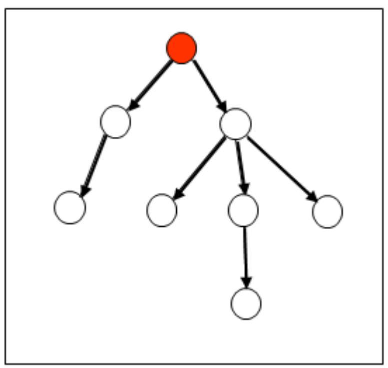
not trees:

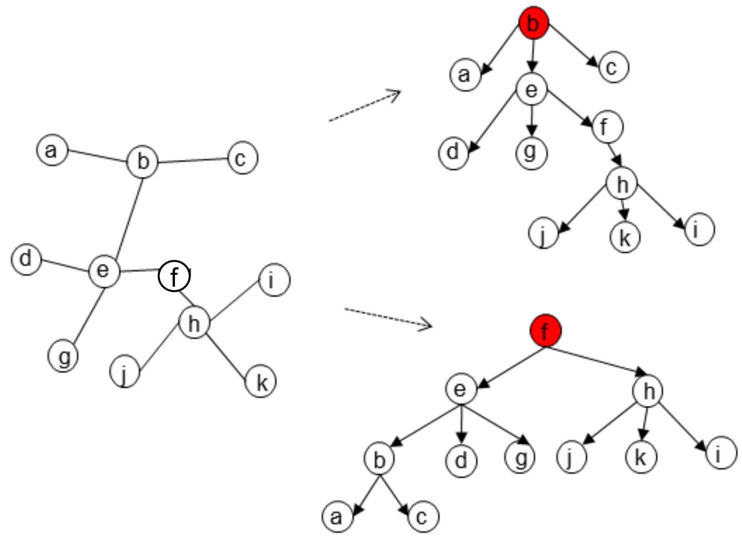


Rooted Trees

Usually, when we think of trees, we assume there is a root.

A rooted tree is a tree in which one of the vertices is designated the root, and all edges are then directed away from that root.





Describing vertices in rooted trees

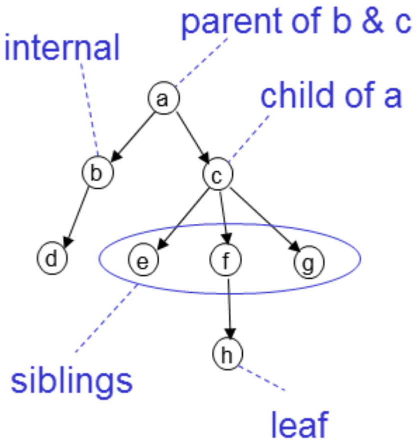
If there is a directed edge from *x* to *y*, then *x* is the parent of *y*, and *y* is a child of *x*.

If two vertices *y* and *w* have the same parent, then they are siblings of each other.

A vertex with no children is a leaf. A vertex with children is an internal vertex.

The ancestors of a vertex *v* are all the vertices in the path from *v* to the root (except for *v* itself).

The descendants of a vertex *v* are all the vertices that have *v* as an ancestor.

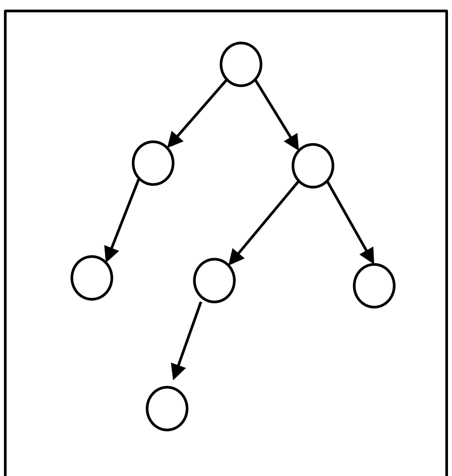


Binary Trees

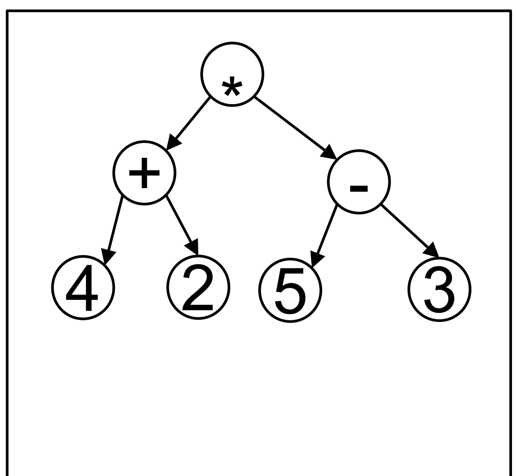
A *binary tree* is a rooted tree in which:

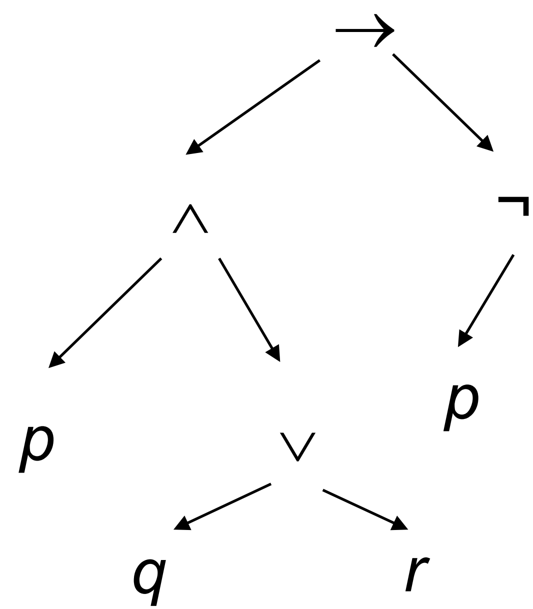
- every node has at most 2 children
- the children of a node are identified as left child and right child

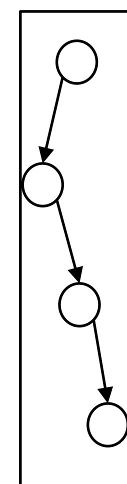
The depth of a tree is the length of the longest path from the root node to a leaf node



depth == 3



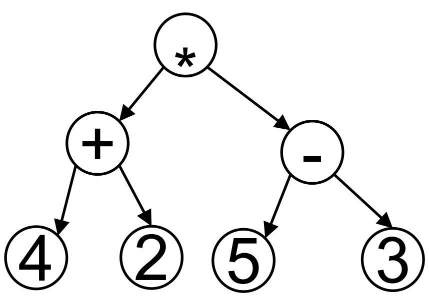


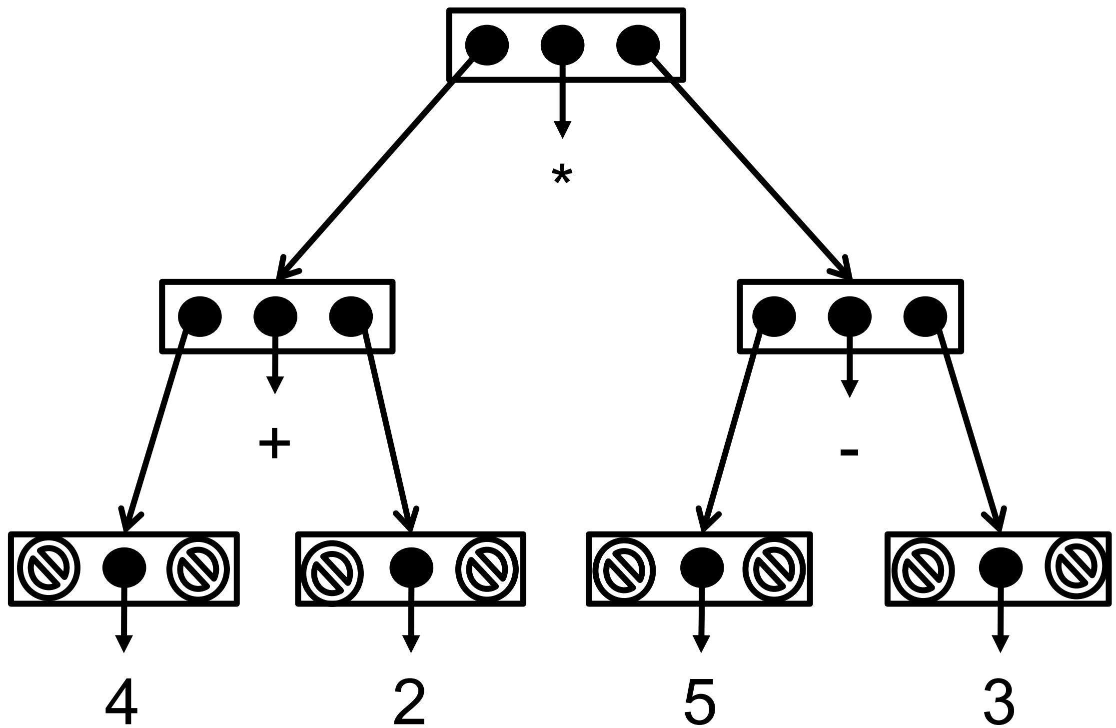


BinaryTreeNode

BinaryTreeNode

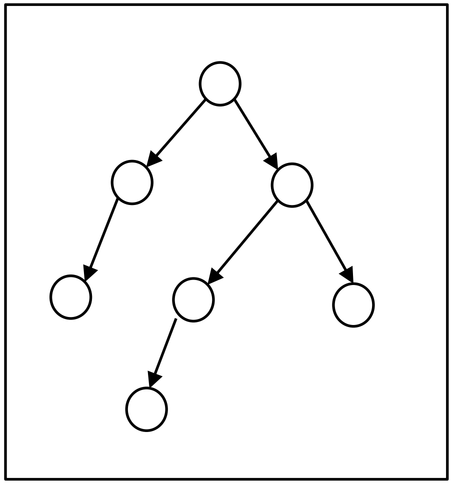
element
leftchild
rightchild





Computing the height of a node

The height of a node is the length of its longest (directed) path to a leaf.



<u>Recursive definition:</u>
height(node) = 0 if node is a leaf
height(node) = 1 + max(height(left), height(right))

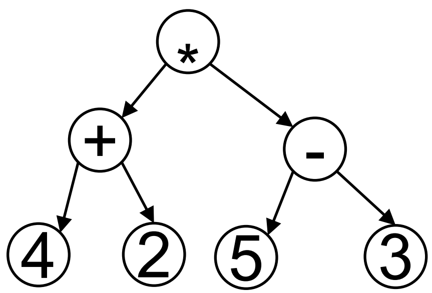
```
def height(node):
    if node == None:
        return 0
    elif node.leftchild == None and node.rightchild == None:
        return 0
    else:
        return 1 + max(height(node.leftchild),
                       height(node.rightchild))
```

recursive

Preorder traversal

to visit a node:
  read the element first, then visit the children in left-to-right order

'preorder' because we do the parent's element before the children



*
+
4
2
–
5
3

```
def preorder_print(node):
    if node:
        print(node.element)
        preorder_print(node.leftchild)
        preorder_print(node.rightchild)
```

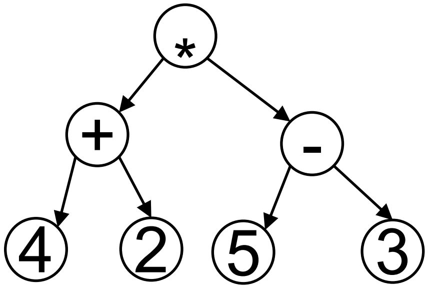
recursive

```
def preorder_str(node):
    if node:
        outstr = str(node.element)
        outstr = outstr + preorder_str(node.leftchild)
        outstr = outstr + preorder_str(node.rightchild)
        return outstr
    else:
        return ''
```

* + 4 2 – 5 3

Inorder traversal

to visit a node:
visit the leftchild, then the parent's element,
then the right child



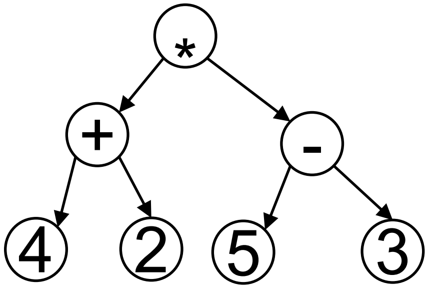
((4 + 2) * (5 - 3))

```
def inorder_str(node):
    if node:
        if node.leftchild or node.rightchild:
            outstr = '(' + inorder_str(node.leftchild)
            outstr += node.element
            outstr += inorder_str(node.rightchild) + ')'
            return outstr
        else:
            return str(node.element)
    else:
        return ''
```

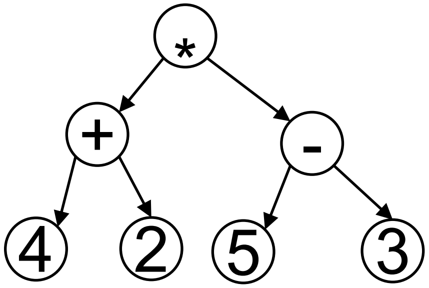
Post-order traversal

Exercise: what would 'post-order' traversal mean?

Write Python code to produce a post-order traversal print of the tree on the right.



Evaluating Expression Trees



evaluate node:
  if node element is a number, return it
  else
    evaluate left child
    evaluate right child
    determine the operator in the node
    apply leftvalue operator rightvalue

Exercise: implement this in Python

Exercise

How would you implement the preorder traversal *without* using explicit recursion?

Next Lecture

Binary Search Trees