Announcements

- Quiz 1 deadline updated to Friday 13 March, 8pm.

- Quiz 2 release after the lecture, due before next lecture.

- Assignment 1 deadline still Thursday 12 March midnight.

- Assignment 2 (programming) release on Friday:
  - Public, private, hidden.

Data structures and Algorithms

Lecture 3: Trees
[GT 2.3]

Dr. André van Renssen
School of Computer Science

Some content is taken from material provided by the textbook publisher Wiley.

THE UNIVERSITY OF SYDNEY

Agenda: Trees

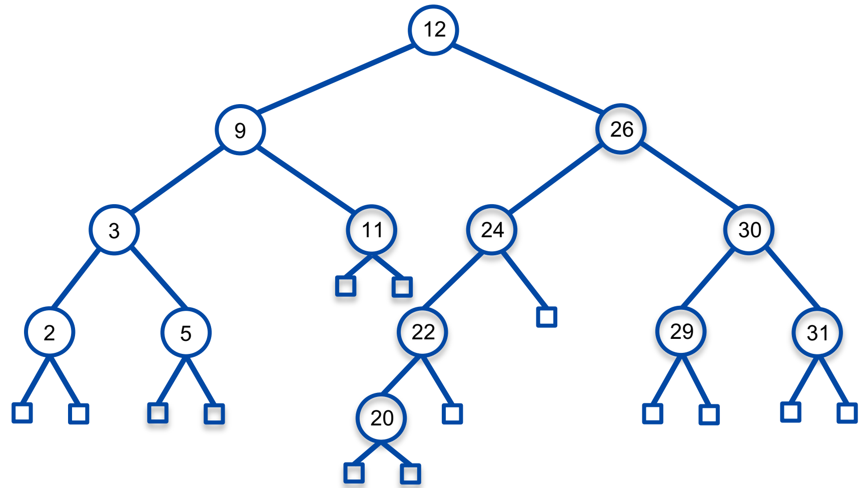
- Definition and terminology
- Applications
- Tree ADT
- Tree traversal algorithms
- Binary trees
- Implementing trees
- Recursive code on trees

Trees

leaves

root

root



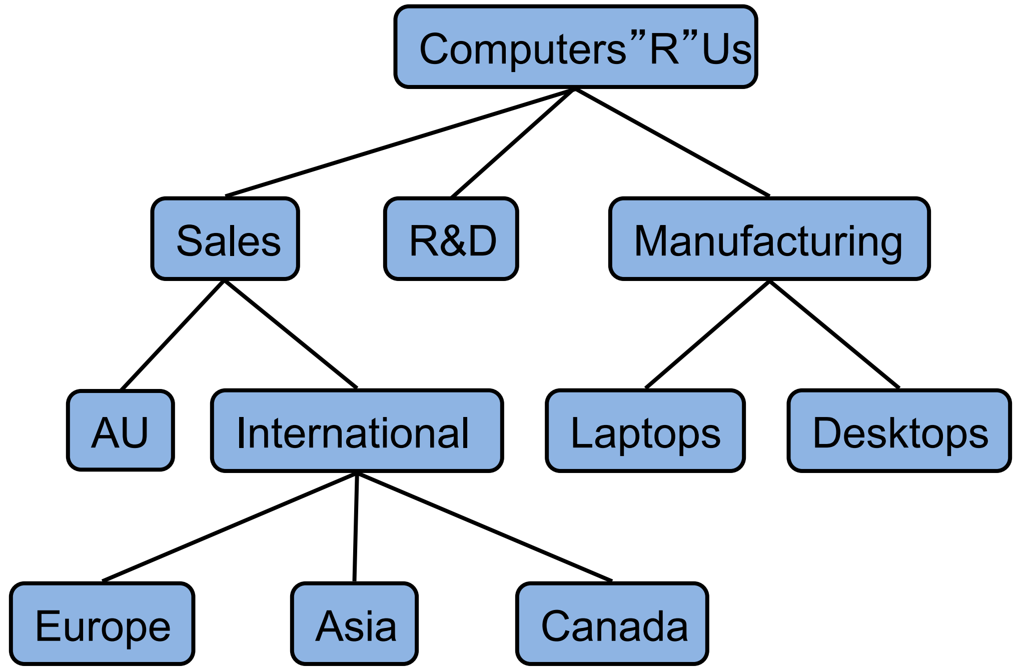
leaves

What is a Tree

In computer science, a tree is an abstract model of a hierarchical structure

A tree consists of nodes with a parent-child relation

- if u is parent of v, then v is a child of u
- a node has at most **one** parent in a tree
- a node can have zero, one or more children



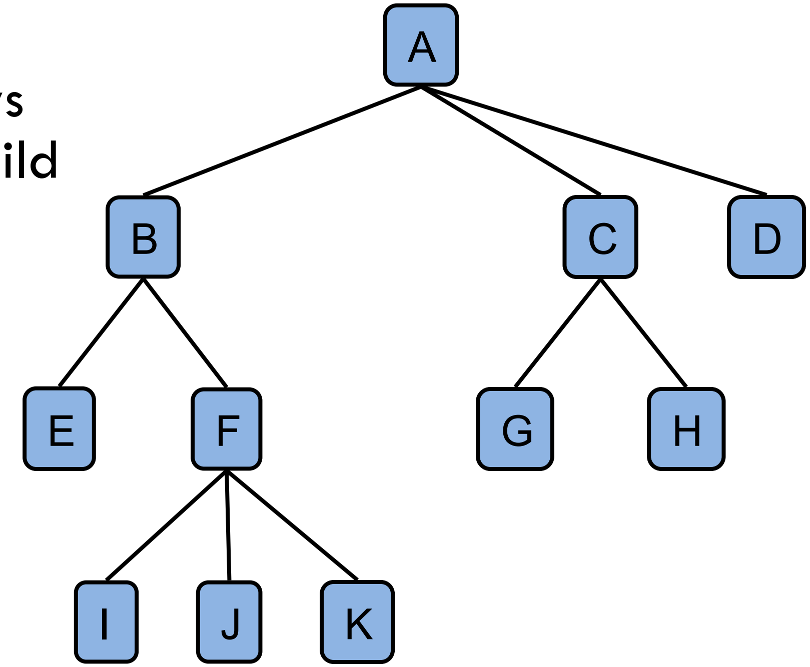
Applications:

- Organization charts
- File systems
- Phrase structure

Formal definition

A tree T is made up of a set of nodes endowed with parent-child relationship with following properties:

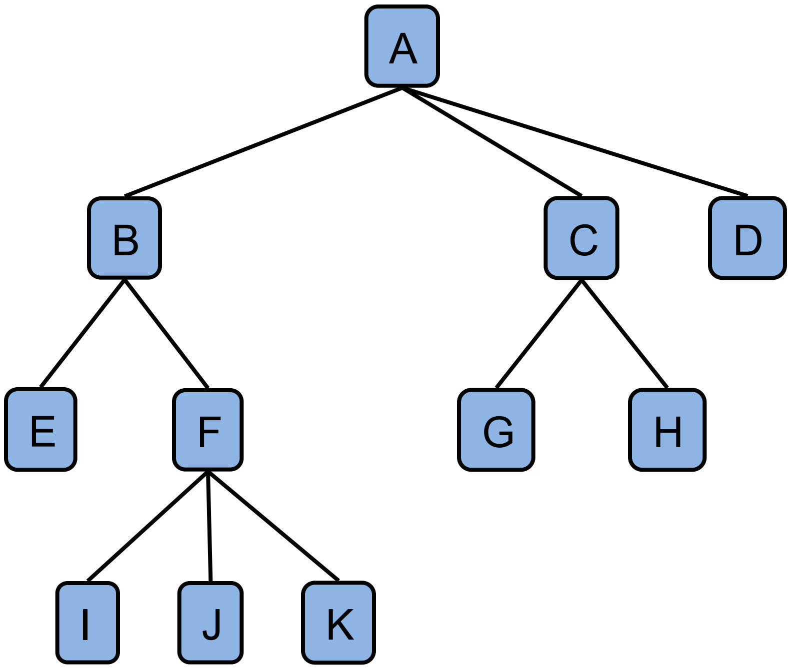
- If T is non-empty, it has a special node called the root that has no parent
- Every node v of T other than the root has a unique parent
- Following the parent relation always leads to the root (i.e., the parent-child relation does not have "cycles")



Tree Terminology

Depending on where they are in the tree, we classify nodes into:

- **Root:** node without parent (e.g., A)
- **Internal node:** node with at least one child (e.g., A, B, C, F)
- **External/leaf node**: node without children (e.g., E, I, J, K, G, H, D)



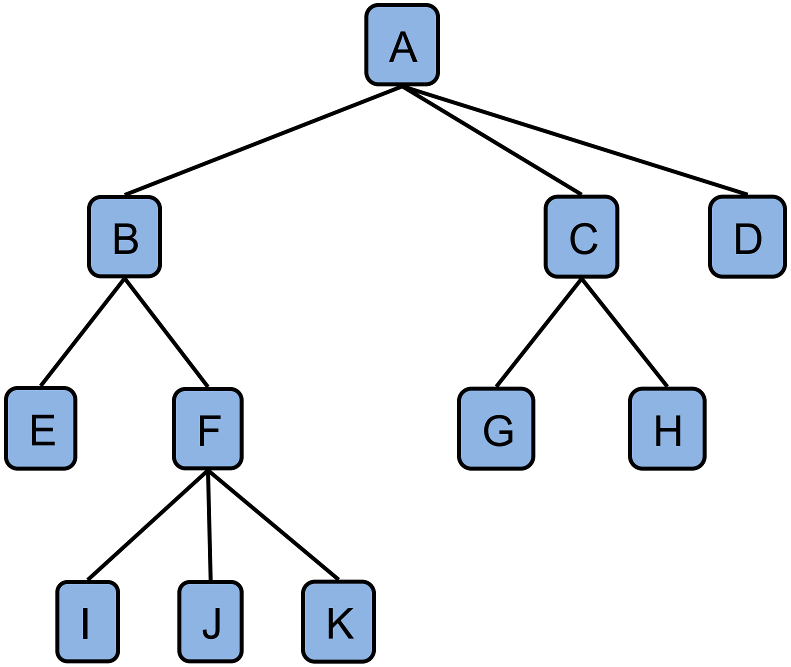
Tree Terminology

We can extend the parent-child relation to capture indirect relations:

- Ancestors: parent, grandparent, great-grandparent, etc. (e.g., ancestors of F are A, B)

- Descendants: child, grandchild, great-grandchild, etc. (e.g., descendants of B are E, F, I, J, K)

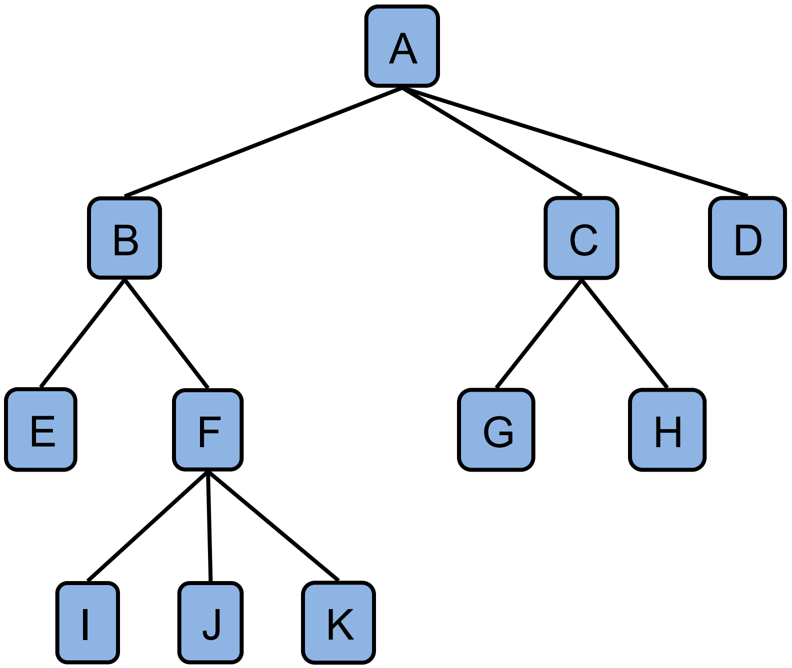
- Two nodes with the same parent are siblings (e.g., B and D)



Tree Terminology

More fine-grained location concepts:

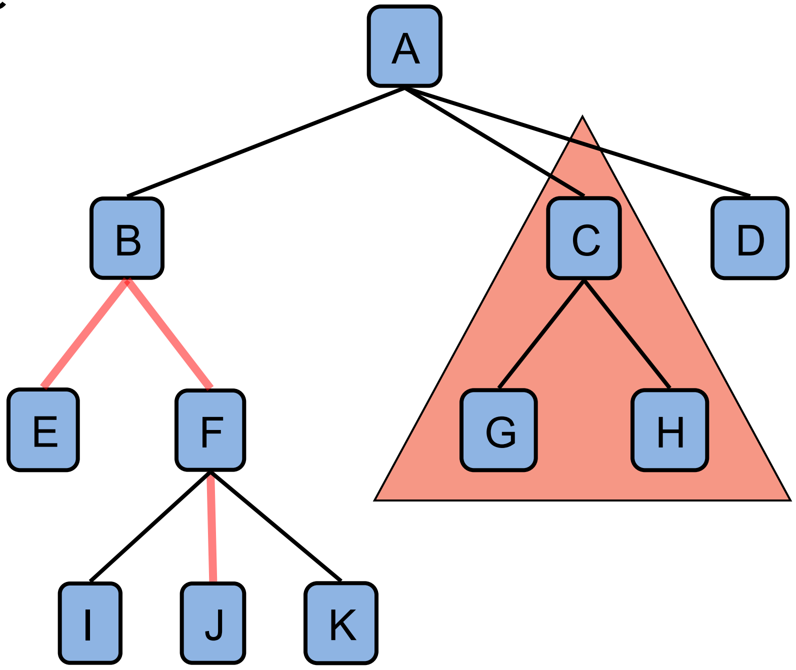
- **Depth of a node:** number of ancestors not including itself (e.g., depth(F) = 2)
- **Level:** set of nodes with given depth (e.g., {E, F, G, H} are level 2)
- **Height of a tree:** maximum depth (e.g., 3)



Tree Terminology

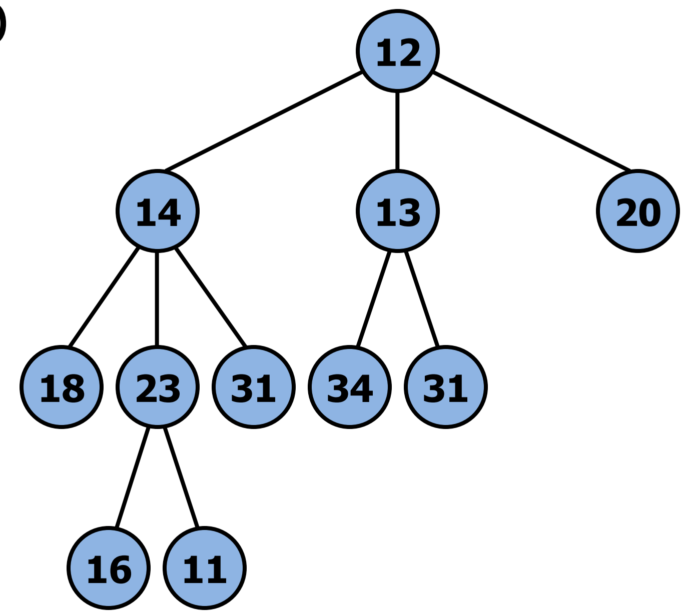
Substructures of a tree:

- **Subtree:** tree made up of some node and its descendants. (e.g., subtree rooted at C is {C, G, H})
- **Edge:** pair of nodes (u, v) such that one is the parent of the other
- **Path:** sequence of nodes such that 2 consecutive nodes in the sequence have an edge (e.g., <E, B, F, J>).



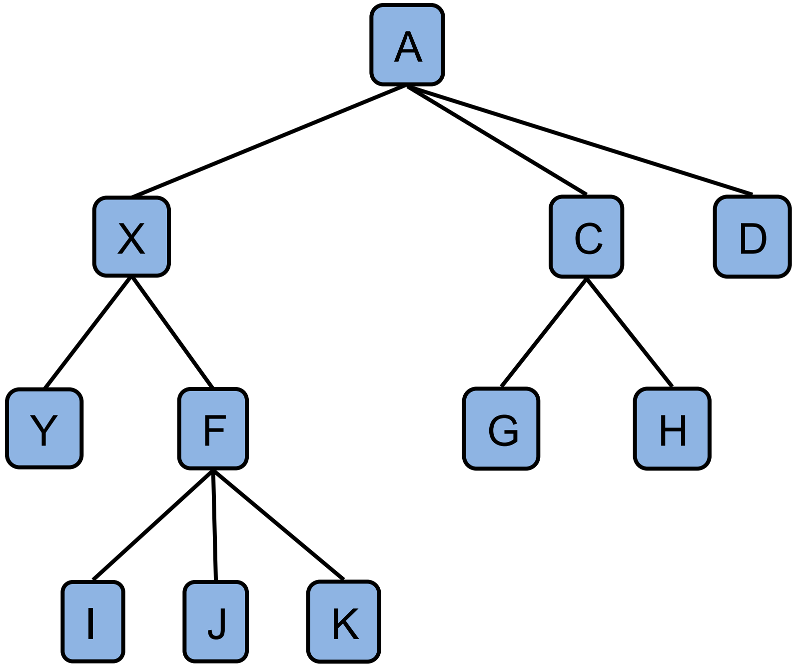
Examples

- Node14 has depth … 1
- The tree has height … 3
- Subtree rooted at node 14 has height … 2
- Any subtree of a leaf has height … 0
- The root has depth … 0



Tree facts

- If node X is an ancestor of node Y, then Y is a descendant of X.
- Ancestor/descendant relations are transitive
- Every node is a descendant of the root
- There may be nodes where neither is an ancestor of the other
- Every pair of nodes has at least one common ancestor.
- The lowest common ancestor (LCA) of x and y is a node z such that z is the ancestor of x and y and no descendant of z has that property



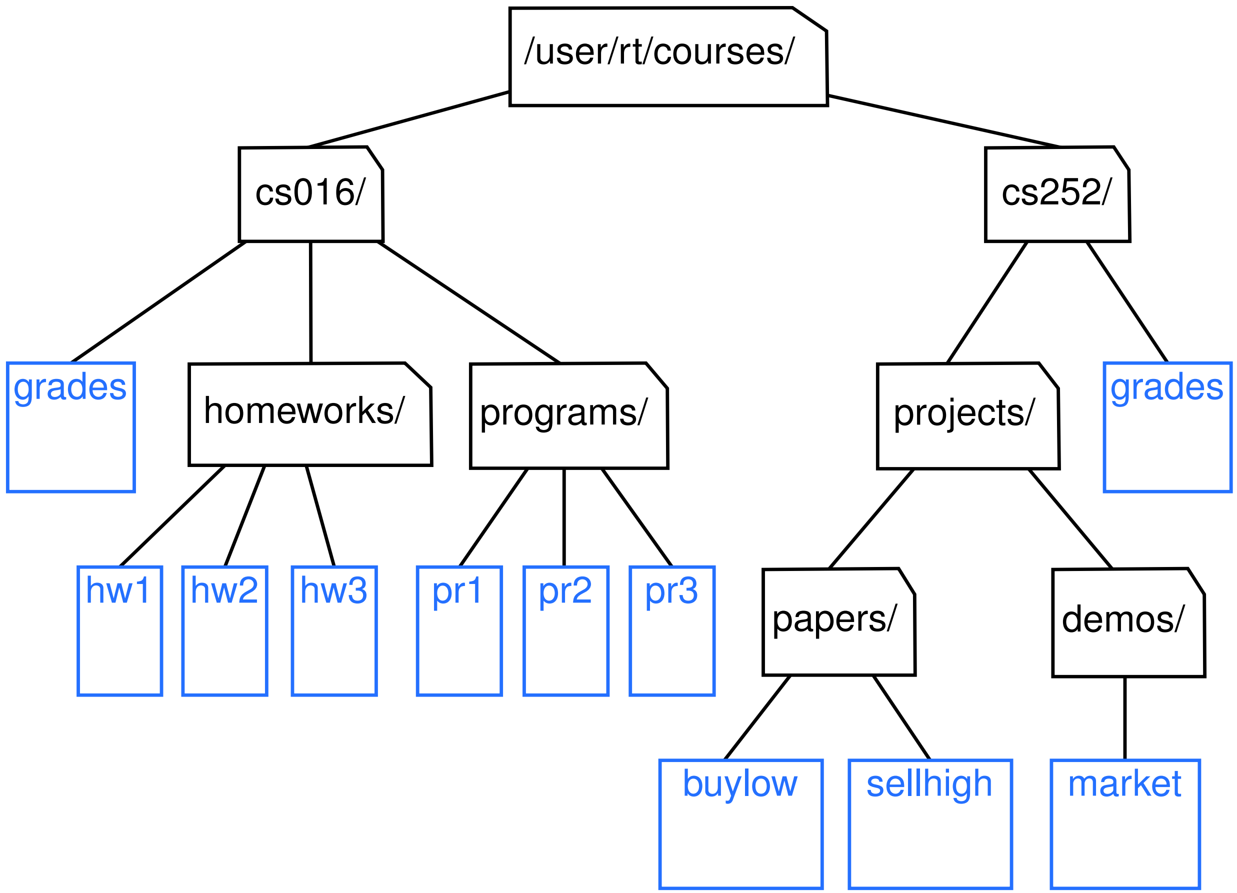
Ordered Trees

Sometimes order of siblings matter

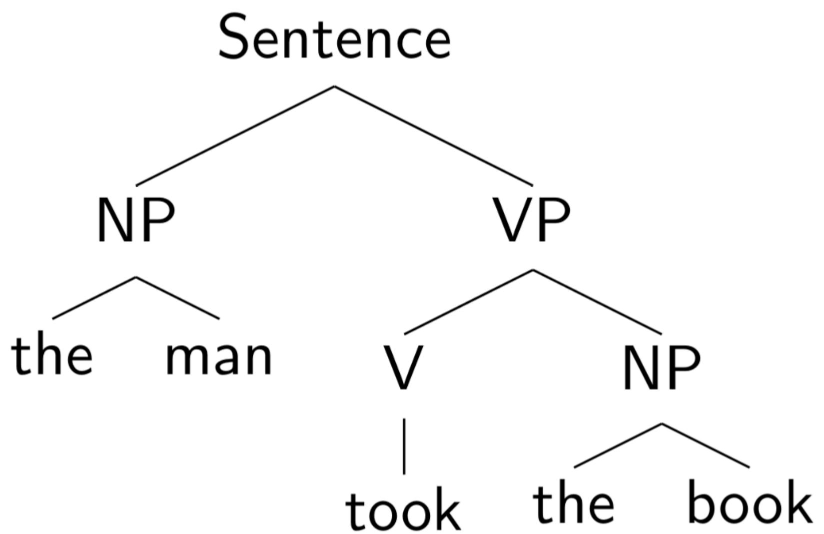
In an ordered tree there is a prescribed order for each node's children

In a diagram this ordering is usually represented by the left to right arrangement of the nodes

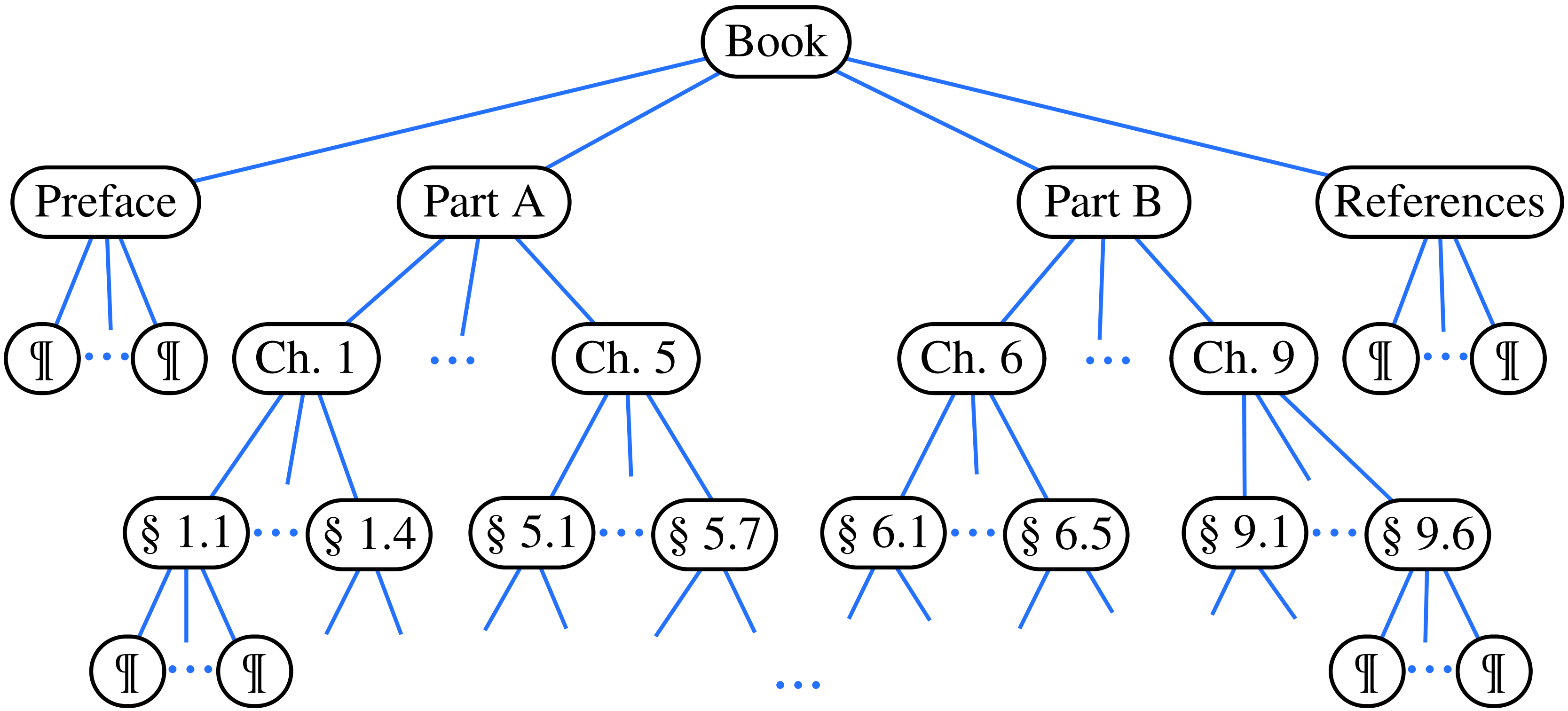
Application: OS file structure



Application: Phrase structure tree



Application: Document structure



Tree ADT

- Position as Node abstraction
- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()
- Access methods:
  - Position root()
  - Position parent(p)
  - Iterable children(p)
  - Integer numChildren(p)

- Query methods:
  - boolean isInternal(p)
  - boolean isExternal(p)
  - boolean isRoot(p)

- Additional update methods may be defined by data structures implementing the Tree ADT

Traversing trees

A traversal visits the nodes of a tree in a systematic manner

When traversing a simpler structure like a list there is one natural traversal strategy (forward or backwards)

Trees are more complex and admit more than one natural way:

- pre-order
- post-order
- in-order (for binary trees)

Preorder Traversal

To do a preorder traversal starting at a given node, we visit the node <u>before</u> visiting its descendants

```
def pre_order(v)
   visit(v)
   for each child w of v
      pre_order (w)
```

If tree is ordered visit the child subtrees in the prescribed order

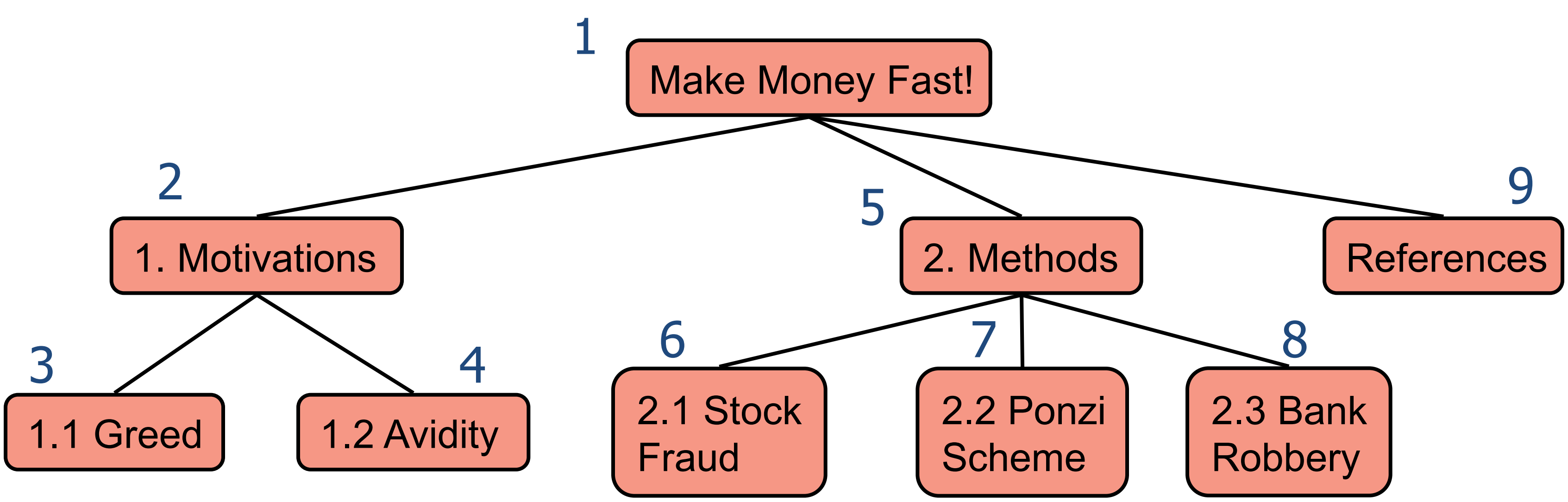
Visit does some work on the node:

- print node data
- aggregate node data
- modify node data

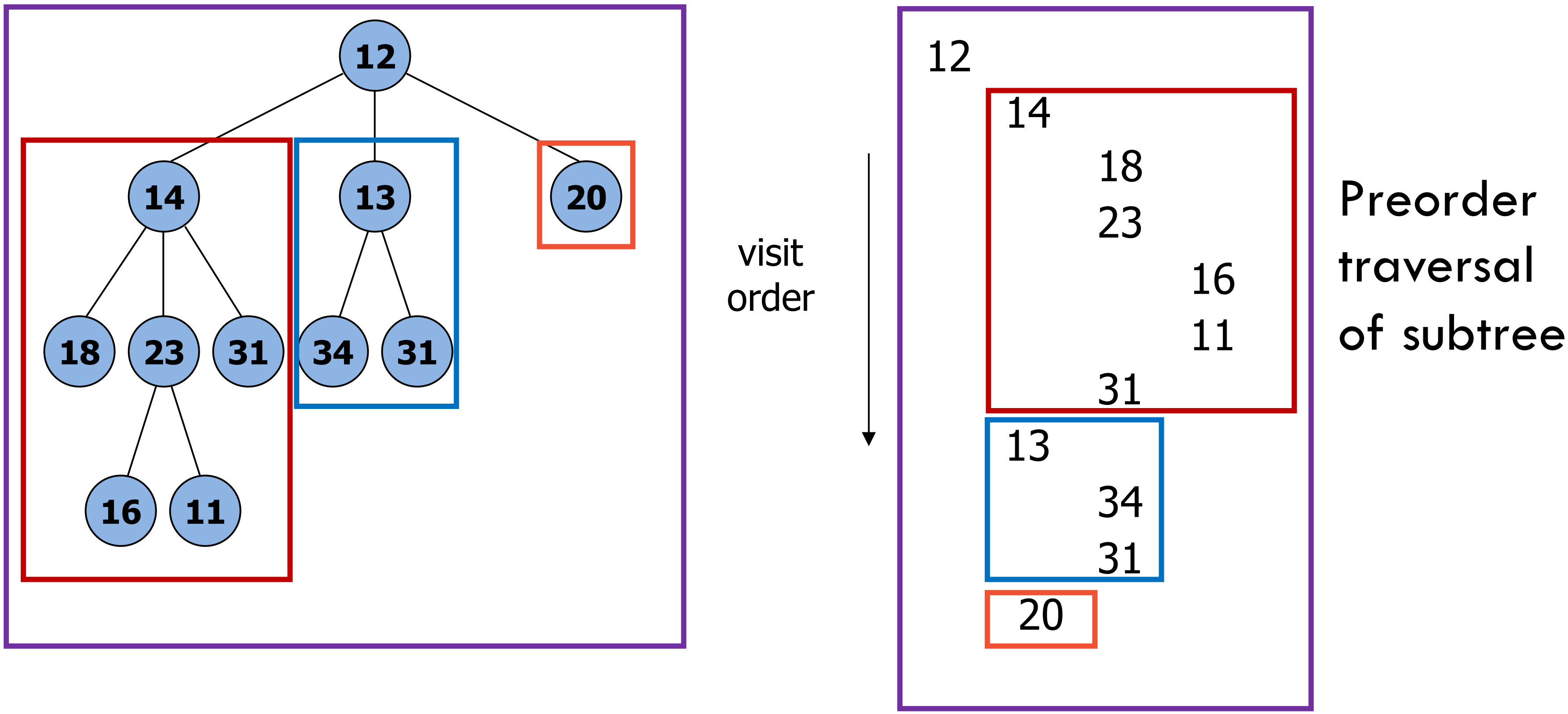
Preorder Traversal Example

Nodes are numbered in the order they are visited when we call `pre_order()` at the root

```
def pre_order(v)
    visit(v)
    for each child w of v
        pre_order (w)
```



Preorder Traversal Example



Postorder Traversal

To do a postorder traversal starting at a given node, we visit the node <u>after</u> its descendants

```
def post_order(v)
   for each child w of v
      post_order (w)
   visit(v)
```

If tree is ordered visit the child subtrees in the prescribed order

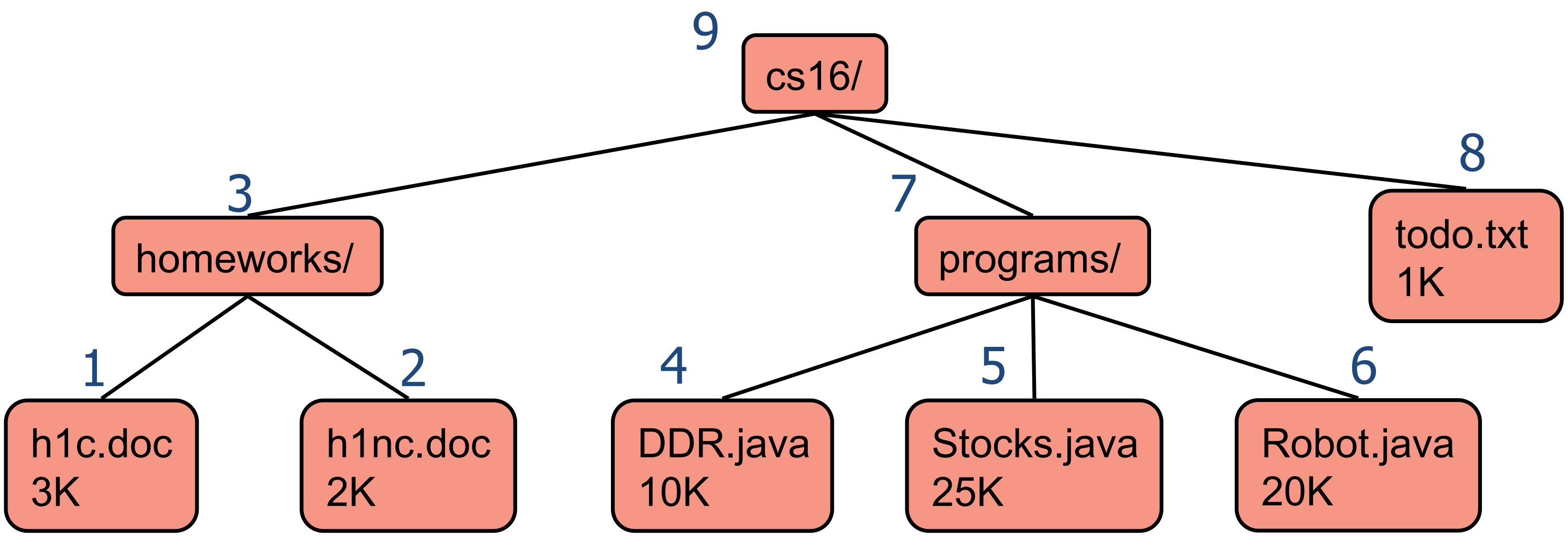
Visit does some work on the node:

- print node data
- aggregate node data
- modify node data

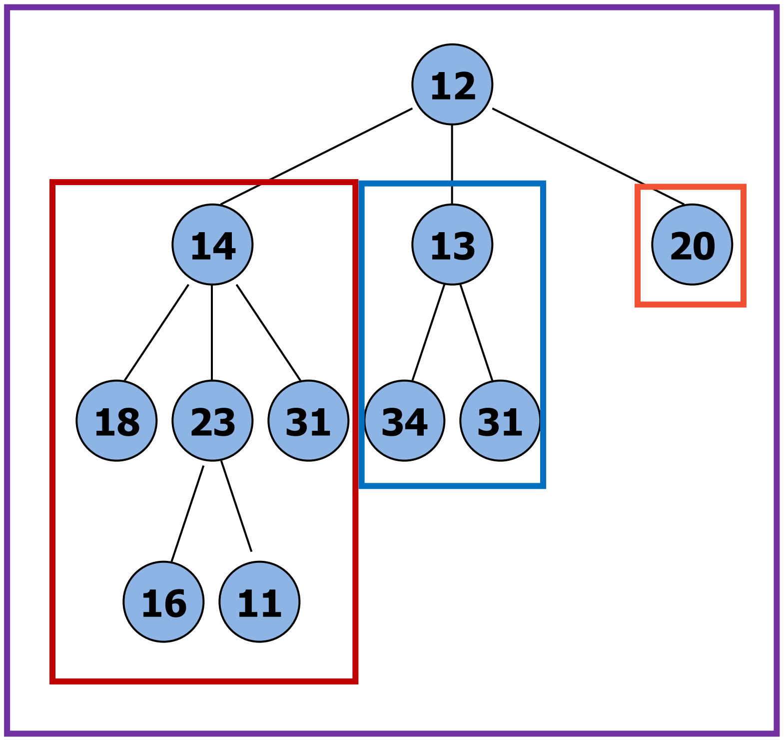
Postorder Traversal

Nodes are numbered in the order they are visited when we call `post_order()` at the root

```
def post_order(v)
   for each child w of v
      post_order (w)
   visit(v)
```



Traversing in postorder



visit order

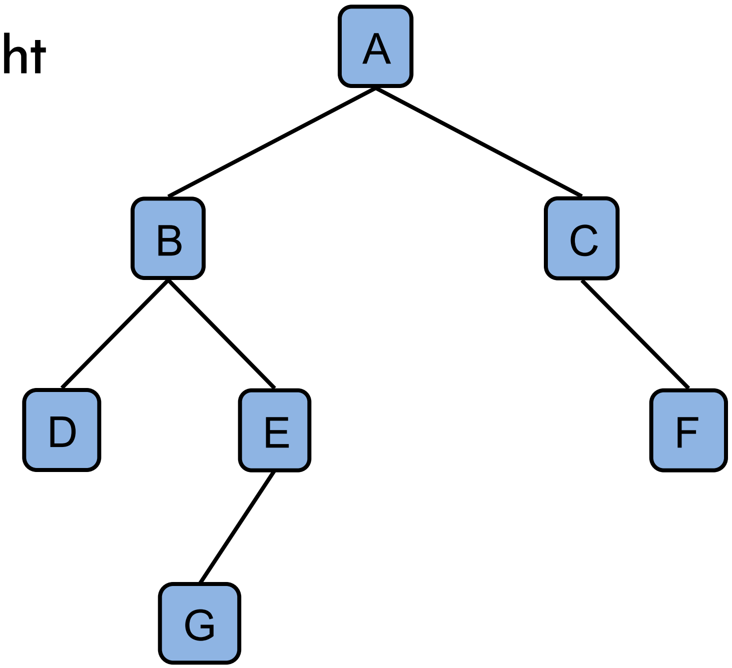
18
16
11
23
31
14
34
31
13
20
12

Postorder traversal of subtree

Binary Trees

A binary tree is an ordered tree with the following properties:

- Each internal node has at most two children
- Each child node is labeled as a left child or a right child
- Child ordering is left followed by right



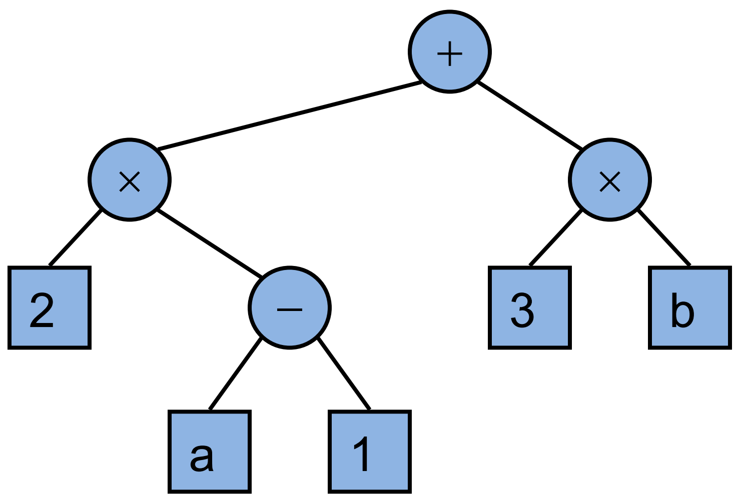
The right/left subtree is the subtree root at the right/left child.

We say the tree is proper if every internal node has two children

Binary tree application: Arithmetic expression tree

Binary tree associated with an arithmetic expression
- internal nodes: operators
- external nodes: operands

Example: Arithmetic expression tree for $(2 \times (a - 1) + (3 \times b))$

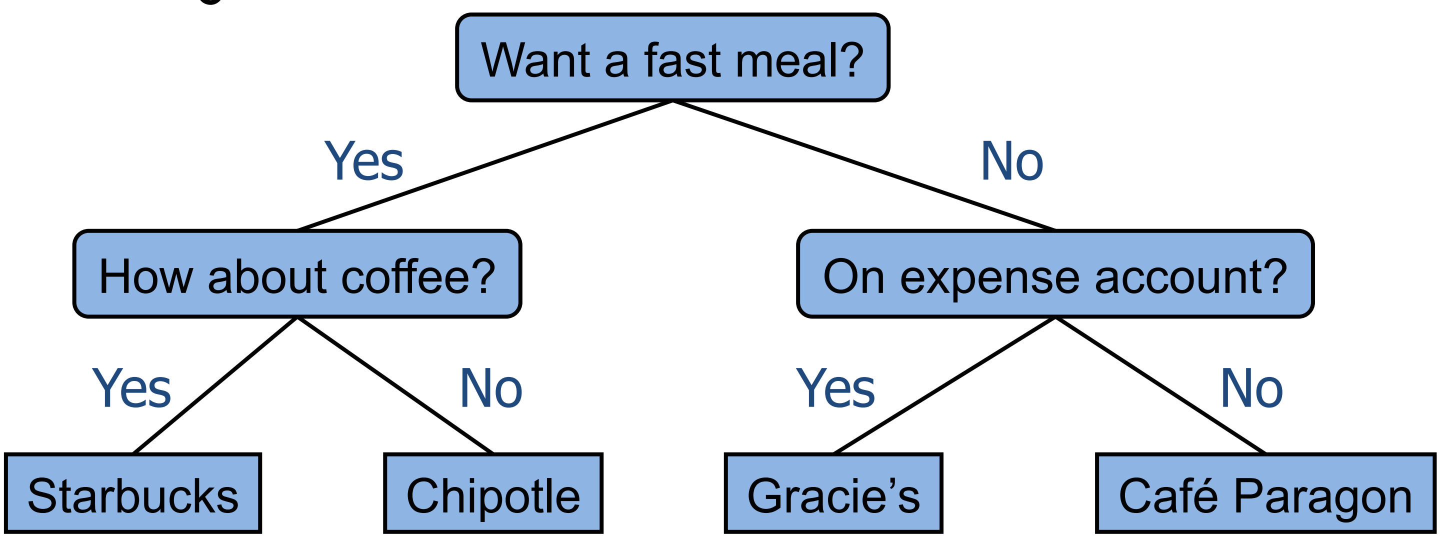


Binary tree application: Decision trees

Tree associated with a decision process

- internal nodes: questions with yes/no answer
- external nodes: decisions

Example: dining decision



Binary Tree Operations

- A binary tree extends the Tree operations, i.e., it inherits all the methods of a tree.
- Update methods may be defined by data structures implementing the binary tree
- Additional methods:
  - position leftChild(p)
  - position rightChild(p)
  - position sibling(p)

return null when there is no left, right, or sibling of p, respectively

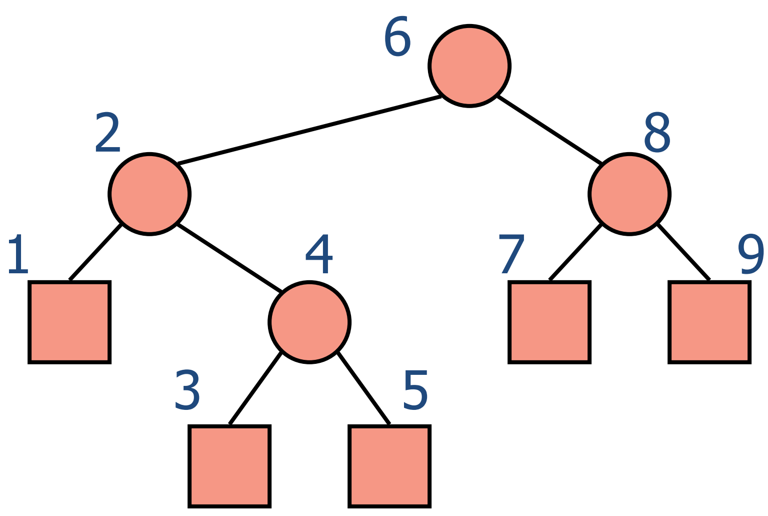
Inorder Traversal

To do an inorder traversal starting at a given node, the node is visited <u>after</u> its left subtree but <u>before</u> its right subtree

Visit does some work on the node:

- print node data
- aggregate node data
- modify node data

```
def in_order(v)
   if v.left ≠ null then
      in_order(v.left)
   visit(v)
   if v.right ≠ null then
      in_order(v.right)
```



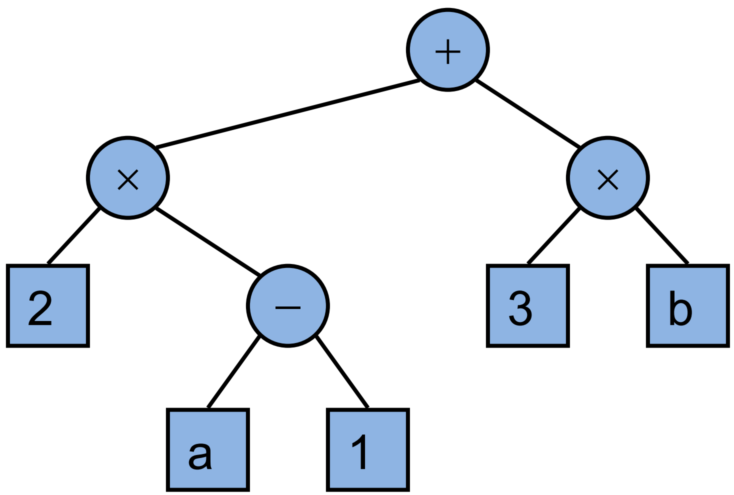
Back to arithmetic expression trees

Binary tree associated with an arithmetic expression

- internal nodes: operators
- external nodes: operands

What traversal would you use to:

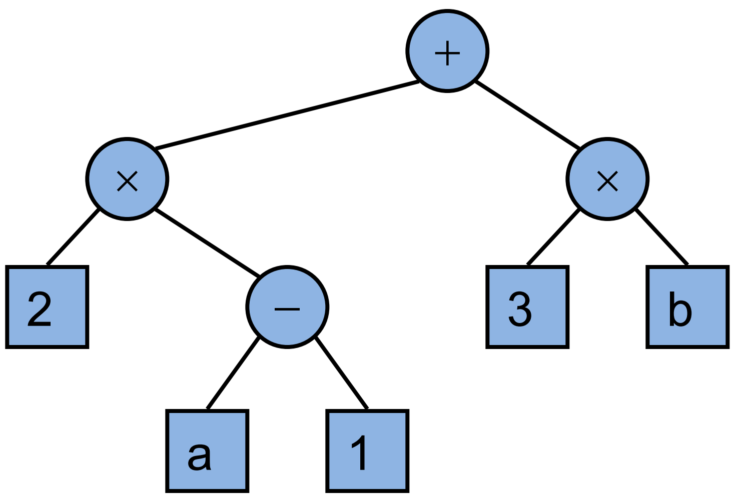
- print expression
- evaluate expression



Print Arithmetic Expressions

Extended inorder traversal:

- print operand or operator when visiting node
- print "(" before left subtree
- print ")" after right subtree




```
def print_expr(v)
    if v.left ≠ null then
        print("(")
        print_ expr(v.left)
    print(v.element)
    if v.right ≠ null then
        print_expr(v.right)
        print (")")
```

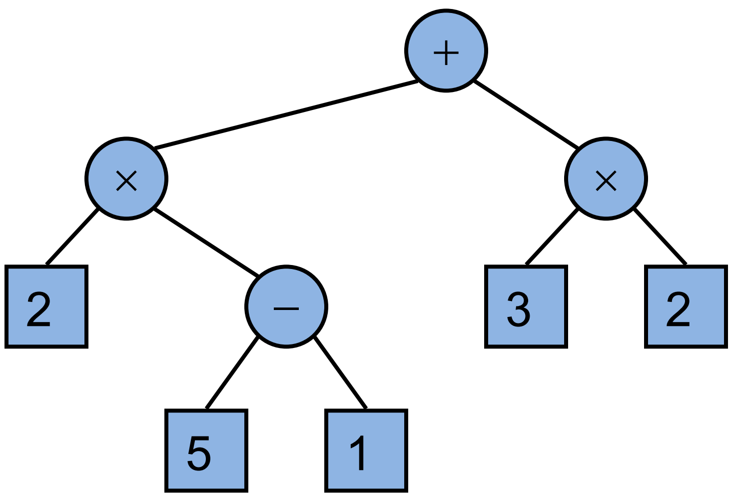
$$((2 \times (a - 1)) + (3 \times b))$$

Evaluate Arithmetic Expressions

Extended postorder traversal

- recursive method returning the value of a subtree
- when visiting an internal node, combine the values of the subtrees

```
def eval_expr(v)
  if v.is_external() then
    return v.element
  else
    x ← eval_expr(v.left)
    y ← eval_expr(v.right)
    ⊕ ← v.element
    return x ⊕ y
```

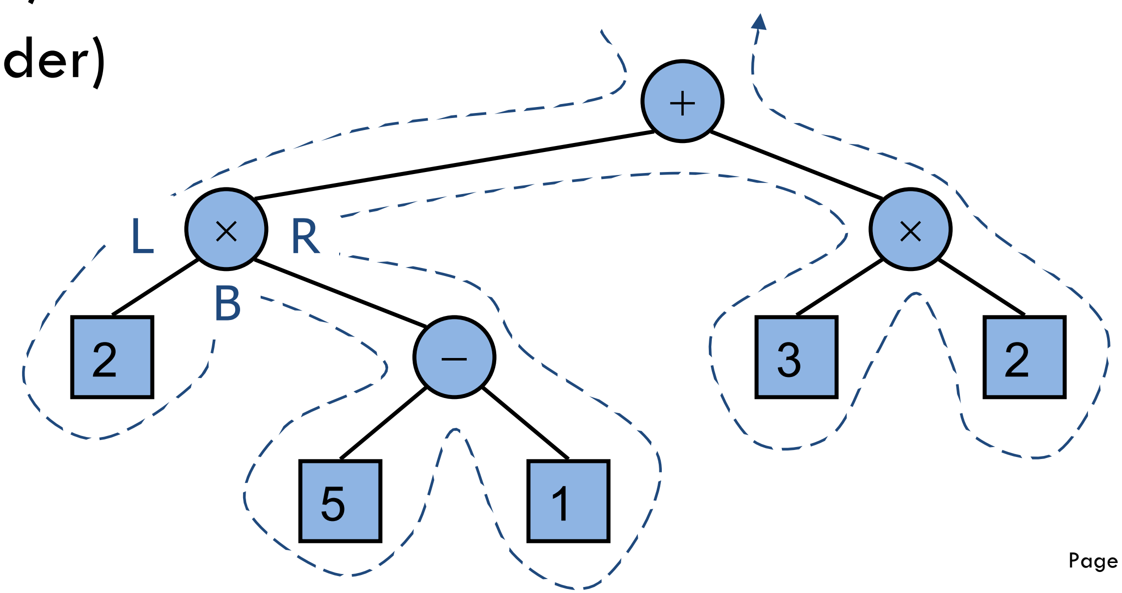


Euler Tour Traversal

Generic traversal of a binary tree. Includes as special cases the preorder, postorder and inorder traversals

Walk around the tree and visit each node three times:

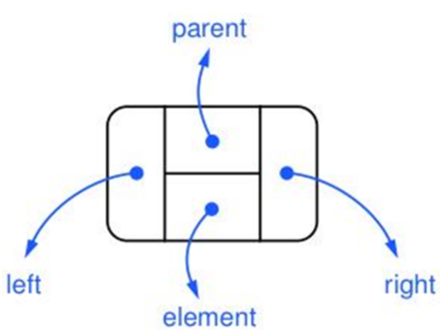
- on the left (preorder)
- from below (inorder)
- on the right (postorder)

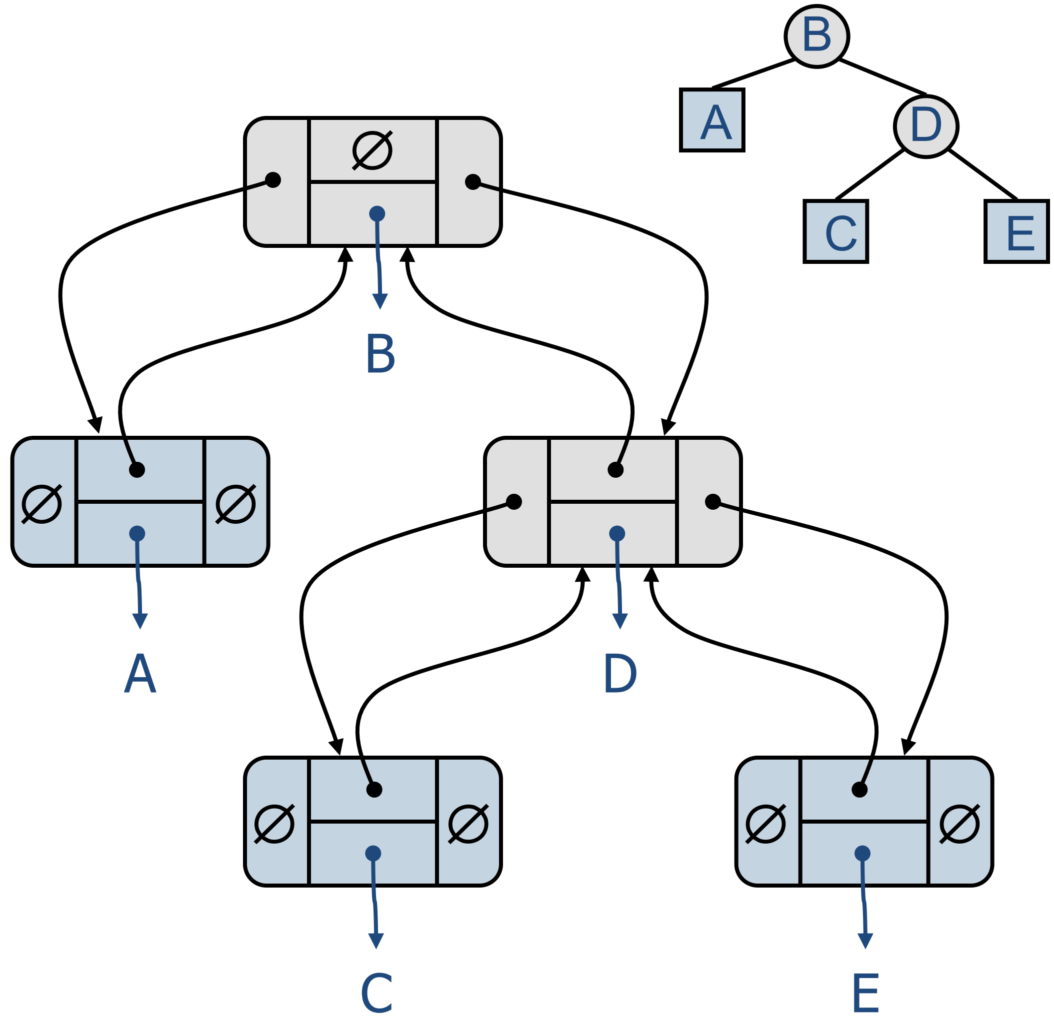


Linked Structure for Binary Trees

A node is represented by an object storing
- Element
- Parent node
- Left child node
- Right child node

Node objects implement the Position ADT



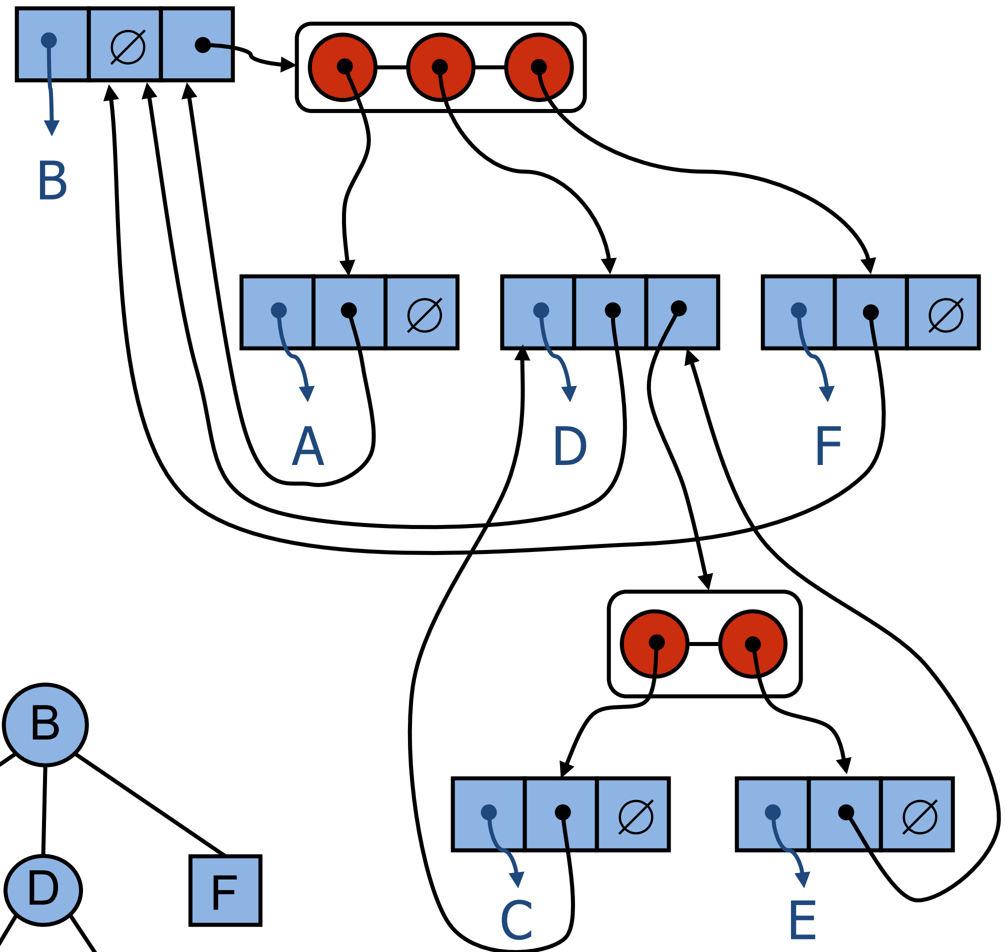


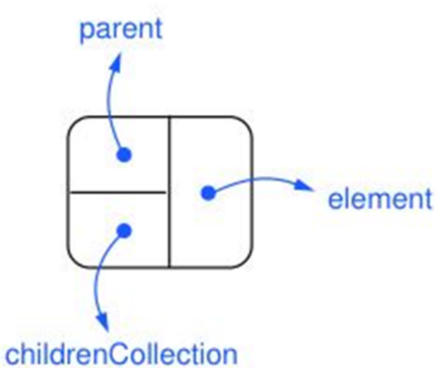
Linked Structure for General Trees

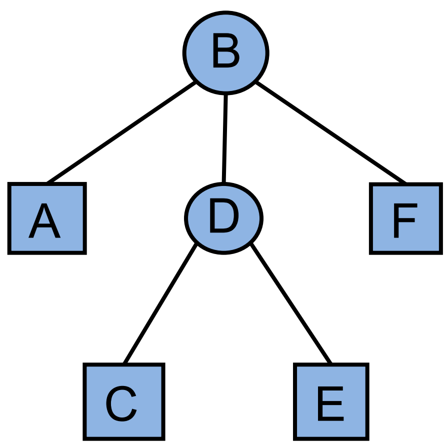
A node is represented by an object storing

- Element
- Parent node
- Sequence of children

Node objects implement the Position ADT







Examples of recursive code on trees

Calculating depth

```
def depth(v)
    if v.parent = null then
        return 0
    else
        return depth(v.parent) + 1
```

Calculating height

```
def height(v)
    if v.isExternal() then
        return 0
    else
        h ← 0
        for each child w of v
            h ← max(h, height(w))
        return h + 1
```

Complexity analysis of recursive algorithms on trees

Sometimes, the method may call itself on all children

- In worst case, do a call on every node
- If the work done, *excluding the recursion*, is constant per call, then the total cost is linear in the number of nodes

Sometimes, the method calls itself on at most one child

- In worst case, do one call at each level of the tree
- If the work done, *excluding the recursion*, is constant per call, then the total cost is linear in the height of the tree

Binary Search Tree

So far we've been focused on the structure of the tree. The real usefulness of trees hinges on the values we store at each element and how these values are laid out.

BST is a data structure for storing values that can be sorted. These values are laid out so that an in-order traversal of the BST visits the values in sorted order.

Can search for elements and insert/delete operations run in O(log n) time provided the tree is "balanced". More on that next week!