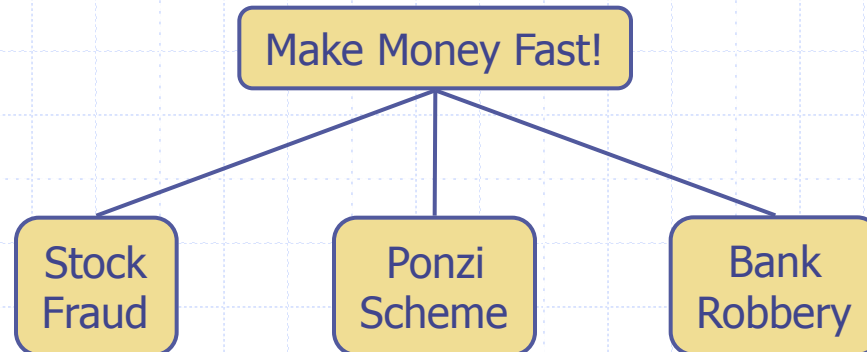
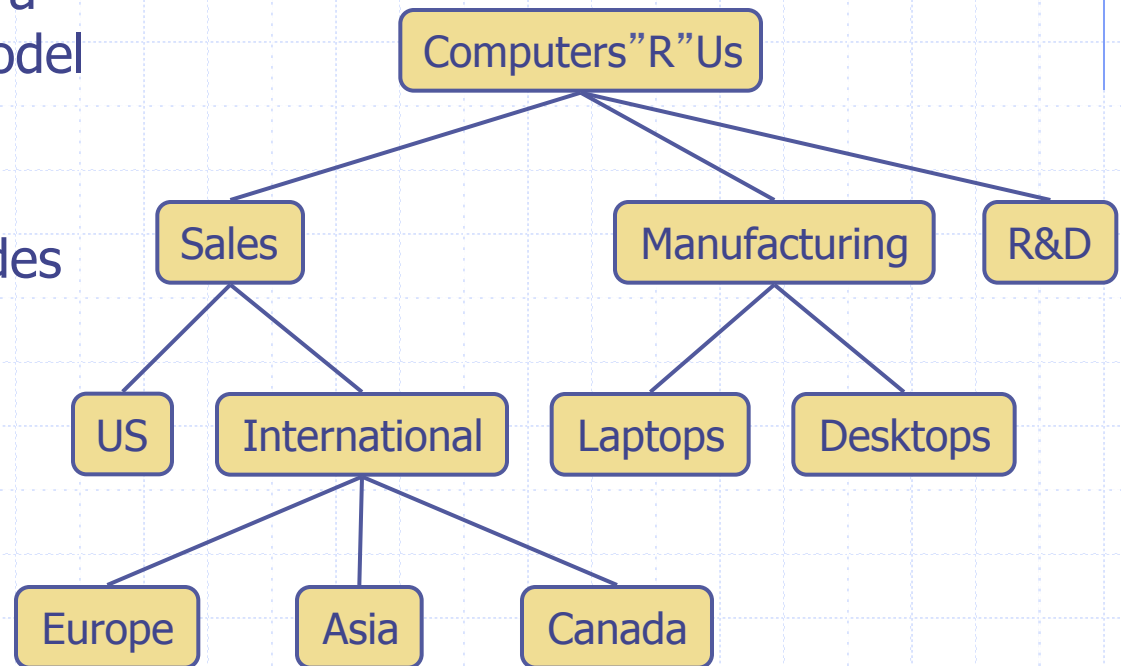


Trees



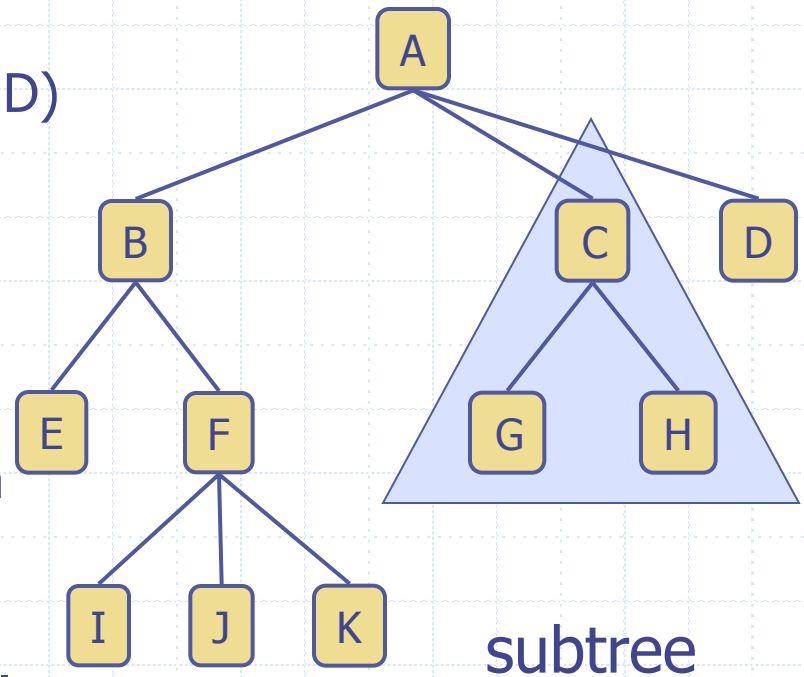
What is a Tree

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
 - Organization charts
 - File systems
 - Programming environments



Tree Terminology

- ❑ Root: node without parent (A)
- ❑ Internal node: node with at least one child (A, B, C, F)
- ❑ External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- ❑ Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- ❑ Depth of a node: number of ancestors
- ❑ Height of a tree: maximum depth of any node (3)
- ❑ Descendant of a node: child, grandchild, grand-grandchild, etc.
- ❑ Subtree: tree consisting of a node and its descendants



Tree ADT

- We use positions to abstract nodes
- Generic methods:
 - Integer `len()`
 - Boolean `is_empty()`
 - Iterator `positions()`
 - Iterator `iter()`
- Accessor methods:
 - position `root()`
 - position `parent(p)`
 - Iterator `children(p)`
 - Integer `num_children(p)`
- ◆ Query methods:
 - Boolean `is_leaf(p)`
 - Boolean `is_root(p)`
- ◆ Update method:
 - element `replace(p, o)`
- ◆ Additional update methods may be defined by data structures implementing the Tree ADT

Abstract Nested Position Class in Python

```
#ADT Tree "interface"
class Tree:

    class Position:

        def element( self ):
            pass

        def __eq__( self, other ):
            pass

        def __ne__( self, other ):
            return not( self == other )
```

Abstract Tree Class in Python...

```
#ADT Tree "interface"
```

```
class Tree:
```

```
    def root( self ):
        pass
```

```
    def parent( self, p ):
        pass
```

```
    def num_children( self, p ):
        pass
```

```
    def children( self, p ):
        pass
```

```
    def __len__( self ):
        pass
```

```
    def is_root( self, p ):
        return self.root() == p
```

```
    def is_leaf( self, p ):
        return self.num_children() == 0
```

```
    def is_empty( self ):
        return len( self ) == 0
```

Trees

Abstract Tree Class in Python

```
def depth( self, p ):
    #returns the number of ancestors of p
    if self.is_root( p ):
        return 0
    else:
        return 1 + self.depth( self.parent() )

def height1( self, p ):
    #returns the maximum depth of the leaf positions
    return max( self.depth( p ) for p in self.positions() if self.is_leaf( p ) )

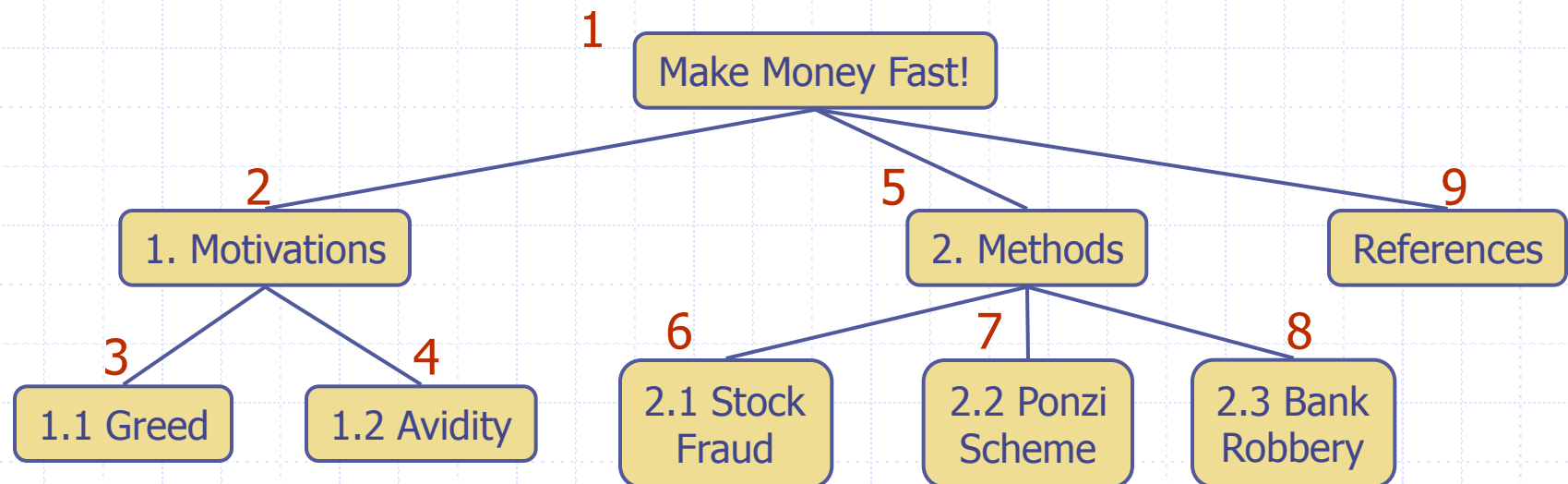
def height2( self, p ):
    #returns the height of the subtree at Position p
    if self.is_leaf( p ):
        return 0
    else:
        return 1 + max( self.height2( c ) for c in self.children( p ) )

def height( self, p = None ):
    #returns the height of the subtree rooted at Position p
    #if p is None, then the height of the entire tree
    if p is None:
        p = self.root()
    return self.height2( p )
```

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

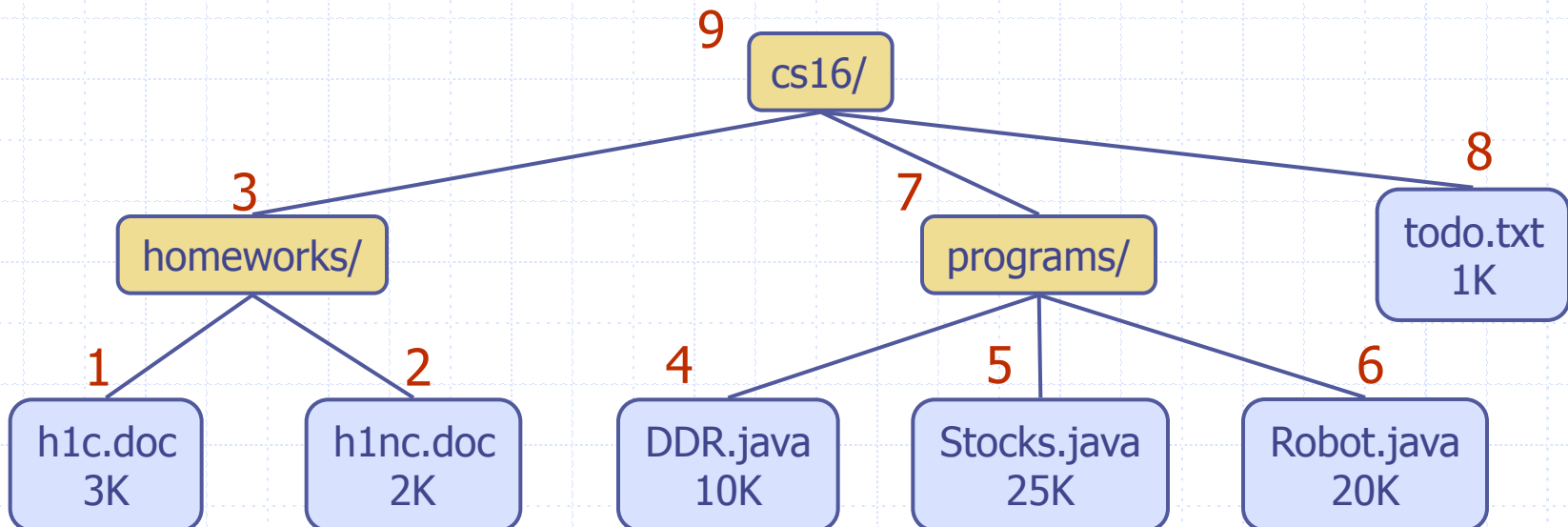
Algorithm *preOrder(v)*
visit(v)
for each child *w* of *v*
preorder(w)



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm *postOrder(v)*
for each child *w* of *v*
 postOrder(w)
visit(v)



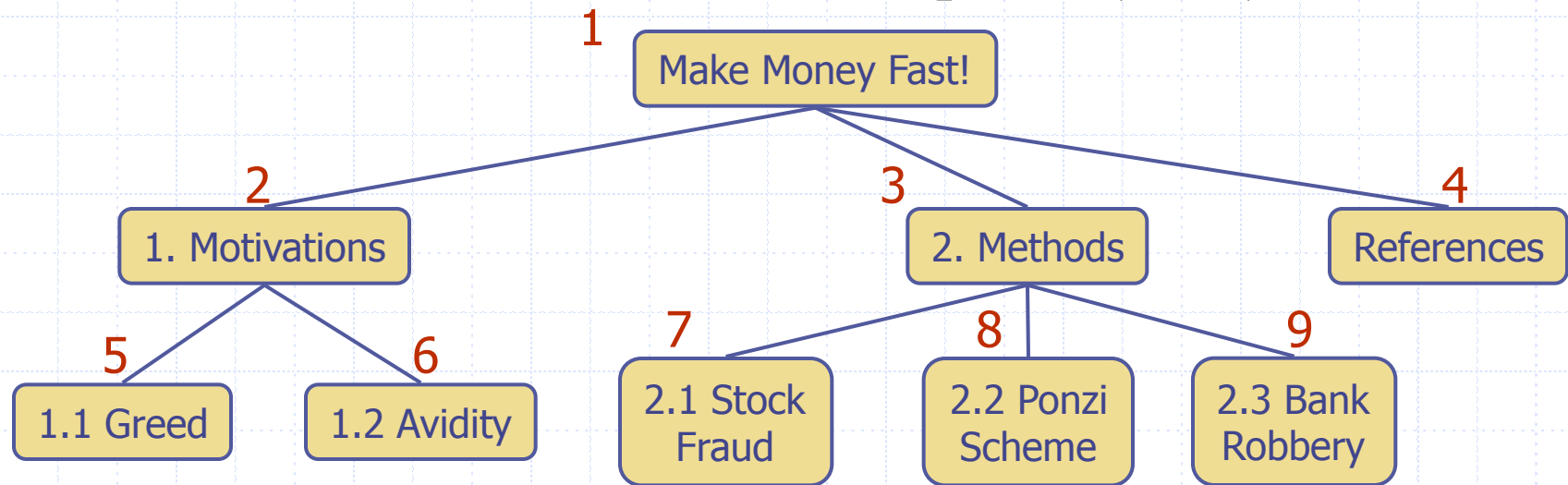
pre- and postorder traversal in Python

```
def preorder_print( self, p ):
    print( p )
    for c in self.children( p ):
        preorder_print( c )

def postorder_print( self, p ):
    for c in self.children( p ):
        postorder_print( c )
    print( p )
```

Breadth-first traversal in Python

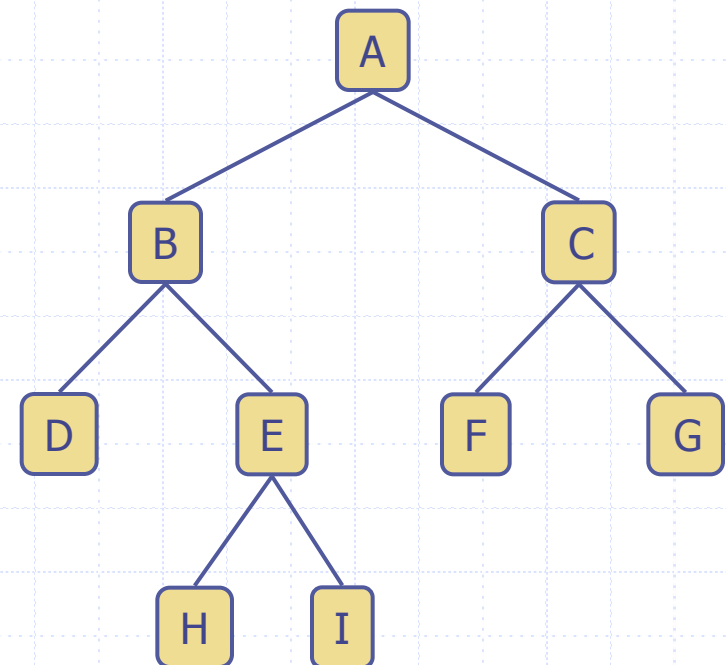
```
def breadth_first_print( self ):
    Q = ArrayQueue()
    Q.enqueue( self.root() )
    while not Q.is_empty():
        p = Q.dequeue()
        print( p )
        for c in self.children( p ):
            Q.enqueue( c )
```



Binary Trees

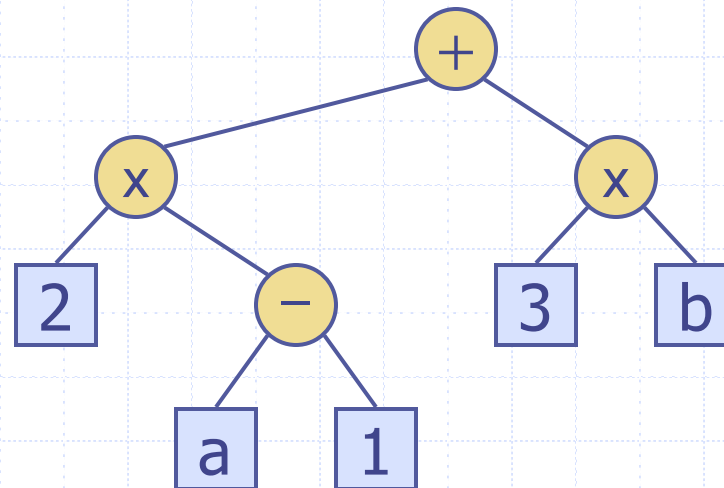
- A binary tree is a tree with the following properties:
 - Each internal node has at most two children (exactly two for **proper** binary trees)
 - The children of a node are an ordered pair
- We call the children of an internal node **left child** and **right child**
- Alternative recursive definition: a binary tree is either
 - a tree consisting of a single node, or
 - a tree whose root has an ordered pair of children, each of which is a binary tree

- Applications:
 - arithmetic expressions
 - decision processes
 - searching



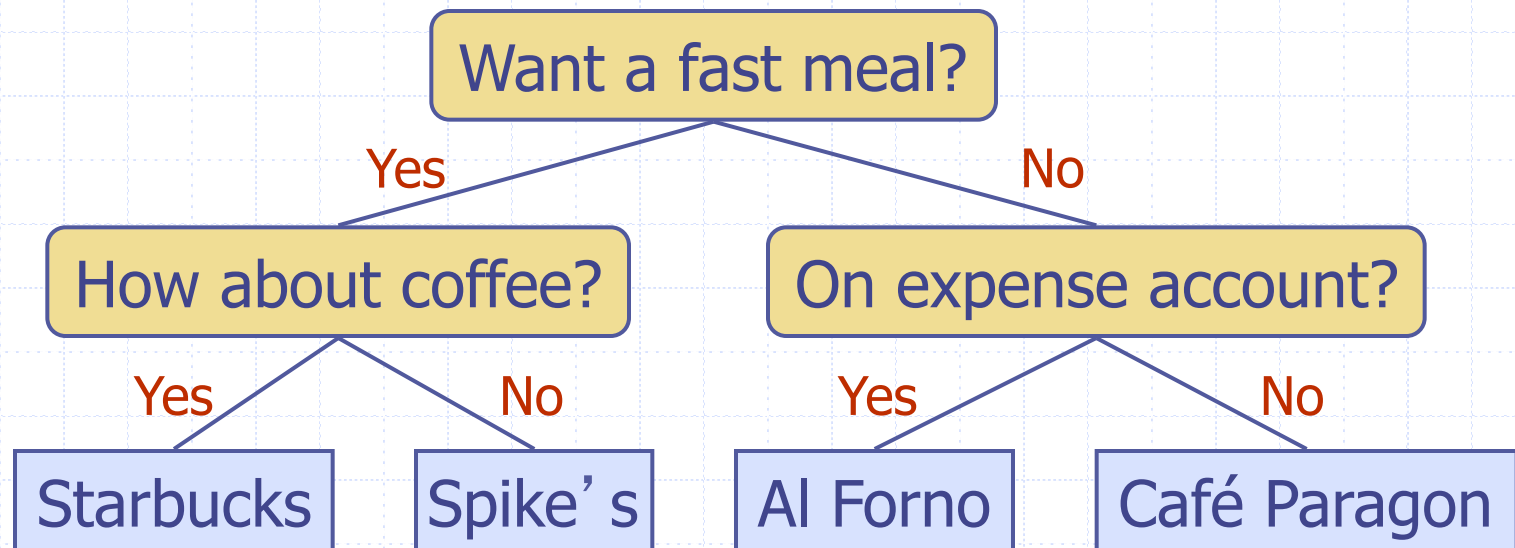
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
 - internal nodes: operators
 - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



Decision Tree

- Binary tree associated with a decision process
 - internal nodes: questions with yes/no answer
 - external nodes: decisions
- Example: dining decision



Properties of Proper Binary Trees

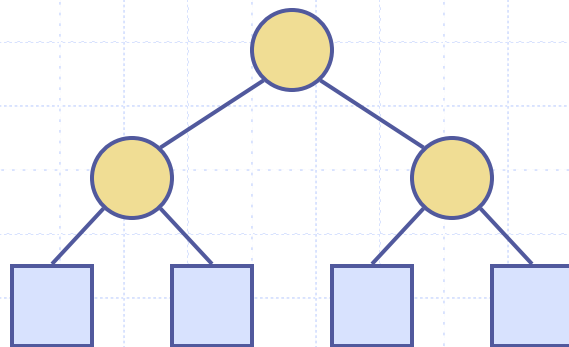
□ Notation

n number of nodes

e number of external nodes

i number of internal nodes

h height



◆ Properties:

■ $e = i + 1$

■ $n = 2e - 1$

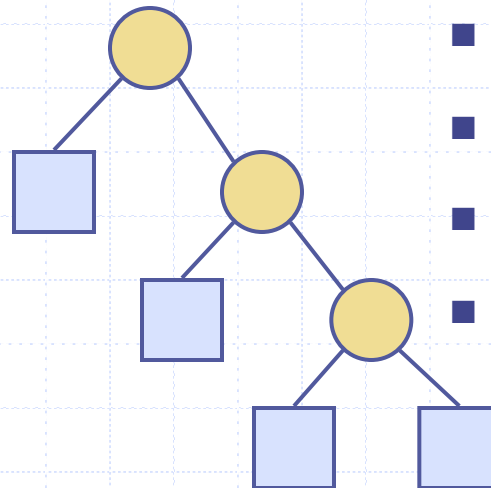
■ $h \leq i$

■ $h \leq (n - 1)/2$

■ $e \leq 2^h$

■ $h \leq \log_2 e$

■ $h \geq \log_2 (n + 1) - 1$



BinaryTree ADT

- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
 - position **left**(p)
 - position **right**(p)
 - position **sibling**(p)
- Update methods may be defined by data structures implementing the BinaryTree ADT

Abstract BinaryTree Class in Python

```
from Tree import Tree

class BinaryTree( Tree ):

    def left( self, p ):
        pass

    def right( self, p ):
        pass

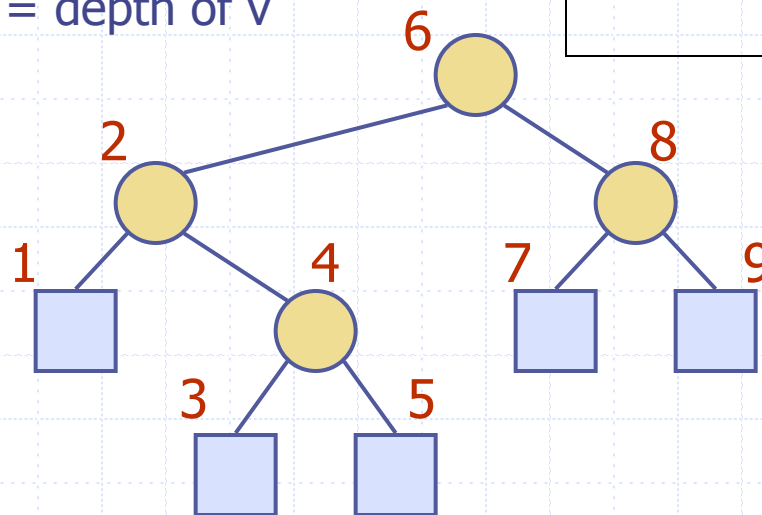
    def sibling( self, p ):
        #return the sibling Position
        parent = self.parent()
        if parent is None:
            return None
        else:
            if p == self.left( parent ):
                return self.right( parent )
            else:
                return self.left( parent )

    def children( self, p ):
        if self.left( p ) is not None:
            yield self.left( p )
        if self.right( p ) is not None:
            yield self.right( p )
```

Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
 - $x(v)$ = inorder rank of v
 - $y(v)$ = depth of v

```
Algorithm inOrder( $v$ )  
  if  $v$  has a left child  
    inOrder (left ( $v$ ))  
  visit( $v$ )  
  if  $v$  has a right child  
    inOrder (right ( $v$ ))
```

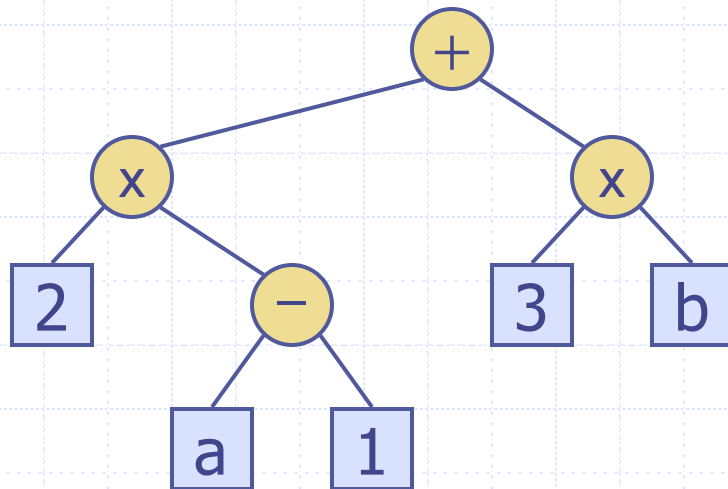


inorder_print in Python

```
def inorder_print( self, p ):
    if self.left( p ) is not None:
        self.inorder_print( self.left( p ) )
    print( p )
    if self.right( p ) is not None:
        self.inorder_print( self.right( p ) )
```

Print Arithmetic Expressions

- Specialization of an inorder traversal
 - print operand or operator when visiting node
 - print “(“ before traversing left subtree
 - print “)” after traversing right subtree



Algorithm *printExpression(v)*

if *v* **has a left child**

print (“(” ’ ’)

inOrder (*left(v)*)

print (*v.element* ())

if *v* **has a right child**

inOrder (*right(v)*)

print (“)” ’ ’)

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

Evaluate Arithmetic Expressions

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

Algorithm *evalExpr(v)*

if *is_leaf(v)*

return *v.element()*

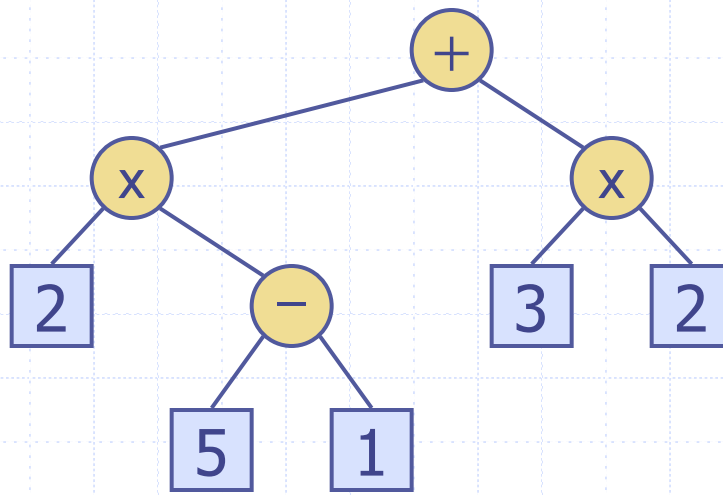
else

x = *evalExpr(left(v))*

y = *evalExpr(right(v))*

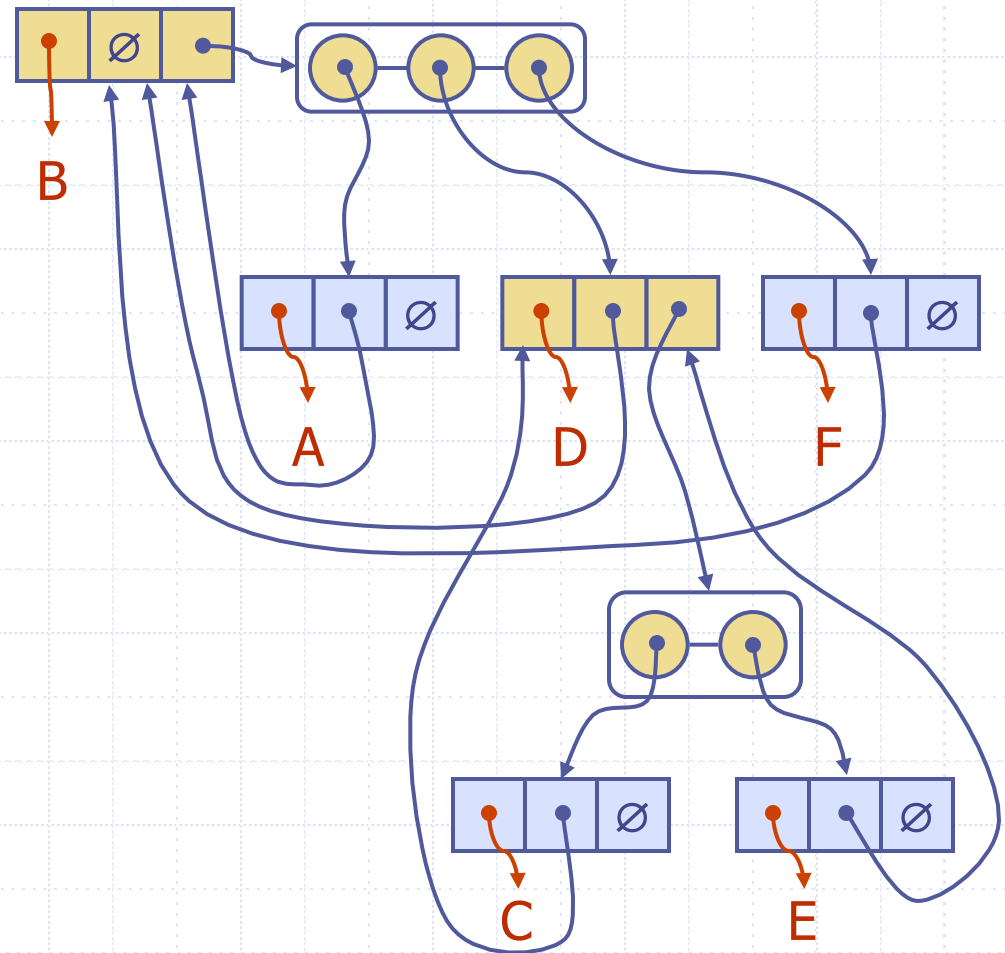
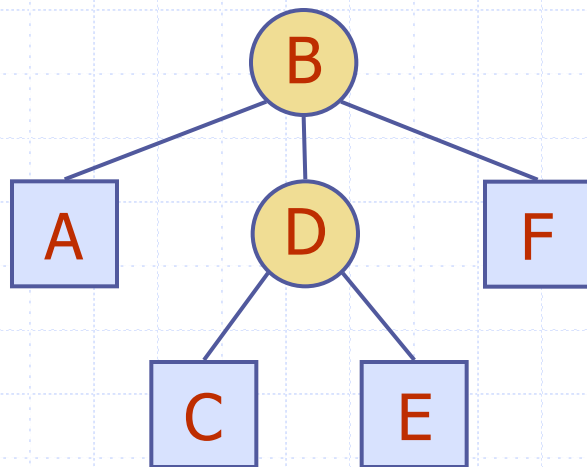
op = operator stored at *v*

return *x op y*



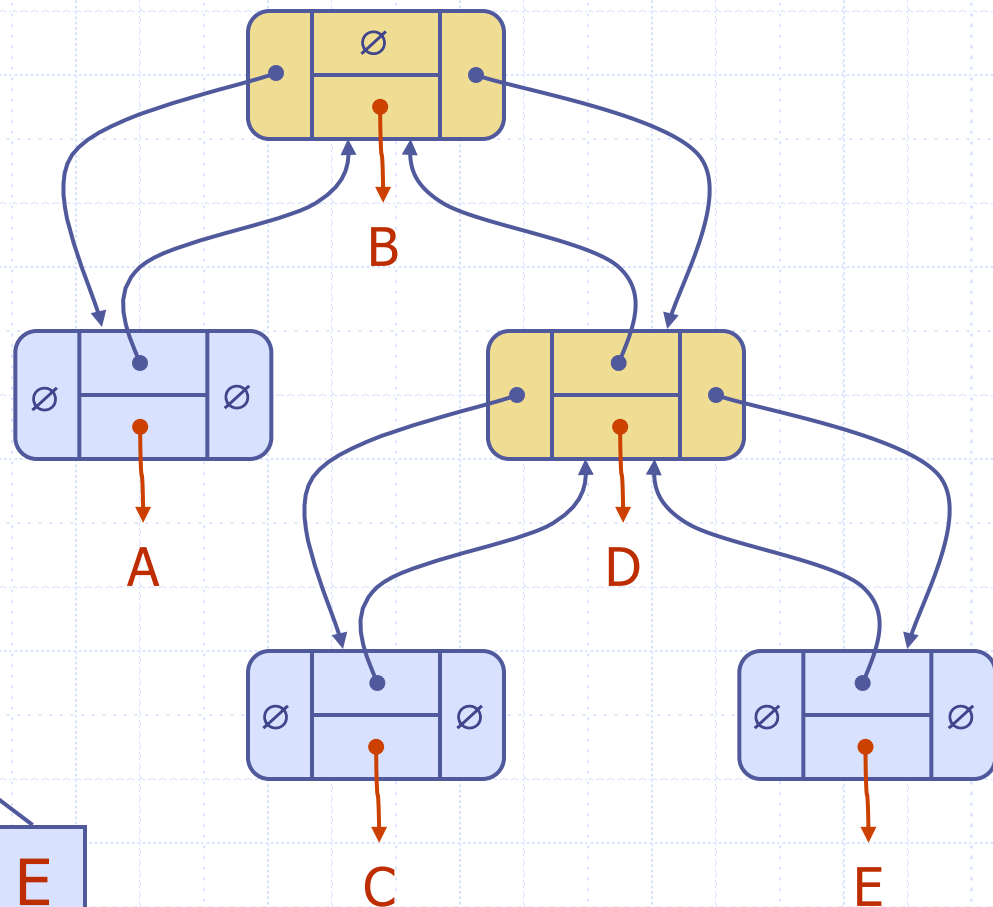
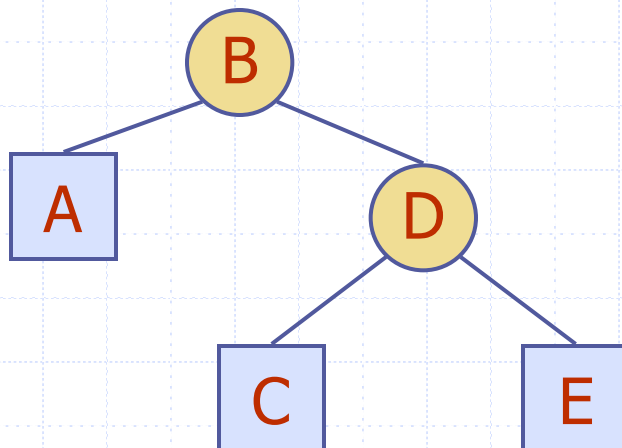
Linked Structure for Trees

- A node is represented by an object storing
 - Element
 - Parent node
 - Sequence of children nodes
- Node objects implement the Position ADT



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



LinkedBinaryTree Class in Python

```
from BinaryTree import BinaryTree

class LinkedBinaryTree( BinaryTree ):

    class _Node:
        def __init__( self, element,
                        parent = None,
                        left = None,
                        right = None ):
            self._element = element
            self._parent = parent
            self._left = left
            self._right = right
```


Nested Abstract Position

```
class Position( BinaryTree.Position ):

    def __init__( self, container, node ):
        self._container = container
        self._node = node

    def __str__( self ):
        return str( self._node._element )

    def element( self ):
        return self._node._element

    def __eq__( self, other ):
        return type( other ) is type( self ) and other._node is self._node
```

Validate and MakePosition

```
def _validate( self, p ):
    #return associated node if position is valid
    if not isinstance( p, self.Position ):
        raise TypeError( 'p must be proper Position type' )
    if p._container is not self:
        raise ValueError( 'p does not belong to this container' )
    if p._node._parent is p._node:
        raise ValueError( 'p is no longer valid' )
    return p._node

def _make_position( self, node ):
    #return Position instance for given node (None if no node)
    return self.Position( self, node ) if node is not None else None
```

LinkedBinaryTree...

```
def __init__( self ):
    #create an initially empty binary tree
    self._root = None
    self._size = 0

def __len__( self ):
    return self._size

def root( self ):
    return self._make_position( self._root )

def parent( self, p ):
    node = self._validate( p )
    return self._make_position( node._parent )

def left( self, p ):
    node = self._validate( p )
    return self._make_position( node._left )

def right( self, p ):
    node = self._validate( p )
    return self._make_position( node._right )
```

LinkedBinaryTree...

```
def num_children( self, p ):
    node = self._validate( p )
    count = 0
    if node._left is not None:
        count += 1
    if node._right is not None:
        count += 1
    return count

def _add_root( self, e ):
    if self._root is not None:
        raise ValueError( 'Root exists' )
    self._size = 1
    self._root = self._Node( e )
    return self._make_position( self._root )
```

LinkedBinaryTree...

```
def _add_left( self, p, e ):
    node = self._validate( p )
    if node._left is not None:
        raise ValueError( 'Left child exists' )
    self._size += 1
    node._left = self._Node( e, node )
    return self._make_position( node._left )

def _add_right( self, p, e ):
    node = self._validate( p )
    if node._right is not None:
        raise ValueError( 'Right child exists' )
    self._size += 1
    node._right = self._Node( e, node )
    return self._make_position( node._right )
```

LinkedBinaryTree...

```
def _replace( self, p, e ):
    node = self._validate( p )
    old = node._element
    node._element = e
    return old

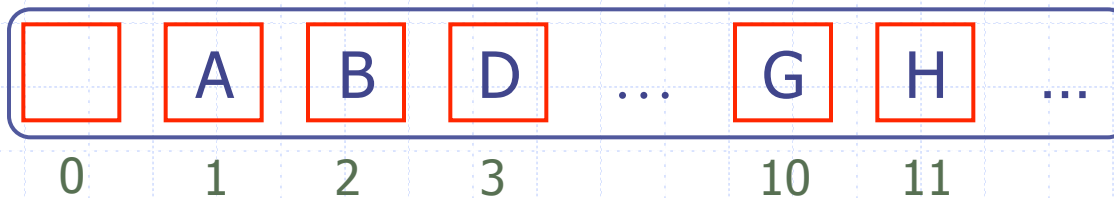
def _delete( self, p ):
    #remove node p and replace it with its child if any
    node = self._validate( p )
    if self._num_children( p ) == 2:
        raise ValueError( 'p has two children' )
    child = node._left if node._left else node._right
    if child is not None:
        child._parent = node._parent
    if node is self._root:
        self._root = child
    else:
        parent = node._parent
        if node is parent._left:
            parent._left = child
        else:
            parent._right = child
    self._size -= 1
    node._parent = None
    return node._element
```

LinkedBinaryTree...

```
def _attach( self, p, t1, t2 ):
    #attach trees t1 and t2 as left and right of leaf p
    node = self._validate( p )
    if not self.is_left( p ):
        raise ValueError( 'position must be leaf' )
    if not type( self ) is type( t1 ) is type( t2 ):
        raise TypeError( 'Tree types must match' )
    self._size += len( t1 ) + len( t2 )
    if not t1.is_empty():
        t1._root._parent = node
        node._left = t1._root
        t1._root = None #set t1 to empty
        t1._size = 0
    if not t2.is_empty():
        t2._root._parent = node
        node._right = t2._root
        t2._root = None #set t2 to empty
        t2._size = 0
```

Array-Based Representation of Binary Trees

- Nodes are stored in an array A



- Node v is stored at $A[\text{rank}(v)]$
 - $\text{rank}(\text{root}) = 1$
 - if node is the left child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 \times \text{rank}(\text{parent}(\text{node}))$
 - if node is the right child of $\text{parent}(\text{node})$,
 $\text{rank}(\text{node}) = 2 \times \text{rank}(\text{parent}(\text{node})) + 1$

