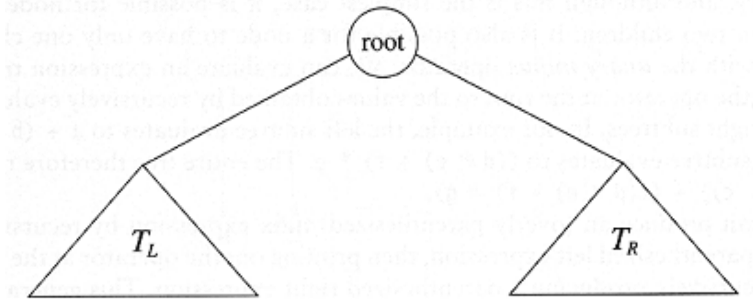
BINARY TREES (8.2)

EECS 2011

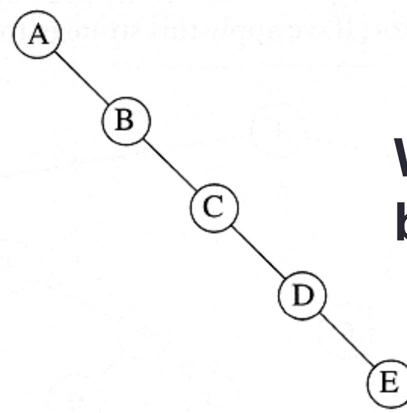
Binary Trees

- A tree in which each node can have at most two children.



Generic binary tree

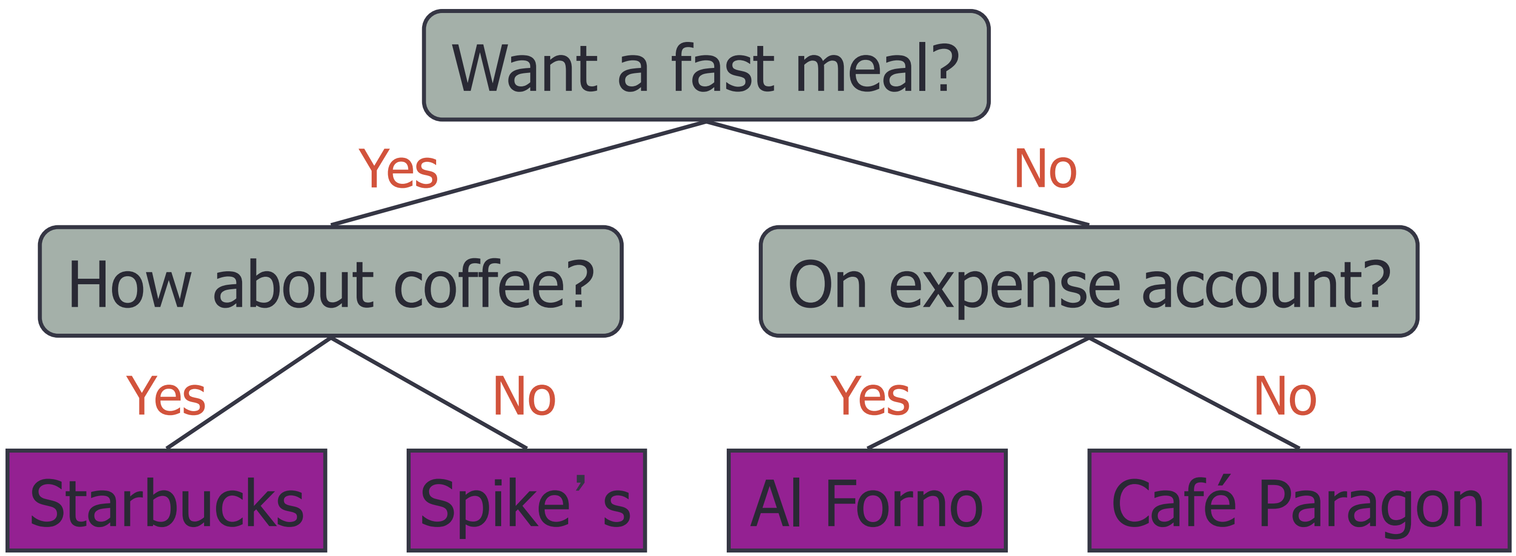
- The depth of an "average" binary tree is considerably smaller than N. In the worst case, the depth can be as large as N – 1.



Worst-case binary tree

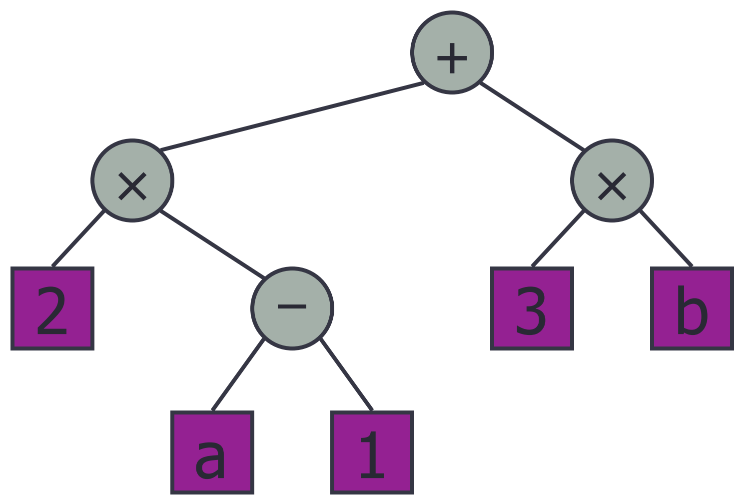
Decision Tree

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



Arithmetic Expression Tree

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



Tree ADT (review)

- We use positions to abstract nodes (position ≡ node)

- Accessor methods:
  - position root()
  - position parent(p)
  - Iterable children(p)
  - Integer numChildren(p)

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

- Generic methods:
  - integer size()
  - boolean isEmpty()
  - Iterator iterator()
  - Iterable positions()

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

BinaryTree ADT

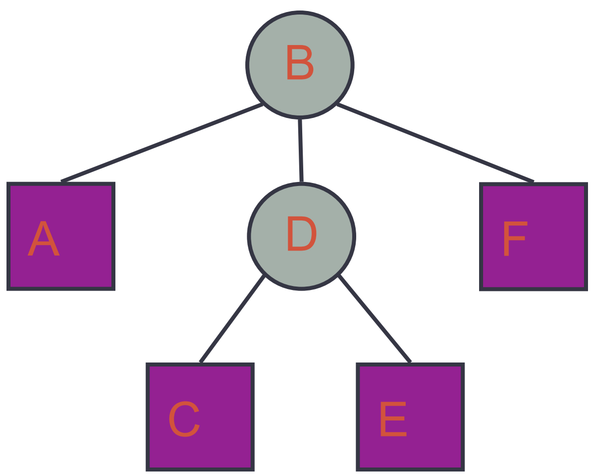
- The BinaryTree ADT extends the Tree ADT
- It inherits all the methods of the Tree ADT
- Additional methods:
  - position left(p)
  - position right(p)
  - position sibling(p)
- The above methods return null when there is no left, right, or sibling of p, respectively
- Update methods may be defined by data structures implementing the BinaryTree ADT

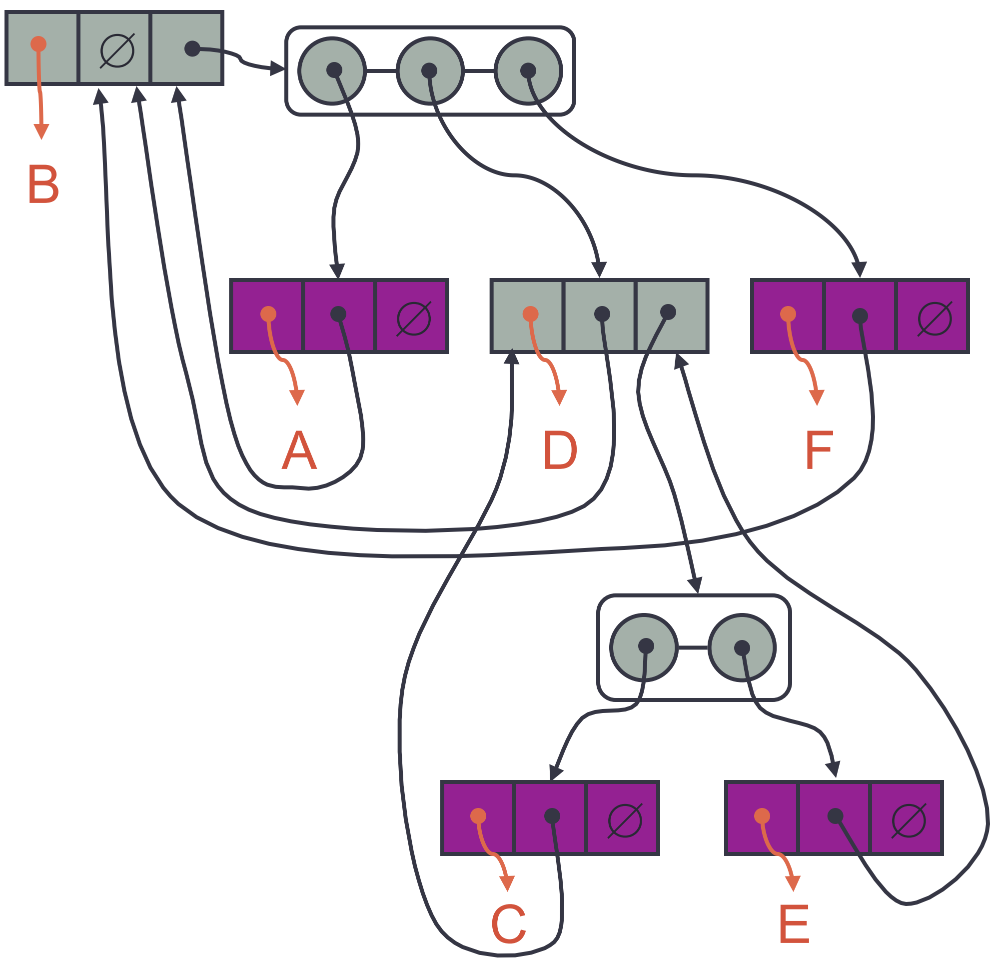
Implementing Binary Trees

- Arrays?
  - Discussed later
- Linked structure?

Linked Structure for General Trees (8.3.3)

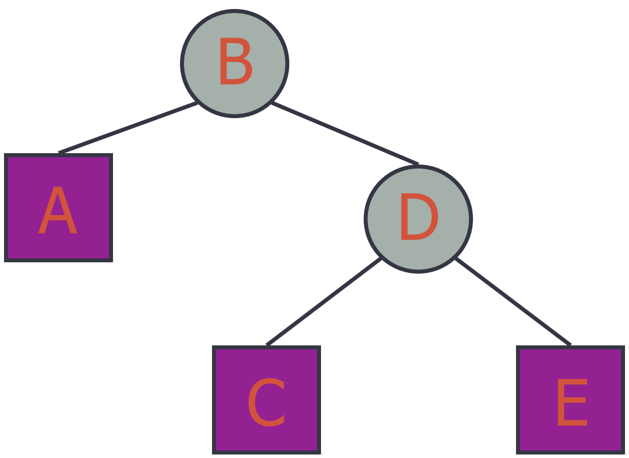
- A node is represented by an object storing
  - Element
  - Parent node
  - Sequence of children nodes

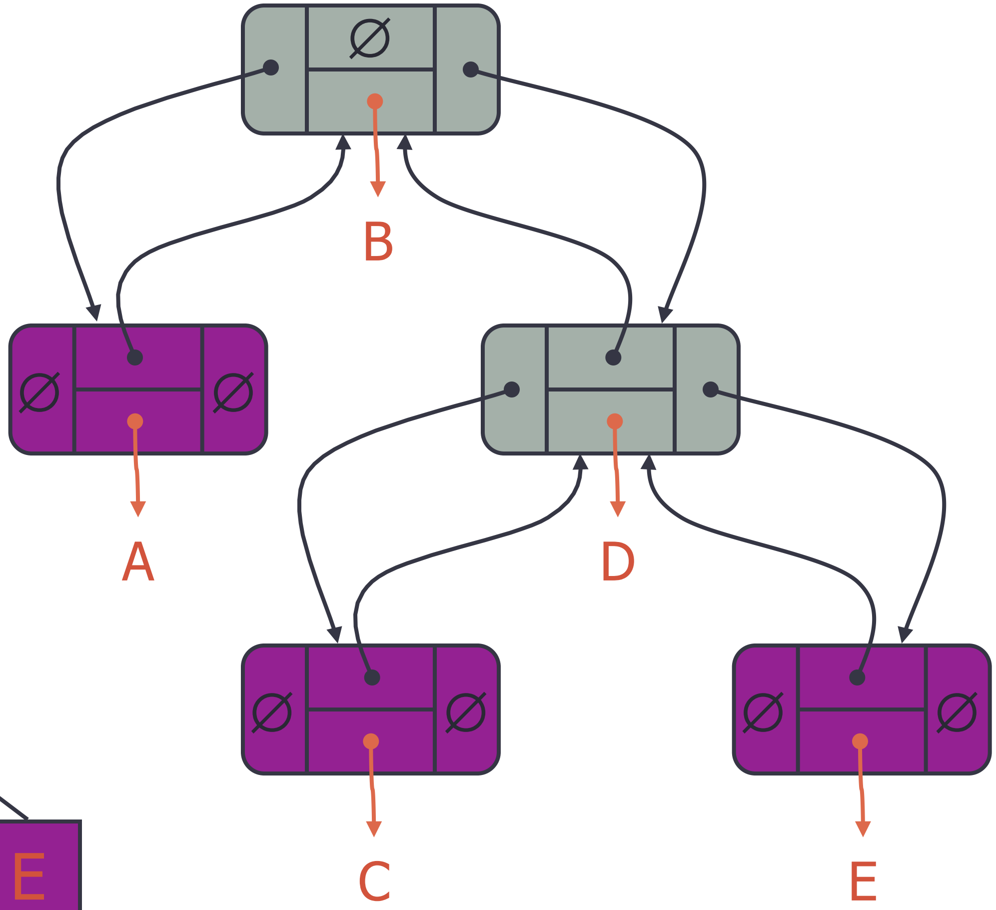




Linked Structure for Binary Trees (8.3.1)

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





Linked Structure for Binary Trees

```
class BinaryNode {
  Object        element;
  BinaryNode    left;
  BinaryNode    right;
  BinaryNode    parent;
}
```

- BinaryNode objects implement the Position ADT.
- Java implementation of a linked binary tree: Code Fragments 8.8, 8.9

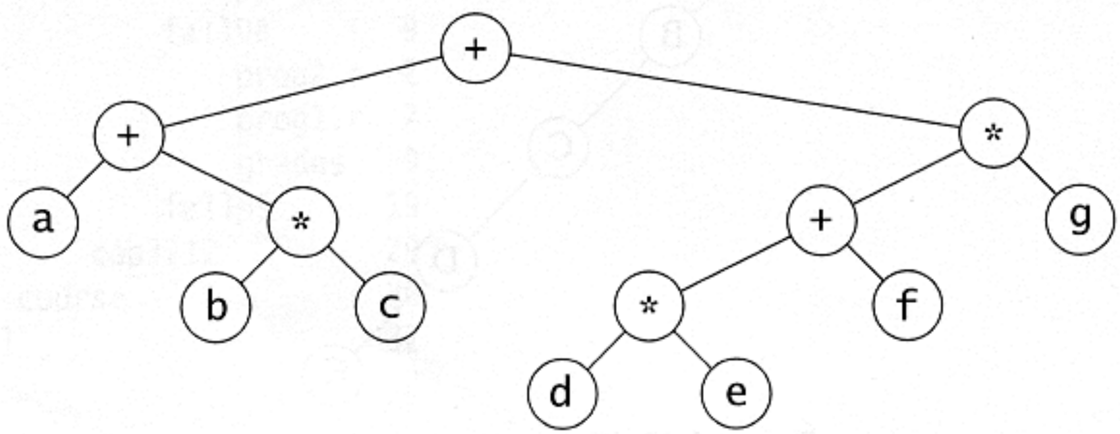


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Binary Tree Traversal

- Preorder (node, left, right)
- Postorder (left, right, node)
- Inorder (left, node, right)

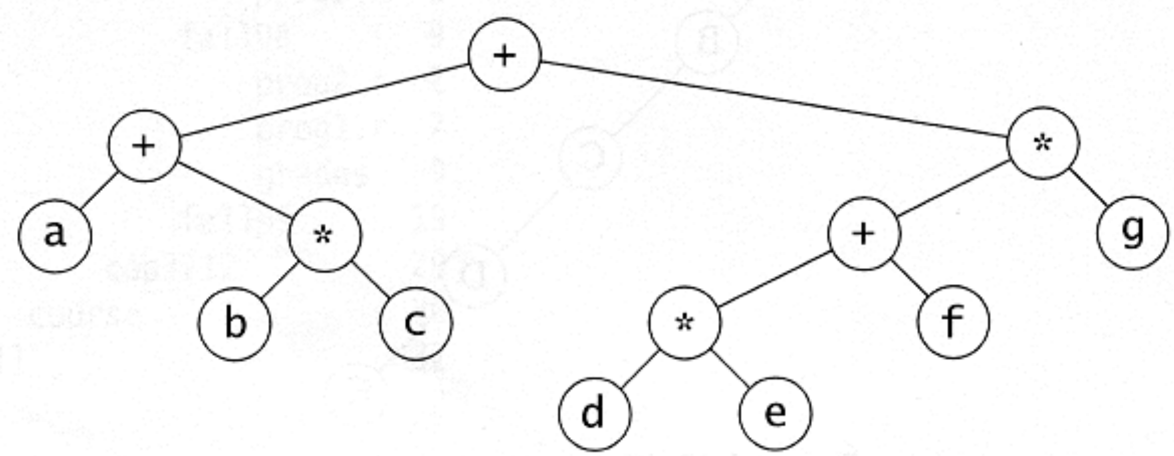


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Preorder Traversal: Example

- Preorder traversal
  - node, left, right
  - prefix expression
    - + + a * b c * + * d e f g

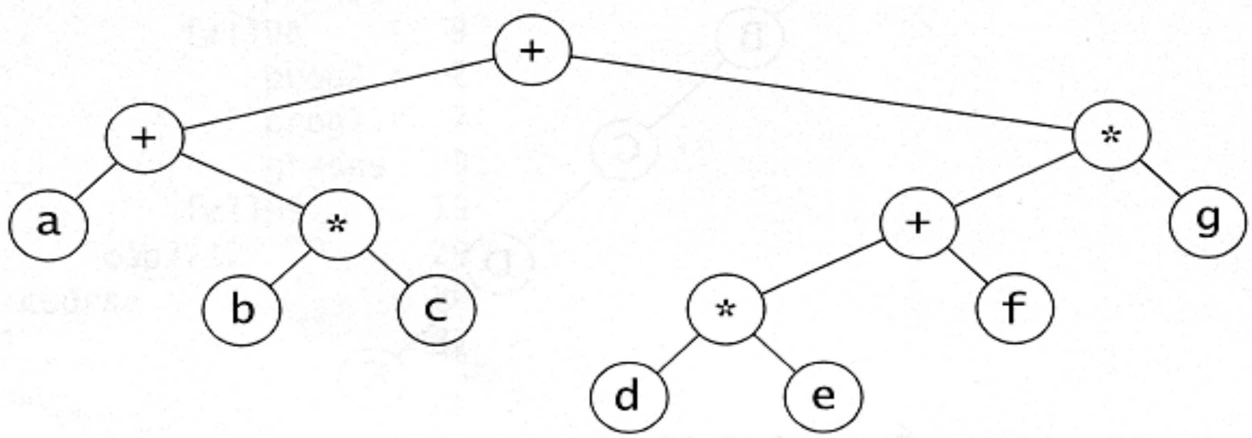


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Postorder Traversal: Example

- Postorder traversal
    - left, right, node
    - postfix expression
        - a b c * + d e * f + g * +

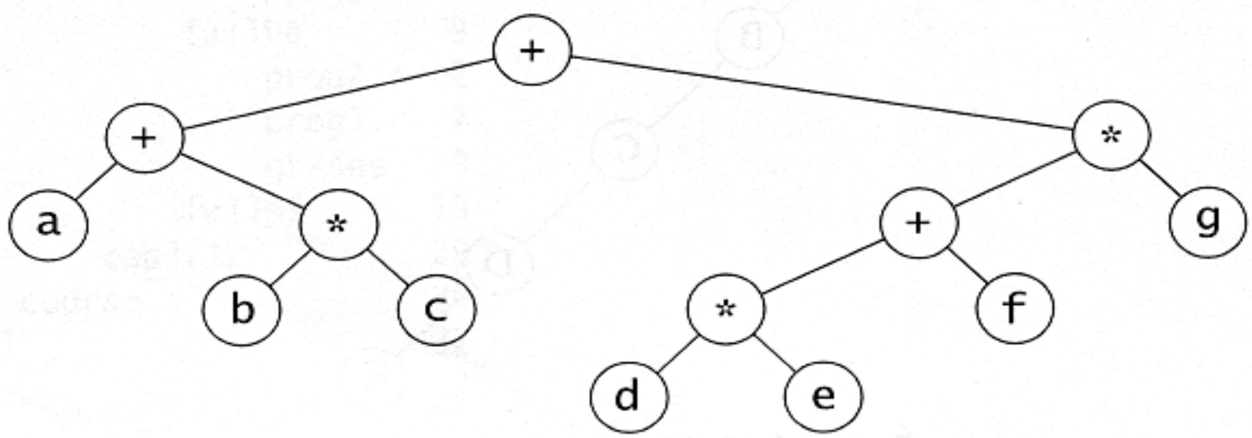
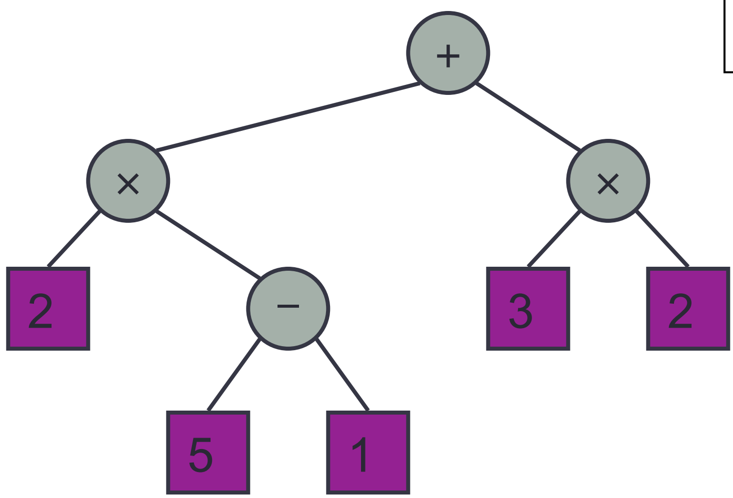


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

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

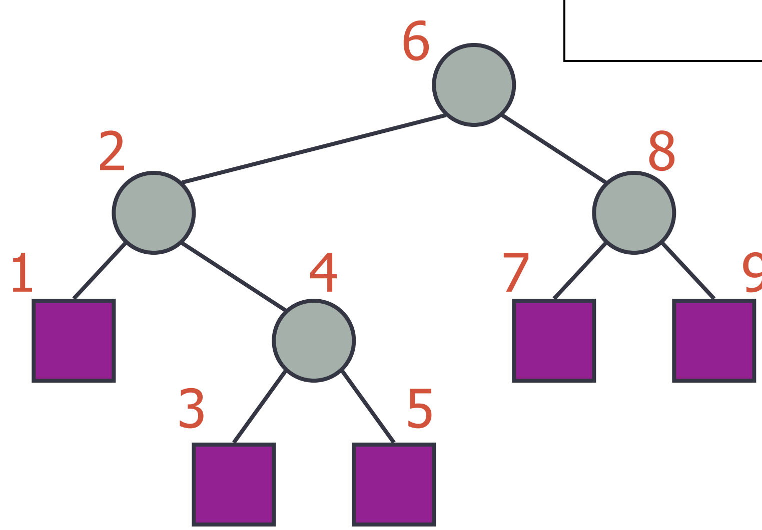
Algorithm *evalExpr*(v)
  if *isExternal* (v)
    return *v.element* ()
  else
    $x \leftarrow$ *evalExpr*(*left*(v))
    $y \leftarrow$ *evalExpr*(*right*(v))
    $\Diamond \leftarrow$ operator stored at v
    return $x \Diamond y$



Inorder Traversal (8.4.3)

- 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 *left* (*v*) $\neq$ **null**
        inOrder (*left* (*v*))
    visit(*v*)
    if *right*(*v*) $\neq$ **null**
        inOrder (*right* (*v*))



Inorder Traversal: Example

- Inorder traversal
  - left, node, right
  - infix expression
    - a + b * c + d * e + f * g

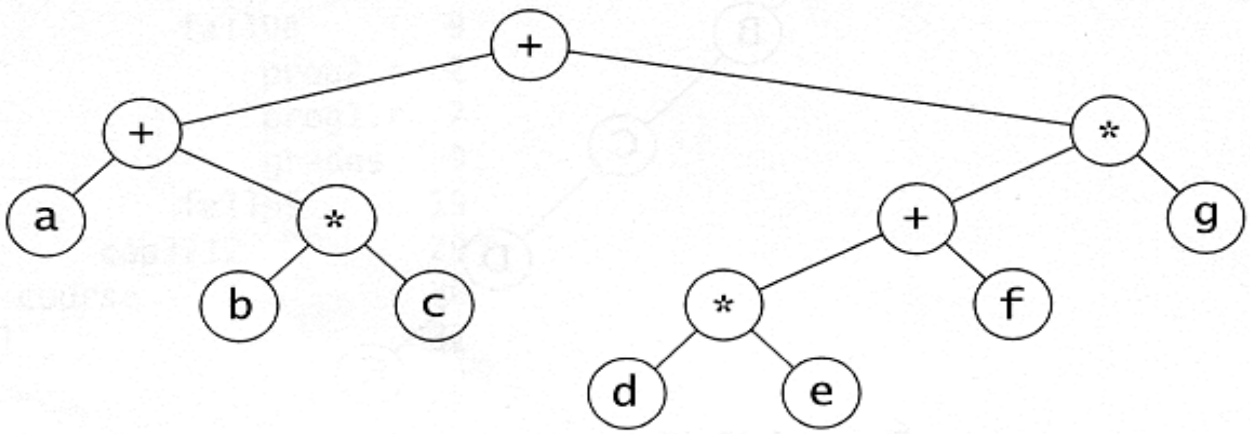
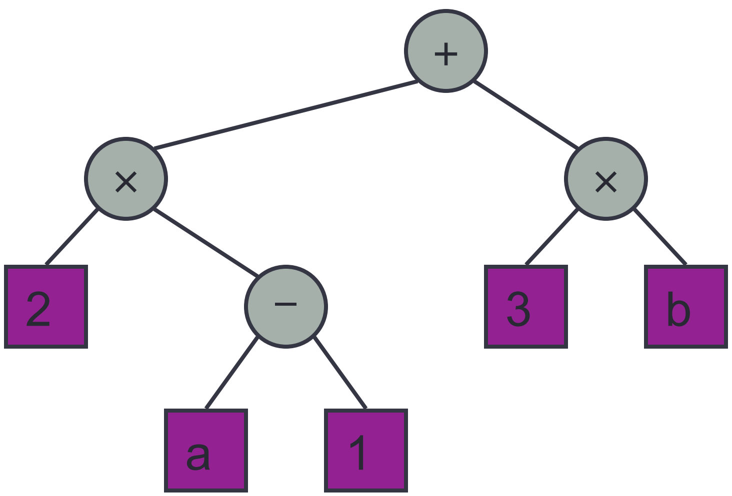


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

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 *left* (v) $\neq$ **null**
        print("(")
        inOrder (*left*(v))
    print(v.*element* ())
    if *right*(v) $\neq$ **null**
        inOrder (*right*(v))
        print (")")

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

Pseudo-code for Binary Tree Traversal

Algorithm *Preorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** output key(x);
3. *Preorder*(left(x));
4. *Preorder*(right(x));

Algorithm *Inorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** *Inorder*(left(x));
3. output key(x);
4. *Inorder*(right(x));

Algorithm *Postorder*(x)
Input: x is the root of a subtree.
1. **if** $x \neq$ NULL
2. **then** *Postorder*(left(x));
3. *Postorder*(right(x));
4. output key(x);

Properties of Proper Binary Trees

- A binary trees is ***proper*** if each node has either zero or two children.
- Level: depth

  The root is at level 0

  Level d has at most 2^d nodes
- Notation:

  n number of nodes

  e number of external (leaf) nodes

  i number of internal nodes

  h height

$$n = e + i$$

$$e = i + 1$$

$$h+1 \leq e \leq 2^h$$

$$n = 2e - 1$$

$$h \leq i \leq 2^h - 1$$

$$2h+1 \leq n \leq 2^{h+1} - 1$$

$$\log_2 e \leq h \leq e - 1$$

$$\log_2 (i + 1) \leq h \leq i$$

$$\log_2 (n + 1) - 1 \leq h \leq (n - 1)/2$$

Properties of (General) Binary Trees

- Level: depth
  The root is at level 0
  Level d has at most 2^d nodes
- Notation:
  n number of nodes
  e number of external (leaf) nodes
  i number of internal nodes
  h height

$$h+1 \le n \le 2^{h+1} - 1$$

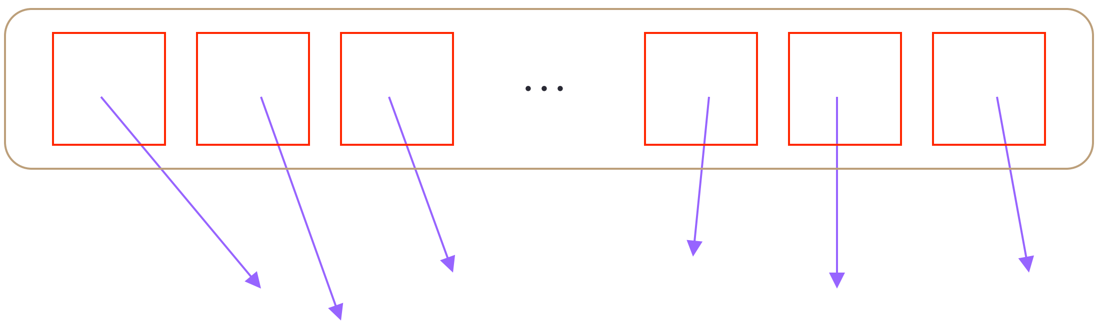
$$1 \le e \le 2^h$$

$$h \le i \le 2^h - 1$$

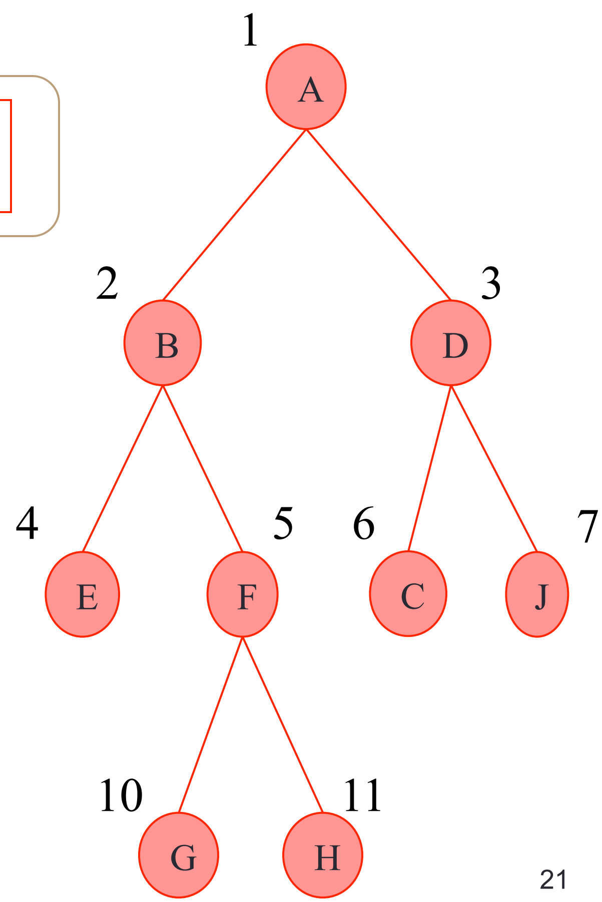
$$\log_2(n+1) - 1 \le h \le n - 1$$

Array-Based Implementation (8.3.2)

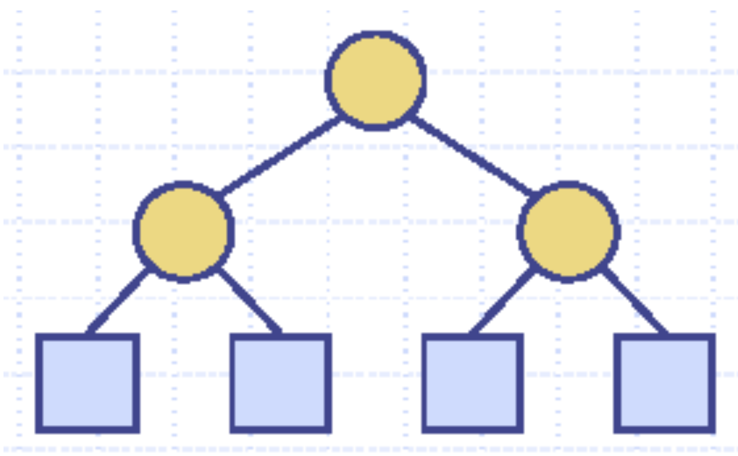
- Nodes are stored in an array.

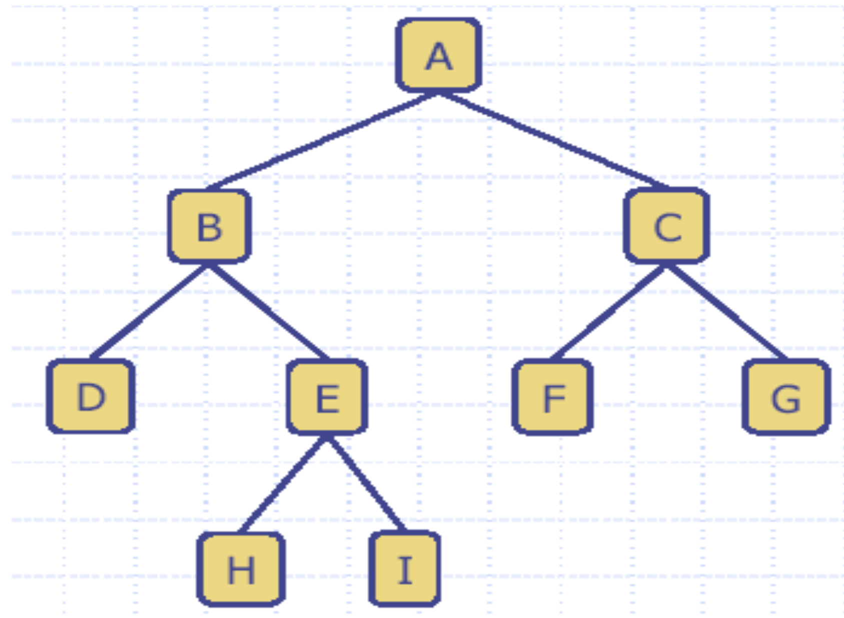


- Node v is stored at A[*rank*(v)]
- Let *rank*(v) be defined as follows:
  - *rank*(root) = 1
  - if v is the left child of parent(v),
    rank(v) = 2 x *rank*(parent(v))
  - if v is the right child of parent(v),
    rank(v) = 2 x *rank*(parent(v)) + 1



Array Implementation of Binary Trees





Each node *v* is stored at index *i* defined as follows:

- If *v* is the root, *i* = 1
- The left child of *v* is in position 2*i*
- The right child of *v* is in position 2*i* + 1
- The parent of *v* is in position ???

Space Analysis of Array Implementation

- n: number of nodes of binary tree T
- p_M: index of the rightmost leaf of the corresponding ***full*** binary tree (or size of the full tree)
- N: size of the array needed for storing T; $N = p_M + 1$

Best-case scenario: balanced, full binary tree $p_M = n$

Worst case scenario: unbalanced tree

- Height $h = n - 1$
- Size of the corresponding full tree:

  $p_M = 2^{h+1} - 1 = 2^n - 1$
- $N = 2^n$

Space usage: $O(2^n)$

Arrays versus Linked Structures

Linked structure

- Slower operations due to pointer manipulations
- Use less space if the tree is unbalanced
- AVL trees: rotation (restructuring) code is simple

Arrays

- Faster operations
- Use less space if the tree is balanced (no pointers)
- AVL trees: rotation (restructuring) code is complex

Next lecture …

- Binary Search Trees (11.1)