Algorithms and Data Structures

Trees

dr Szymon Murawski

Comarch SA

March 28, 2019

Table of contents I

Course plan

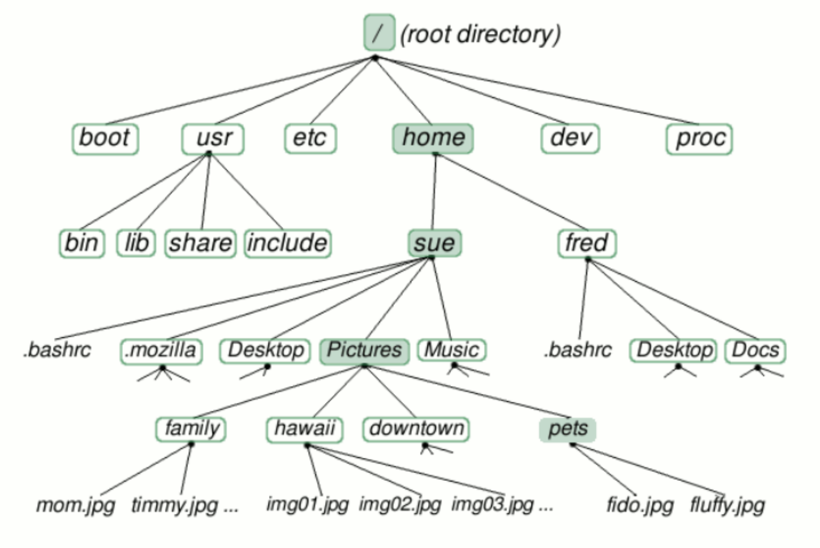
Tree data structure

- A tree is an abstract model of hierarchical structure
- Tree is the basic non-linear data structure
- A tree consists of nodes with a parent-child relation
- Each element (except the top element) has a parent and zero or more children elements

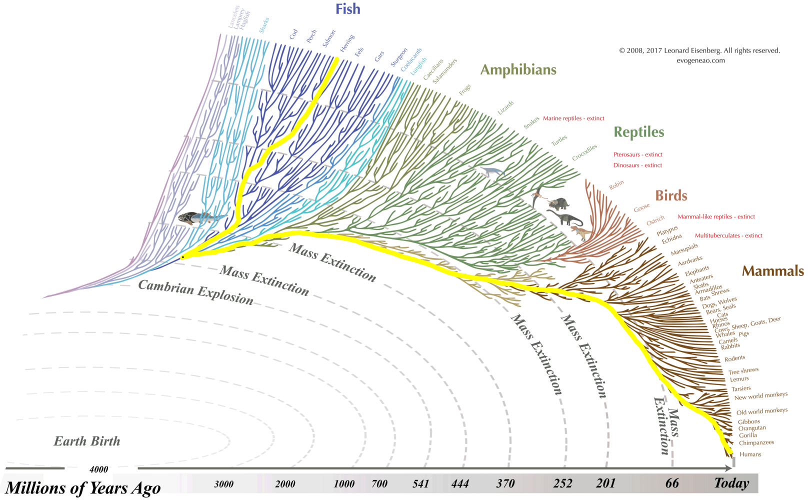
Recursive definition

Tree is either empty, or consists of node r and a set of trees (might be empty) whose roots are the children of r.

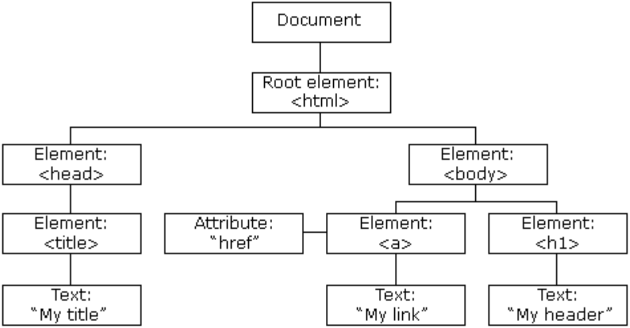
Unix file system



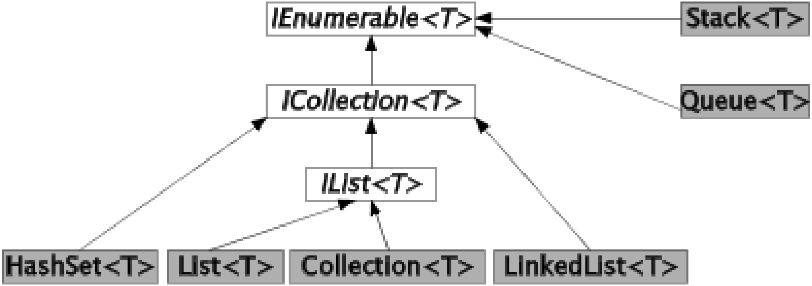
Phylogenetic tree



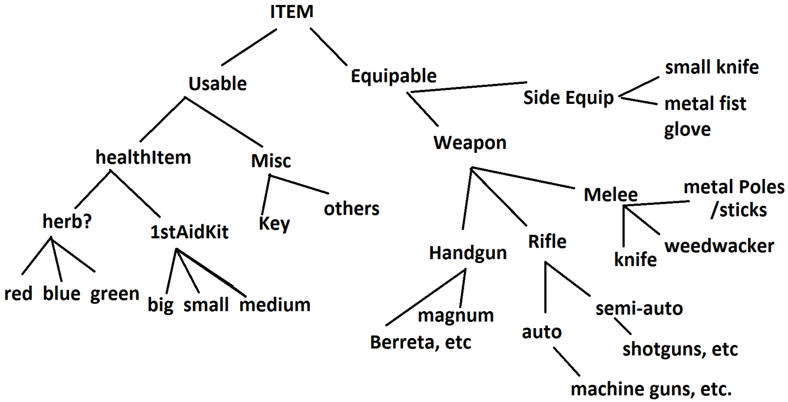
DOM tree


Document
Root element: <html>
Element: <head>
Element: <body>
Element: <title>
Attribute: "href"
Element: <a>
Element: <h1>
Text: "My title"
Text: "My link"
Text: "My header"

Inheritance tree



Game objects tree



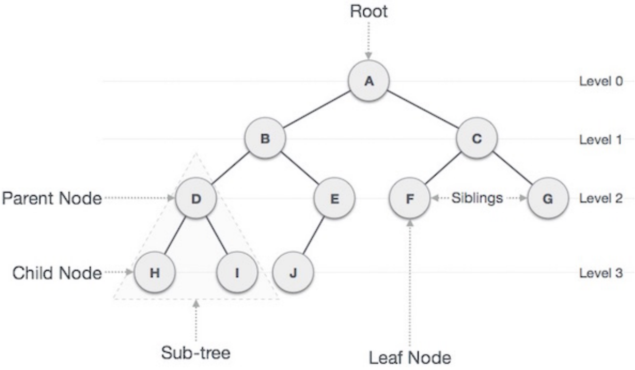
Course plan

1. Trees
   - Tree interface
   - Tree Traversal
2. Binary trees
3. Representation of trees
4. Arithmetic expression tree

Tree terminology

- Root - a node without any parent
- Internal node - a node with at least one child
- External node (leaf) - a node without children
- Subtree - tree consisting of a node and its descendants
- Ancestors of a node - parent, grandparent grand-grandparent, etc.
- Descendant of a node - child, grandchild, grand-grandchild, etc.
- Siblings - nodes, that have the same parent
- Depth of a node - distance from node to the root
- Height of a node - maximum distance from node to leaf

Tree terminology



Basic interface

- root() - returns a root of a tree
- size() - returns number of elements in a tree
- isEmpty() - checks whether a tree is empty
- parent(node) - returns a parent of specific node
- children(node) - returns a list of all children of specific node
- isInternal(node) - checks whether a specific node is an internal node
- isExternal(node) - checks whether a specific node is an external node

Depth and height

Depth

Depth of a specific node is the distance from this node to the root. Depth of a tree is the maximum depth of one of its leaves.

```
Depth(node)
  if node == root return 0
  else return 1 + depth(node.parent)
```

Height

Height of a node is the maximum distance from the node to one of the leaves. Height of the tree is the height of a root. Depth and height are symmetrical.

```
Height(node)
  if isExternal(node) return 0
  h = 0
  for each child of node do
    h = max(h, height(child))
  return h + 1
```

Course plan

1. Trees
   - Tree interface
   - Tree Traversal
2. Binary trees
3. Representation of trees
4. Arithmetic expression tree

Tree Traversal

- Tree traversal is a process of visiting each node of a tree
- Unlike linear data structures, there are many ways a tree can be traversed
- Breath-first search - we visit each node on the current level before going down one level. In other words we first visit siblings.
- Depth-first search - we go as deep as possible on the current branch. In other words we first visit children. There are many ways a DFS can be performed (for simplicity we assume that every node has only left and right child):
  - Pre-order (NLR) - process node N, then process Left subtree, then process Right subtree
  - In-order (LNR) - process Left subtree first, then node N, then Right subtree
  - Out-order (RNL) - process Right subtree, then node N, then Left subtree
  - Post-order (LRN) - process Left subtree, then Right subtree, then node N

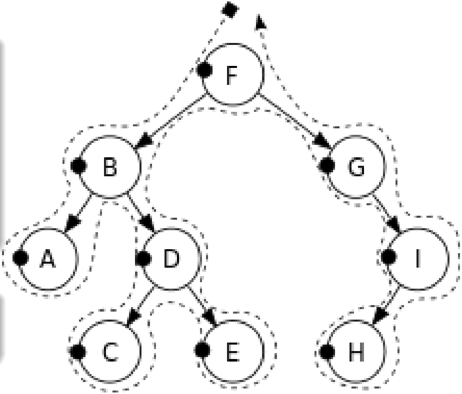
Pre-order traversal (NLR)

Pseudocode

```
Preorder(node)
   return if node == null
   print node.data
   Preorder(node.leftChild)
   Preorder(node.rightChild)
```

Node sequence

F, B, A, D, C, E, G, I, H



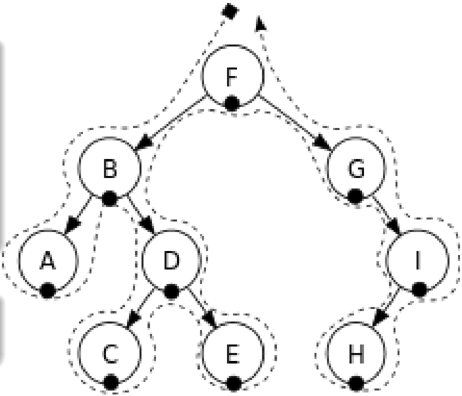
In-order traversal (LNR)

Pseudocode

```
Inorder(node)
   return if node == null
   Inorder(node.leftChild)
   print node.data
   Inorder(node.rightChild)
```

Node sequence

A, B, C, D, E ,F, G, H, I



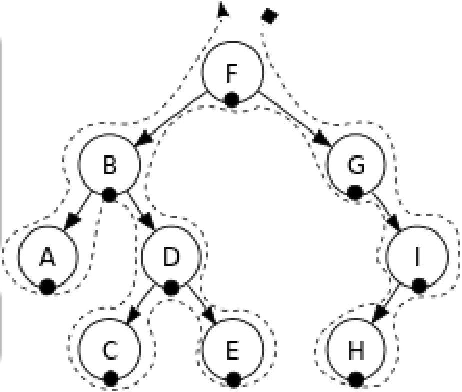
Out-order traversal (RNL)

Pseudocode

```
Outorder(node)
   return if node == null
   Outorder(node.rightChild)
   print node.data
   Outorder(node.leftChild)
```

Node sequence

I, H, G, F, E, D, C, B, A



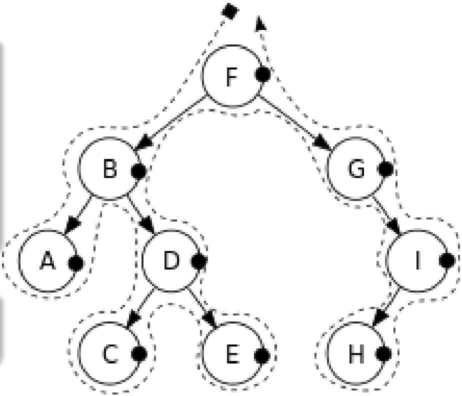
Post-order traversal (LRN)

Pseudocode

```
Postorder(node)
  return if node == null
  Postorder(node.leftChild)
  Postorder(node.rightChild)
  print node.data
```

Node sequence

A, C, E, D, B, H, I, G, F



Course plan

Binary tree

Definition

- A binary tree is a tree in which each node has at most two children
- We name the children left and right

Recursive definition

- Tree is either empty
- Or it consists of
  - a root
  - a binary tree called left subtree
  - a binary tree called the right subtree

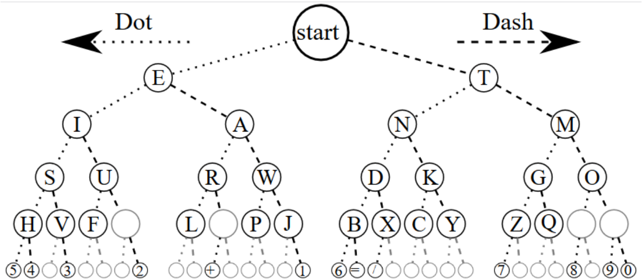
Additional functions

- left(node) - returns left child of a node
- right(node) - returns right child of a node
- hasLeft(node) - checks whether a node has left child
- hasRight(node) - checks whether a node has right child

Binary trees application

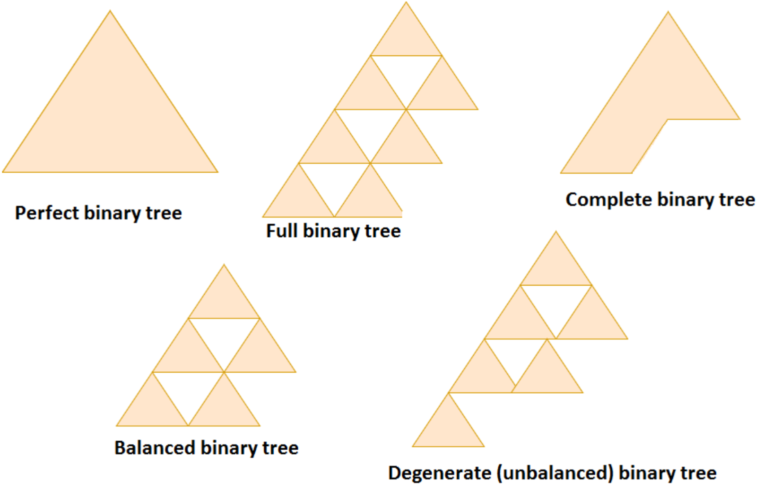
- Binary trees are very popular and widely used data structure
- They are less complex than n-ary trees (ternary, quaternary etc.) and provide comparable speed of algorithms
- Some applications:
  - Binary search trees (BST) - used in applications when data is constantly being added/removed
  - Binary space partition (BSP) - used in 3D games to determine which objects needs to be rendered
  - Heaps - used for one of the best sorting algorithms and priority queues
  - Huffman coding - efficient and lossless data compression algorithm
  - Syntax tree - used by compilers and calculators to parse expressions
  - Decision trees - any form of a problem that is a series of yes/no questions can be represented as binary tree

Morse code



Types of binary trees

- Rooted binary tree has a root node and every node has at most two children
- Full binary tree is a tree in which every node has either zero or two children
- Complete binary tree is a tree in which every level, except possibly the last, is completely filled and all nodes in the last level are as far left as possible
- Perfect binary is a tree in which every interior node has exactly two children and each leaf has the same depth
- Balanced binary tree is a tree in which the left and right subtrees of every node differ in height by no more than 1
- Degenerate tree is a tree in which every parent has at most one child, thus making the tree a linked list


Perfect binary tree
Full binary tree
Complete binary tree
Balanced binary tree
Degenerate (unbalanced) binary tree

Binary trees properties

The maximum number of nodes n_{max} at level 'l' of a binary tree is 2^l

- Level is a number of nodes from root to the node
- Level of root is 0
- Proof by induction:
  - For root, $l = 0$, $n_{max} = 2^0 = 1$
  - Assume that on level l $n_{max} = 2^l$, then on next level $l + 1$ $n_{max} = 2^{l+1}$
  - Since in binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^l = 2^{l+1}$

Maximum number of nodes in a binary tree of height 'h' is $2^{h+1} - 1$

- Derived from previous point
- A tree has maximum nodes if all levels have maximum nodes
- So for height h $n_{max} = 1 + 2 + 4 + \cdots + 2^h = \sum_{k=0}^{h} 2^k = 2^{h+1} - 1$

Binary trees properties

Minimum number of nodes in a binary tree of height h is $h + 1$

- Minimum number of nodes at every level is equal to 1, meaning that every node has exactly one child
- Because of the above, minimum number of nodes must be equal to number of levels plus one for the root, $n_{min} \geq h + 1$
- Combining two above properties we arrive at $h + 1 \leq n \leq 2^{h+1} - 1$

A binary tree with L leaves has at least $\lceil lg\ L \rceil$ levels

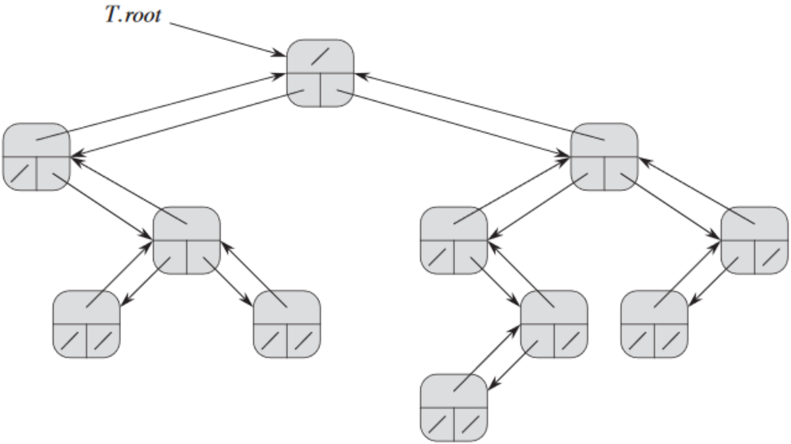
- A binary tree has maximum number of leaves (and minimum number of levels) when all levels are fully filled
- Let all leaves be at level l, then:
- $L \leq 2^l \rightarrow lg\ L \leq l \rightarrow l = \lceil lg\ L \rceil$

Course plan

Linked list representation of binary tree

- In linked list representation of a binary tree every list element is also a node of a tree
- We use triple-linked list data structure to represent relations between nodes
- Every node is composed of the following fields:
  - data - data that the node carries
  - parent - pointer to parent of a node
  - left - pointer to left child of a node
  - right - pointer to right child of a node
- If `parent` is equal to NULL, then the node is the root node
- If both left and right point to NULL, then the node is a leaf
- In any other situation the node is an internal node
- Parent field can be omitted if memory is an issue, but it will increase complexity of some operations

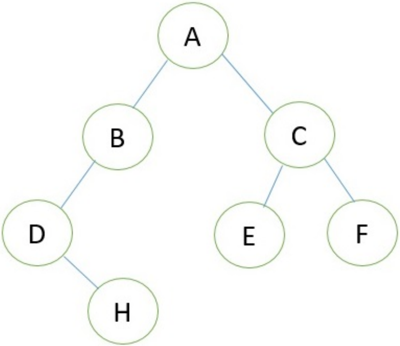
Linked list representation of binary tree



Single array representation of binary tree

- We use simple array as underlying data structure for storing information about nodes
- Array is constructed by reading tree level by level and inserting into array value of a node. If node does not exist we insert NULL
- If the tree is not complete then a lot of memory goes to waste
- We cannot easily resize the array in case additional nodes are to be added to a tree

Single array representation of binary tree



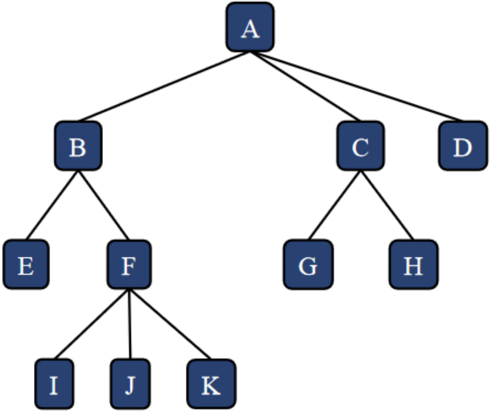
A	B	C	D	-	E	F	-	H		

Parallel arrays representation

- We use two parallel arrays to represent a tree
- Index of each array corresponds to specific node
- First array holds information about the data of node
- Second array stores index of parent node
- We cannot mess up order of the arrays!
- Instead of arrays we can use lists, to dynamically resize the structure

Parallel arrays representation

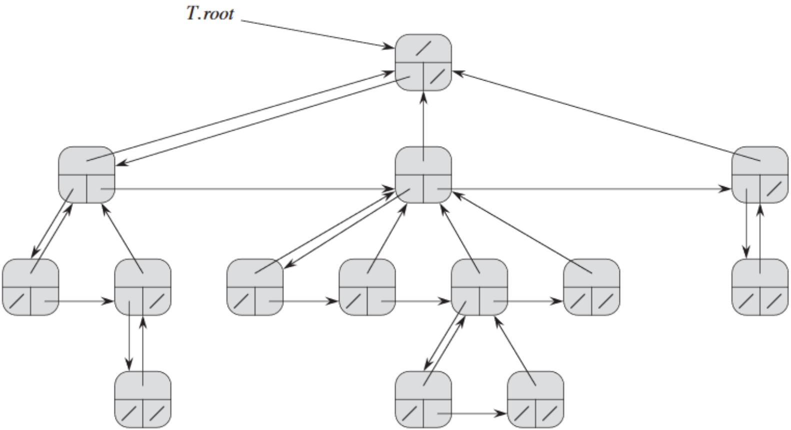
index	data	parent
0	A	0
1	B	0
2	C	0
3	D	0
4	E	1
5	F	1
6	G	2
7	H	2
8	I	5
9	J	5
10	K	5



Siblings representation

- In this representation Every node has only two fields:
  - leftChild - pointer to the left child of a node
  - rightSibling - pointer to the right sibling of a node
  - parent - pointer to parent of a node
- This representation works for any n-ary tree
- It uses only O(n) memory for any n-node tree
- We do not have to deal with multiple arrays or lists for each of the nodes

Siblings representation



Course plan

Arithmetic expression tree

- Arithmetic expression tree is a tree, where every leaf is an operand (value) and every internal node is an operator
- Every leaf is a single operand
- Every internal node is a single binary operator
- The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree
- The levels of the nodes indicate their relative precedence of evaluation - we do not need parenthesis to indicate precedence
- This technique is used in modern calculators
- We can build the tree using postfix notation (just as in RPN calculator)
- Depending on traversal method we can arrive at different results

Arithmetic expression tree traversal

Post-order (LRN)

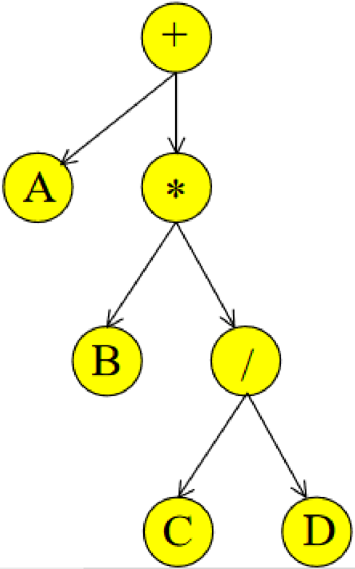
- We get postfix notation
- Sequence obtained: $ABCD/*+$

In-order (LNR)

- We get infix notation
- Sequence obtained: $A+B*C/D$

Pre-order (NLR)

- We get prefix notation
- Sequence obtained: $+A*B/CD$



Constructing expression tree

- We construct expression tree from postfix notation
- We already know an algorithm that turns infix notation into postfix (recall our stack lecture)

```
postfixToTree(string expression)
  create new stack
  foreach input in expression
    if input is operand
      tree = new tree(input)
      stack.push(tree)
    else if input is operator
      tree2 = stack.pop()
      tree1 = stack.pop()
      tree = new tree(input)
      tree.addLeft(input, tree1)
      tree.addRight(input, tree2)
      stack.push(tree)
  return stack.pop()
```

Printing infix expression from a tree

- We use inorder traversal (left child, node, right child)
- We print operand or operator when visiting node
- We print "(" before traversing left subtree
- We print ")" after traversing right subtree

```
printExpression(node)
  if hasLeft(node)
    print("(")
    printExpression(node.leftChild)
  print(node.data)
  if hasRight(node)
    printExpression(node.rightChild)
    print(")")
```

Evaluating expression from an expression tree

- We use postorder traversal method (first children, then node)
- When we are visiting a leaf, we return the value stored in leaf
- When visiting an internal node, we combine the values of left and right subrees using operator stored in node

```
evalExpression(node)
  if isExternal(node)
    return node.data
  else
    x = evalExpression(node.leftChild)
    y = evalExpression(node.rightChild)
    operator = node.data
    return eval(x operator y)
```