UNIVERSITY OF WATERLOO

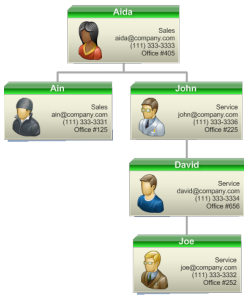
David R. Cheriton School of Computer Science

CS 115: Introduction to Computer Science

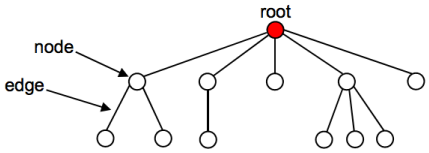
Presented By Ahmed Ibrahim

Fall 2015

Company Organization

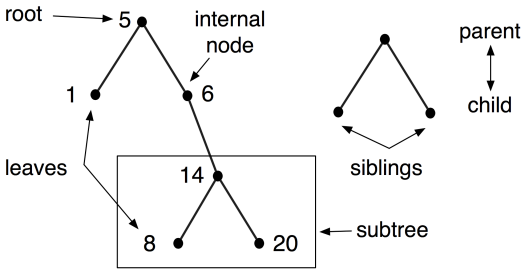


What is Trees

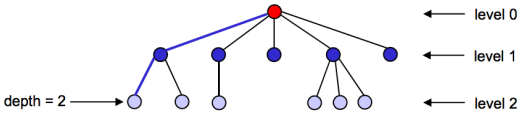


A tree consists of:
- a set of nodes
- a set of edges, each of which connects a pair of nodes

Tree terminology



Path, Depth, Level, and Height



- There is exactly one path (one sequence of edges) connecting each node to the root.
- **depth** of a node = # of edges on the path from it to the root.
- Nodes with the same depth form a **level** of the tree.
- The **height** of a node is the number of edges from the node to the deepest leaf.
- The **height** of a tree is the maximum depth of its nodes.

Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

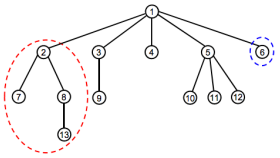
- Trees reflect **structural relationships** in the data.
- Trees are used to represent **hierarchies.**
- Trees provide an efficient insertion and searching.

Variations on trees

- Number of children of internal nodes:
  - at most two
  - exactly two
  - any number
- Labels:
  - on all nodes
  - just on leaves
- Order of children (matters or not)
- Tree structure (from data or for convenience)

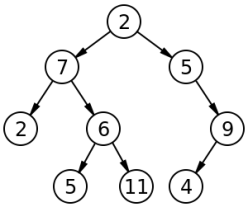
A Tree is a Recursive Data Structure

- Each node in the tree is the root of a smaller tree!
  - refer to such trees as sub-trees to distinguish them from the tree as a whole
  - example: node 2 is the root of the sub-tree circled above
  - example: node 3 is the root of a sub-tree with only one node
- We'll see that tree algorithms often lend themselves to recursive implementations.



Binary Tree (BT)

- In computer science, a **binary tree** is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child.



Binary arithmetic expressions

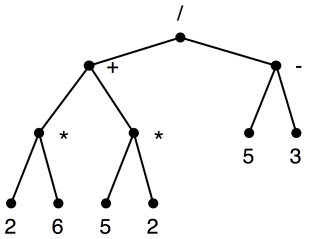
- A binary arithmetic expression is made up of numbers joined by binary operations ∗, +, /, and −.
- ((2 ∗ 6) + (5 ∗ 2))/(5 − 3) can be defined in terms of *two* smaller binary arithmetic expressions, (2 ∗ 6) + (5 ∗ 2) and 5 − 3.
- Each smaller expression can be defined in terms of even smaller expressions.
- **The smallest expressions are numbers**.

Representing binary arithmetic expressions

- Internal nodes each have **exactly two children**.
- Leaves have **number labels**.
- Internal nodes have **symbol labels**.
- For subtraction and division, we care about the order of children.
- The structure of the tree is dictated by the expression.
- **Rules**:
  - Rule1: Operators can have children but operands can't.
  - Rule2: Nodes can only have 2 children.

Visualizing binary arithmetic expressions

((2 ∗ 6) + (5 ∗ 2))/(5 − 3) can be represented as a **tree**:



Representing binary arithmetic expressions (cont.)

- How can we group together information for an internal node?
- How can we allow different definitions for leaves and internal nodes?

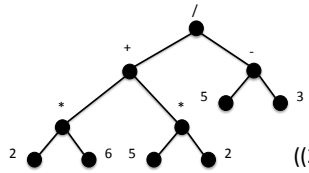
Node structure

```
(define-struct binode (op arg1 arg2))
;; A Binary arithmetic expression Internal Node (BINode)
;; is a (make-binode (anyof '* '+ '/ '-) BinExp BinExp)

;; A Binary arithmetic expression (BinExp) is one of:
;; * a Num
;; * a BINode
;; Examples
5
(make-binode '* 2 6)
(make-binode '+ 2 (make-binode '- 5 3))
```


A more complex example

```
(make-binode '/
        (make-binode '+
                (make-binode '* 2 6)
                (make-binode '* 5 2))
        (make-binode '- 5 3))
```



((2 * 6) + (5 * 2))/(5 – 3)

Template for binary arithmetic expressions

The only new idea in forming the template is the application of the recursive function to *each* piece that satisfies the data definition.

```
;; my-binexp-fun: BinExp → Any
;; (define (my-binexp-fun ex)
;;   (cond
;;     [(number? ex) . . . ]
;;     [else . . . (binode-op ex) . . .
;;           . . . (my-binexp-fun (binode-arg1 ex)) . . .
;;           . . . (my-binexp-fun (binode-arg2 ex)) . . . ]))
```

```
;; A Binary arithmetic
;; expression (BinExp) is one of:
;; * a Num
;; * a BINode
```

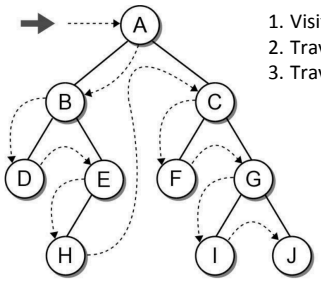
Evaluation of expressions

```
(define (eval ex)
    (cond
        [(number? ex) ex]
        [else
            (cond
              [(symbol=? (binode-op ex) '*)
                 (* (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))]
              [(symbol=? (binode-op ex) '+)
                 (+ (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))]
              [(symbol=? (binode-op ex) '/)
                 (/ (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))]
              [(symbol=? (binode-op ex) '-)
                 (- (eval (binode-arg1 ex)) (eval (binode-arg2 ex)))])))))
```

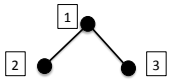

Traversals

- A traversal is a process that visits all the nodes in the tree.
  - breadth-first traversal
  - depth-first traversal
- There are three different types of depth-first traversals:
  - PreOrder traversal - visit the parent first and then left and right children;
  - InOrder traversal - visit the left child, then the parent and the right child;
  - PostOrder traversal - visit left child, then the right child and then the parent;

Pre-order Traversal

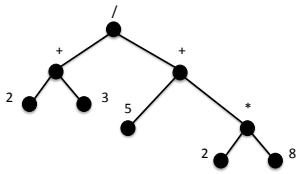


1. Visit the root node.
2. Traverse the left sub-tree.
3. Traverse the right sub-tree.



Ref.: Data Structures and Algorithms Using Python, by Rance D. Necaise.

Pre-order Traversal Example

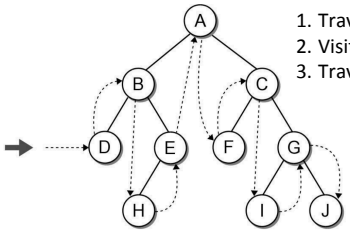


/ + 2 3 + 5 * 2 8

(/ (+ 2 3) (+ 5 (* 2 8)))

In-order Traversal

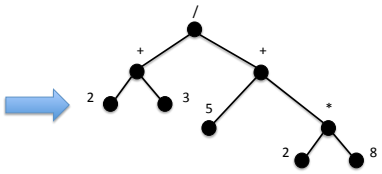
Similar to the preorder traversal, but we traverse the left sub-tree before visiting the root.



1. Traverse the left sub-tree.
2. Visit the root node.
3. Traverse the right sub-tree.

Ref.: Data Structures and Algorithms Using Python, by Rance D. Necaise.

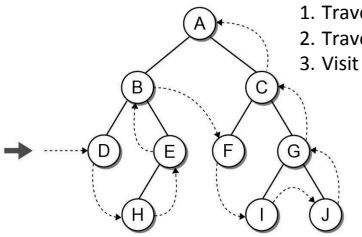
In-order Traversal Example



2 + 3 / 5 + 2 * 8

(2 + 3) / (5 + (2 * 8))

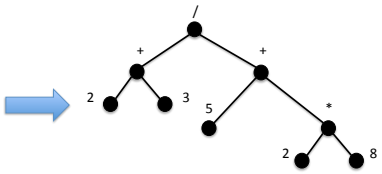
Post-order Traversal



1. Traverse the left sub-tree.
2. Traverse the right sub-tree.
3. Visit the root node.

Ref.: Data Structures and Algorithms Using Python, by Rance D. Necaise.

Post-order Traversal Example



2 3 + 5 2 8 * + /

(2 3 +) (5 (2 8 *) + /