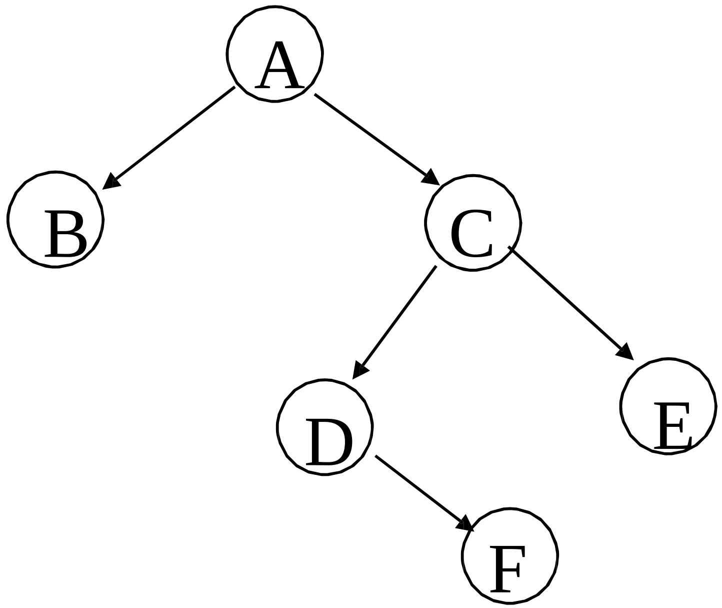
SCC120 Fundamentals of Computer Science
Unit 8: Trees (Traversals)

Jidong Yuan
yuanjd@bjtu.edu.cn

Tree Traversal

- Traversal of trees is simpler than traversal of graphs
  - There are no loops, so there is no need to mark nodes as they are visited
- We can do breadth-first and depth-first traversal

Breadth-First Traversal

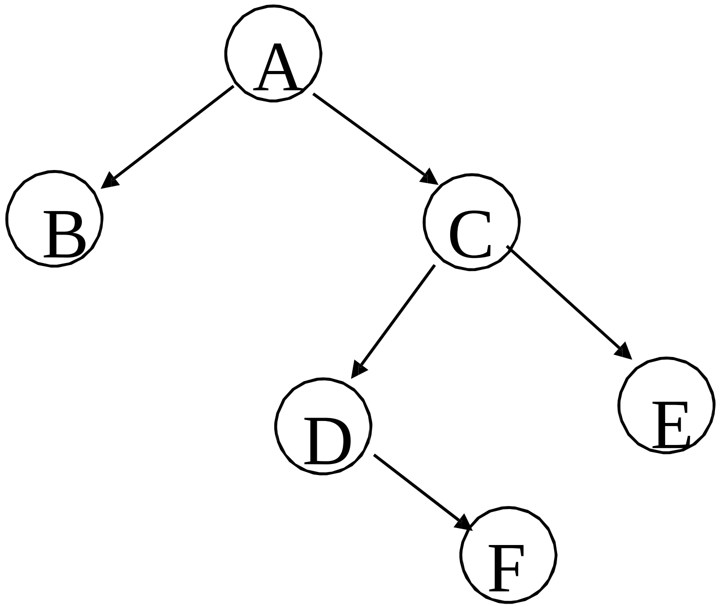


- Normally “away from the root”
  - Order of nodes visited: A, B, C, D, E, F
- Alternatively “towards the root”
  - Order of nodes visited: F, D, E, B, C, A

Depth-First Traversal

- Three types of these traversals:
  - Preorder
  - Inorder
  - Postorder

Preorder Traversal or “Prefix Walk”

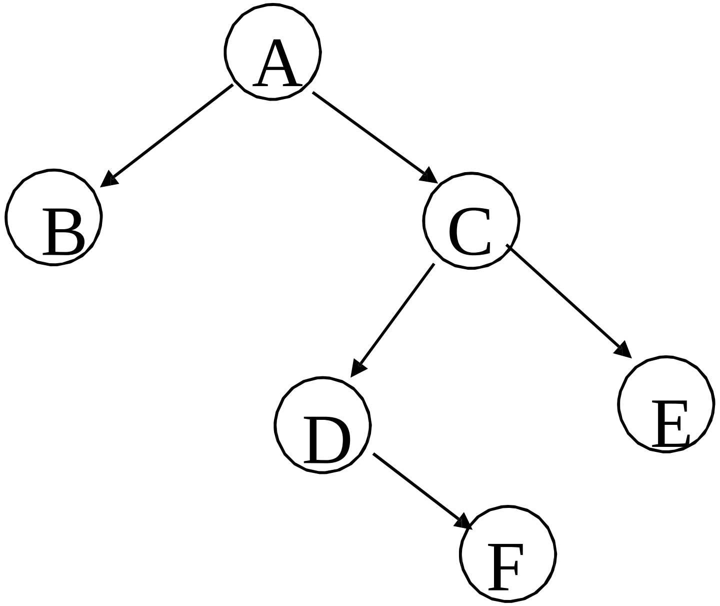


- Visit the “node” first, before visiting each of its subtrees in turn (first subtree then second subtree, and so on)
- This is the natural order for searching
- Order is: A, B, C, D, F, E

Order of Subtrees

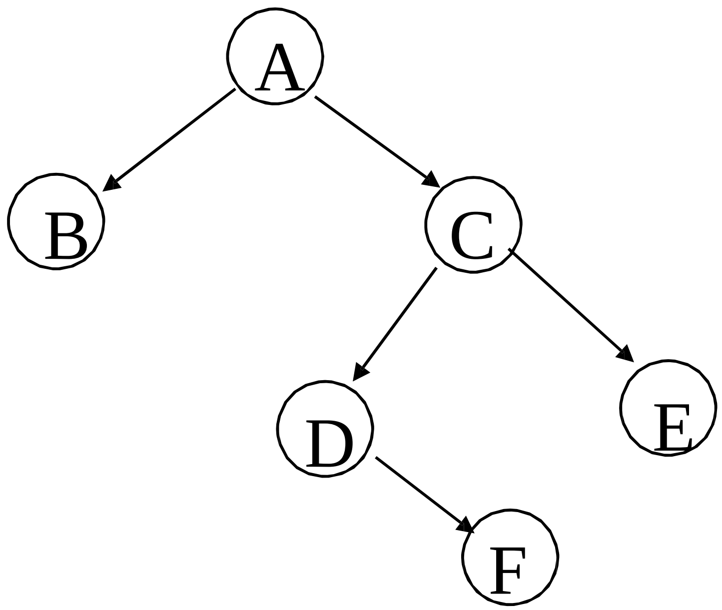
- These traversals make sense for some ordering of the subtrees:
  - 1st subtree, 2nd subtree, …
  - Left subtree, Right subtree
- For oriented binary trees and preorder traversal:
  - Visit the node
  - Then visit the left subtree (if any)
  - Then visit the right subtree (if any)

Inorder Traversal or “Infix Walk”



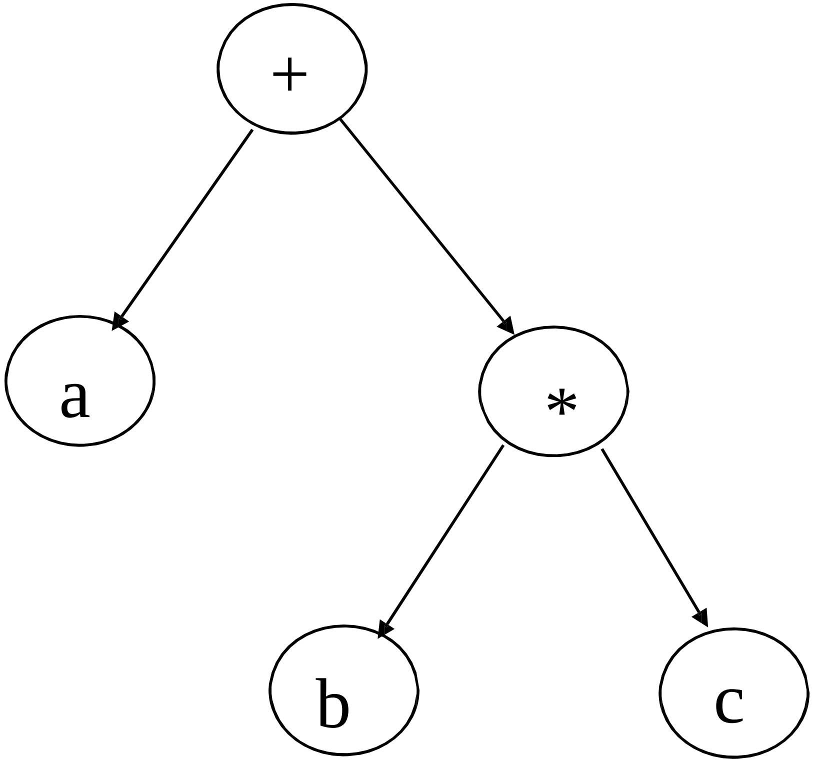
- Visit the node’s first subtree, then the node, then the second subtree
- Order is: B, A, D, F, C, E

Postorder Traversal or “Suffix Walk”



- Visit each of the node’s subtrees in turn, and finally the node itself at the end
- Order is: B, F, D, E, C, A

Evaluating Arithmetic Expression with Postorder Traversal



- If we wanted to evaluate this expression (from the computer's point of view), postorder traversal is the right way
- Postorder traversal gives us: a b c * +

Evaluating Arithmetic Expression with Postorder Traversal

- Postorder traversal gives us: a b c * +
- Suppose the variable “a” is 6, “b” is 3, and “c” is 4
- The computer goes through these steps:
  - Extract a (=6)
  - Extract b (=3)
  - Extract c (=4)
  - Multiply last two elements (3*4=12)
  - Add last two elements (6+12=18)
  - Result is 18

Evaluating Arithmetic Expression with Postorder Traversal

- In the steps (on previous slide), we can use a stack to hold the intermediate results
  - How?

Preorder and Postorder Traversal Pseudocode

```
public void depthFirstTraversal(Tree T, Node N)
{
    visitNode(N) here for preorder traversal
    for each child node X attached to N
        depthFirstTraversal(T, X);
    //visitNode(N) here for postorder traversal
}
```

InOrder Traversal Pseudocode for a Binary Tree

```
public void inOrder(BinaryTree B, Node N)
{
    if N has a left child node L
        inOrder(B, L);
    visitNode(N);
    if N has a right child node R
        inOrder(B, R);
}
```

Comments on These Traversals

- The above code examples are *recursive*
- We can use a *runtime stack* to record where we are in the pattern of “methods calling methods”
- They work because the recursive calls always operate on a **smaller (sub)tree** than the previous call

Summary of Traversals for Arithmetic Expressions

- A preorder traversal gives +a*-b4c

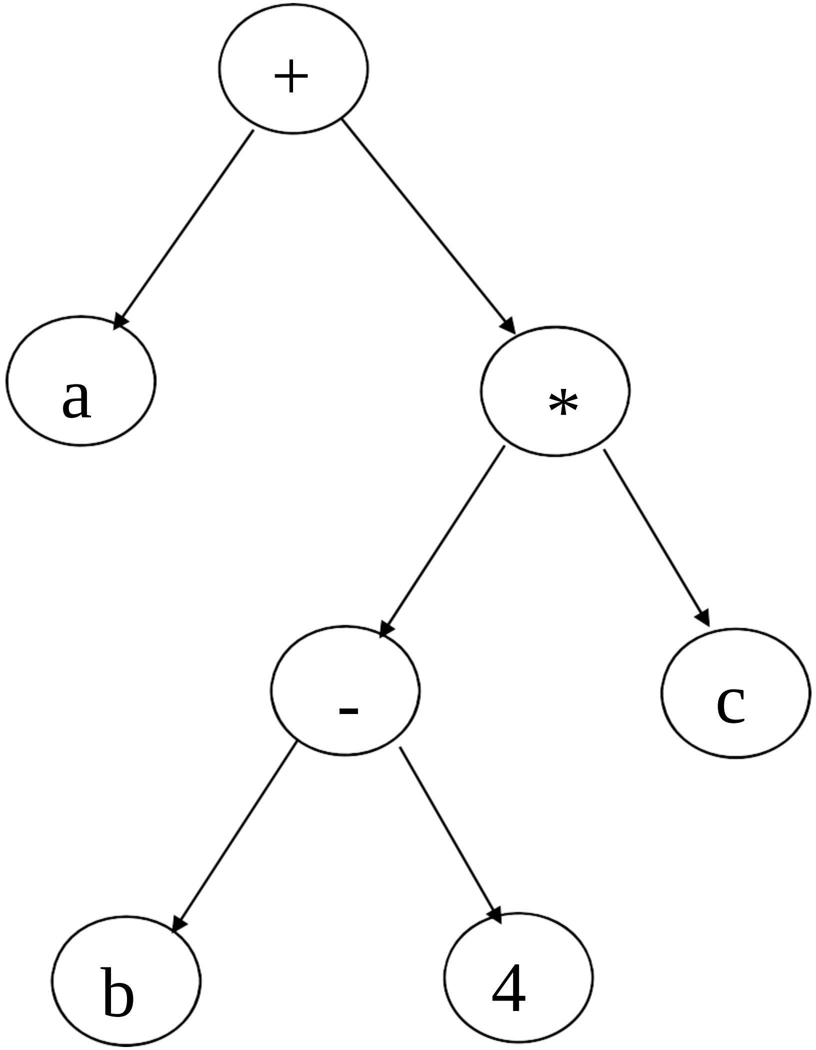
  This is called the *prefix notation* for an arithmetic expression - it is unambiguous

- A postorder traversal gives ab4-c*+

  This is called the *postfix notation* (or Reverse Polish) - it is also unambiguous

- An inorder traversal gives a+b-4*c

  This is the traditional *infix* notation



Precedence and Brackets

- The problem is that a+b-4*c by itself is ambiguous

  Is the last part b-(4*c) or (b-4)*c ?

- We have *rules of precedence* to say that

  * and / bind more tightly than + and -

  * and / have *higher precedence* than + and -

- If we want a different order or evaluation, we introduce brackets

  So here we can write a+(b-4)*c

Conversion Steps for a Computer

- The problem a computer/compiler has to deal with is to go from
  - the infix notation of an arithmetic expression
  - plus the rules of precedence
  - plus any brackets inserted by the programmer
- to the tree form of the expression
- and from there to do a postorder traversal
  - to evaluate the expression (or to generate code to evaluate the expression) in the right order

More on Precedence

- We need rules of precedence for *all* operators for when they appear without explicit bracketing
  - We need to consider *binary operators* with two operands like + and /
  - But also *unary operators* with just one operand such as “not”
  - Minus can be a binary operator as in (a-b) or a unary operator as in (a*-b)

Operator Precedence in Java

Level 1 ++ --

Level 2 unary + unary - !

Level 3 * / %

Level 4 + -

Level 5 < <= > >=

Level 6 == !=

Level 7 &

Level 8 |

Level 9 &&

Level 10 ||

Level 11 = += -= *= /= %=

Besides Traversals, Typical Operations on Trees include

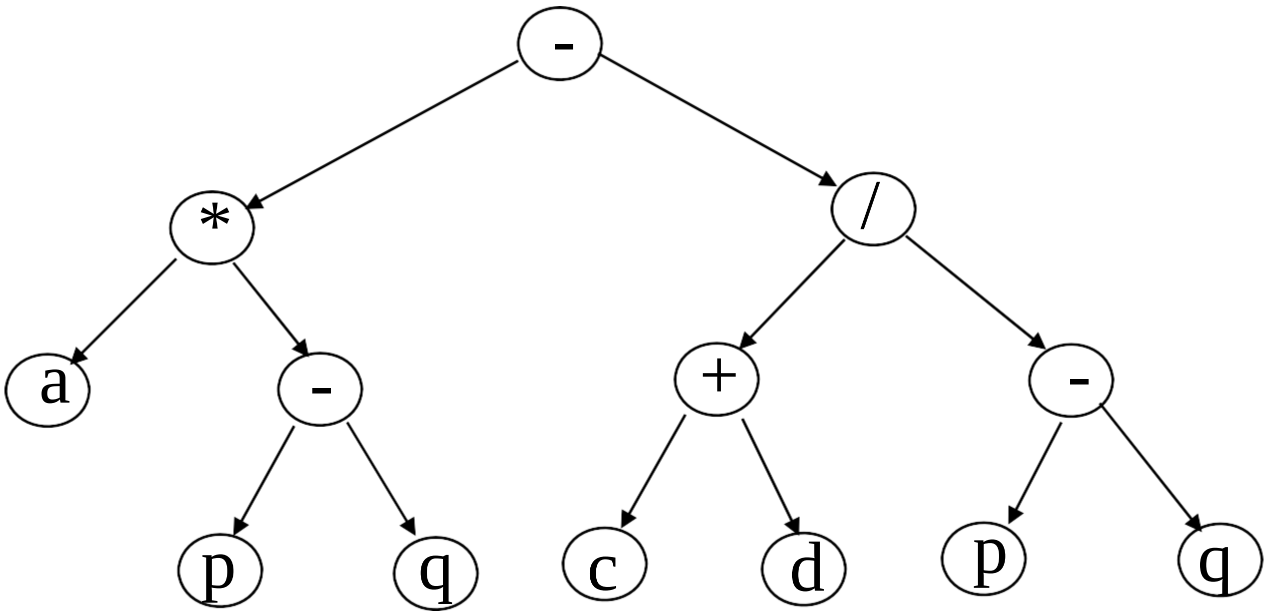
- Adding nodes or subtrees
  - How can you add a node?
- Deleting nodes or subtrees
  - This may be complicated if you have to maintain the rest of the tree (e.g. keep it balanced)
- Re-arranging the tree
  - E.g. moving nodes to make a balanced tree
  - Converting a non-binary tree to binary form (generally to make it simpler to implement)
- Updating node values

Besides Traversals, Typical Operations on Trees include

- Searching and traversing
  - Usually starting from the root
- Accessing a node
  - Usually starting from the root

Saving Space: Expressions and Trees

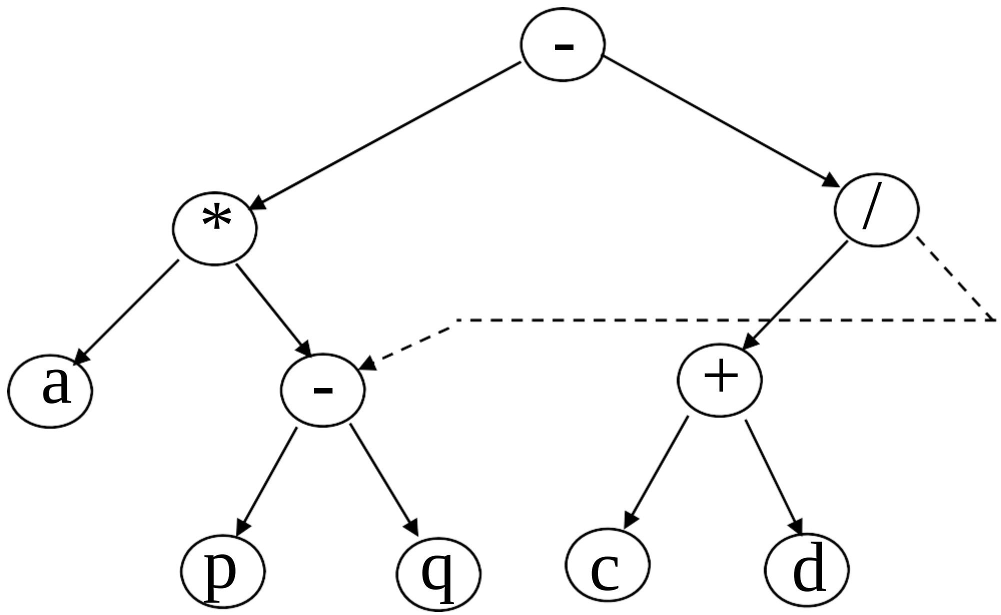
- An example arithmetic expression: a*(p-q)-(c+d)/(p-q)
- This gives a tree:



- *Saving space*: The computer could recognise that (p-q) is repeated, and can be stored and evaluated only once

Saving Space: Expressions and Trees

- We can draw it this way (which fits with the arithmetic expression):



- But then we get a graph rather than a tree (so there are tradeoffs)

SCC120 ADT (Weeks 7-13)

- Week 7 — Abstractions; Set
  Stack
- Week 8 — Queues
  Priority Queues
- Weeks 9-10 — Graphs (Terminology)
  Graphs (Traversals)
  Graphs (Representations)
- Week 12 — Trees (Terminology)
  Trees (Traversals)
- Week 13