CSC148H Lecture 4

Dan Zingaro
OISE/UT

September 29, 2008

Motivating Trees

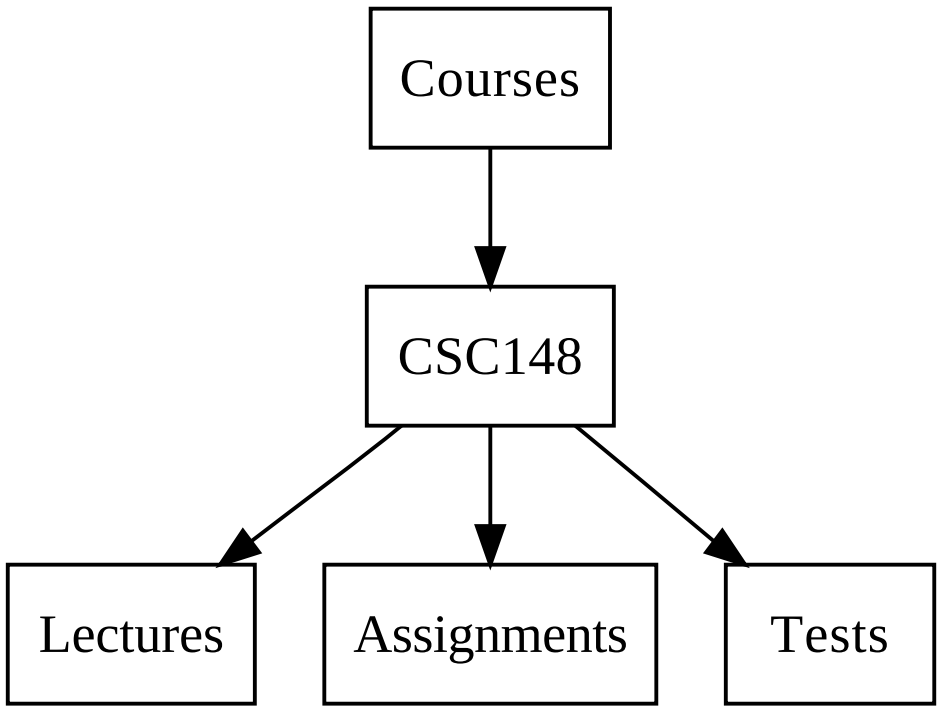
- A data structure is a way of organizing data
- Stacks, queues, lists, dequeues, and naive priority queues are all linear structures
- They are linear in the sense that data is ordered (i.e. first piece of data is followed by second is followed by third ...)
- This makes sense for many applications:
  - A lineup in a bank (queues)
  - Function calls in programs (stacks)

Motivating Trees...

- It doesn't make sense to organize certain types of data into a linear structure
- Consider directories in a file system. They have a natural hierarchical structure that is difficult to represent linearly
- If we want to use a list, we might imagine storing the root directory at the first position and its subdirectories and files to its right
- But how would we know when the files of a subdirectory ends and we are back up one level?
- Other examples:
  - Structure of an HTML document
  - Structure of a Java program

Tree Definition

- A tree consists of
  - A set of nodes and
  - a set of edges, each of which connects two nodes



Tree Terminology

- Node: point in a tree; usually contains data
- Edge: connects two nodes. The edge is outgoing from one node and incoming into another node
- Root: the only node that has no incoming edge
- Children: Set of nodes that have incoming edges from the same node
- What are the nodes, edges, root and children of the tree on the previous slide?

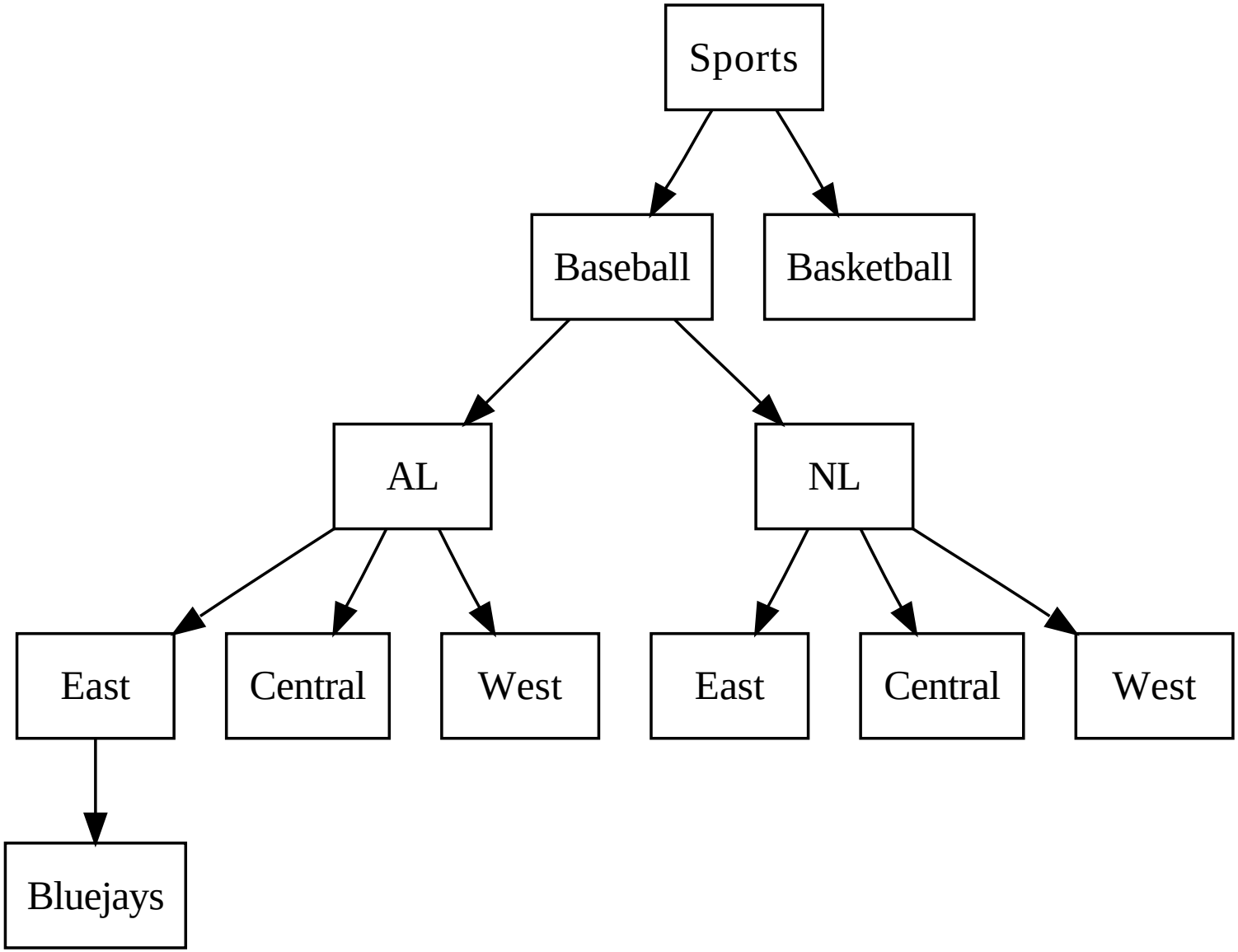
Tree Terminology...

- Parent: A node is the parent of all nodes to which it has outgoing edges.
- Siblings: Set of nodes that share a common parent.
- Leaf: A node that has no children (i.e., no outgoing edges).
- Internal node: A nonleaf node.
- Path: An ordered list of nodes from parent to child that are connected by edges
- Descendant: A node n is a descendant of some other node p if there is a path from p to n
- Subtree: A subtree of some tree T is a tree whose root node r is a node in T, and which consists of all the descendants of r in T and the edges among them

Tree Terminology...

- Branching Factor: Maximum number of children for any node
- Level (Depth): The level (or depth) of node n is the number of edges on the path from the root node to n
- Length of a path: Number of edges on a path
- Height: The maximum level of all nodes in the tree.
- **Yay! No more word salad!**

Sample Tree



What is the height of the tree? Branching factor? Depth of baseball? Length of path from sport to AL?

Back to Tree Definition

The nodes and edges of a tree must satisfy the following:

- One node in the tree is designated as the root node
- Each node, except the root, has exactly one parent
- There is a unique path from the root to every node
- There are no cycles; i.e, no paths that form "loops"
- Question: is any of these properties redundant?

Common Operations on Trees

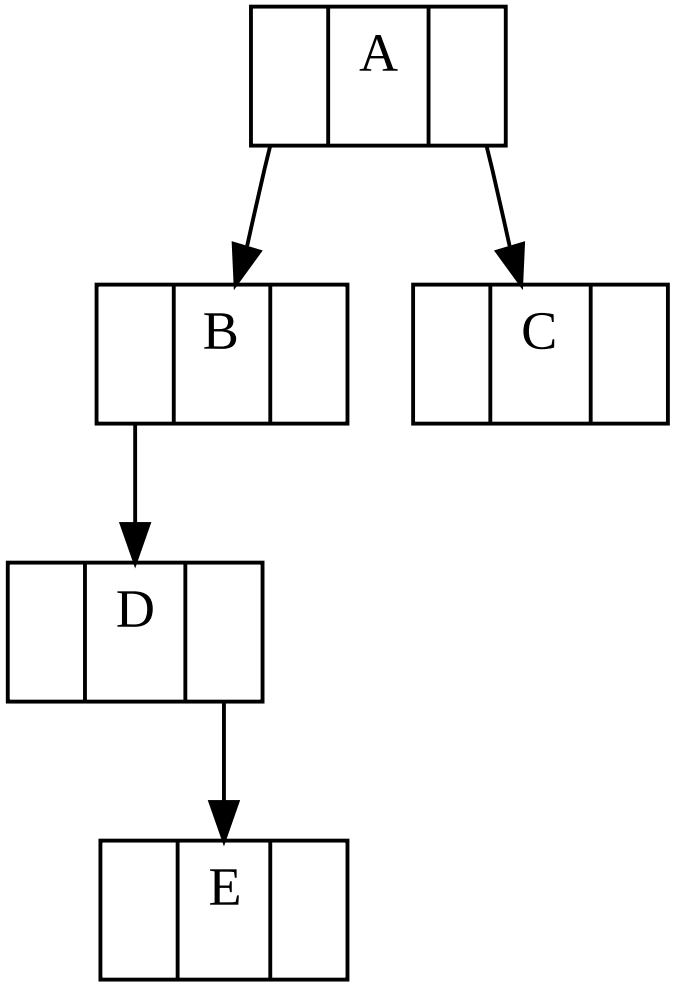
- Insert a new node
- Remove a node
- Traverse a tree: visit the nodes in some order and apply operations to each
- Attach a subtree at a node
- Remove a subtree

Representing Binary Trees

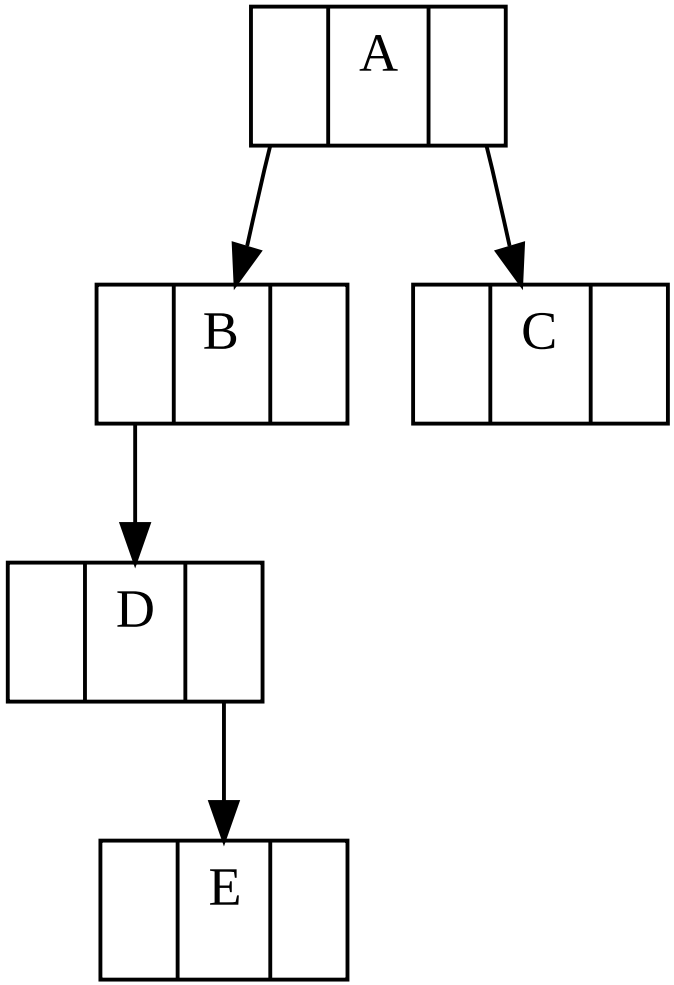
A tree with a maximum branching factor of 2 is called a binary tree. We can represent them programmatically by a list of lists, or nodes and references

- Using a list of lists
  - First element of the list contains the label of the root node.
  - Second element is the list that represents the left subtree.
  - Third element is the list that represents the right subtree.

TPS: Convert to List of Lists



TPS: Convert to List of Lists




```
['A',
['B', ['D', [], ['E', [], []]], []],
['C', [], []]]
```

Implementation of List of Lists

```
def BinaryTree(r):
  return   [r,   [] ,   [] ]

def insertLeft(root,newBranch):
  t = root.pop(1)
  if len(t) > 1:
    root.insert(1,[newBranch,t,[]])
  else:
    root.insert(1,[newBranch, [], []])
  return root
```

Is there an alternative for the first case?

Representing Binary Trees...

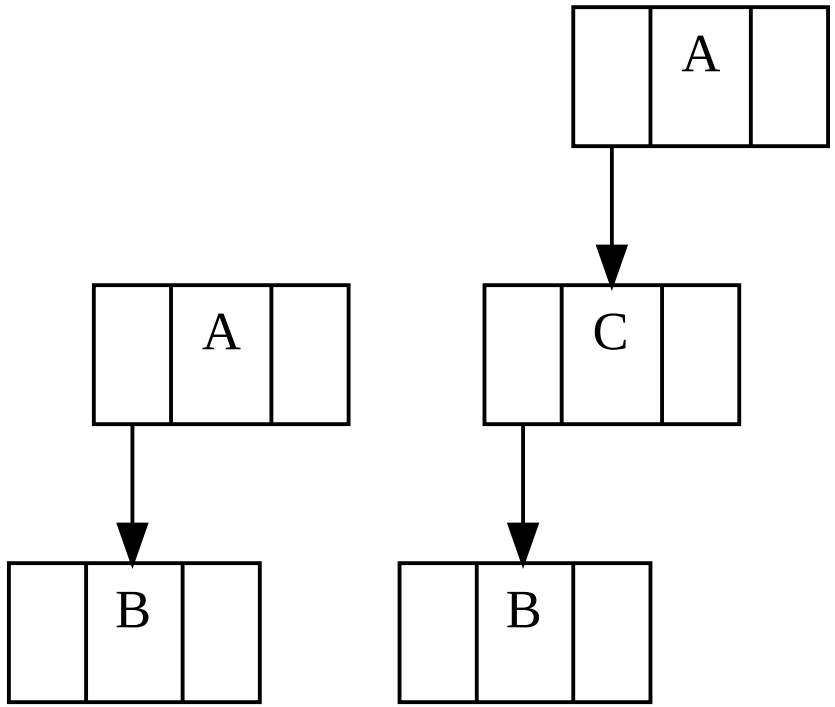
Using nodes and references

- Since the left and right children of a node are each roots of (sub)trees, we can use a recursive data structure
- Our tree objects will have attributes for the root value, left child and right child

```
class BinaryTree:
def __init__(self, rootObj):
  self.key = rootObj
self.left = None
self.right = None

def insertLeft(self,newNode):
  if self.left == None:
    self.left = BinaryTree(newNode)
  else:
    t = BinaryTree(newNode)
    t.left = self.left
    self.left = t
```

Example of Node Insertion



(a) Original Tree (b) Original Tree after inserting node C

Figure: Inserting a Node Using Code on Previous Slide

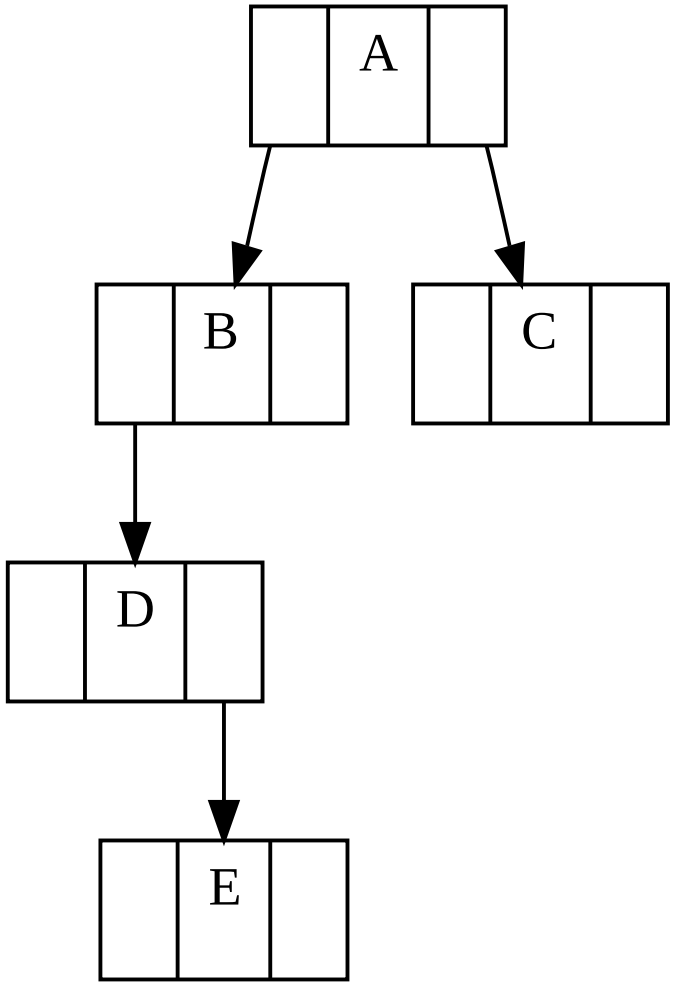
Tree Traversals

- If we traverse a list, we access each element in the list, possibly performing some operation with each element.
- We say we have visited a node when we have done something with it (like print it or otherwise process it)
- Two traversals on lists: left to right, right to left
- Trees give us more ways to systematically visit each node:
  - Preorder
  - Inorder
  - Postorder

Tree Traversals...

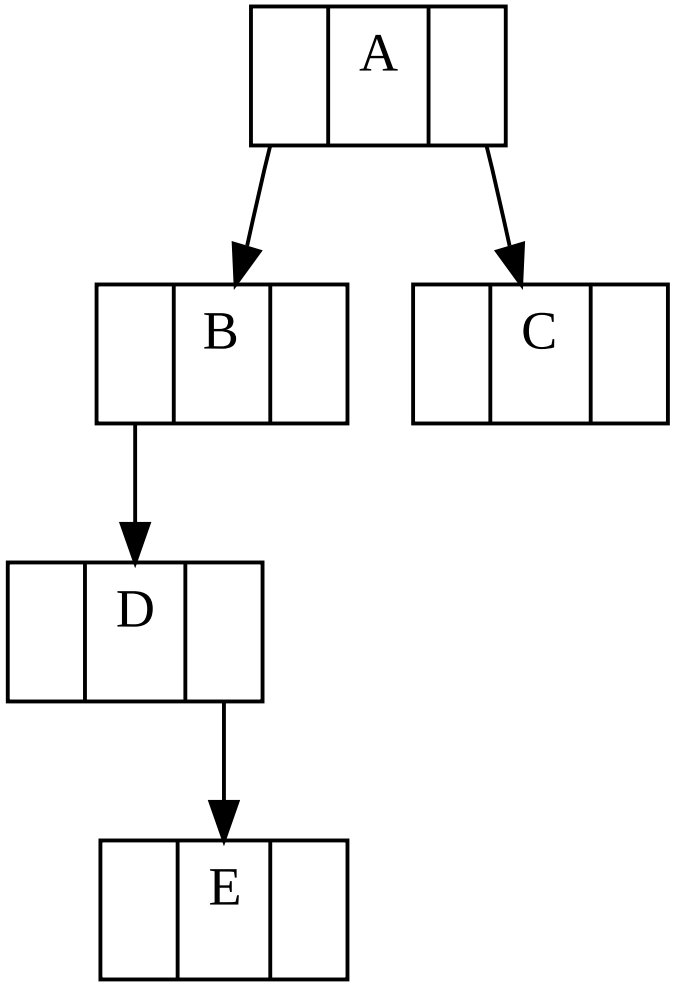
- Preorder: Visit the root node, do a preorder traversal of the left subtree, and do a preorder traversal of the right subtree
- Inorder: Do an inorder traversal of the left subtree, visit the root node, and then do an inorder traversal of the right subtree.
- Postorder: do a postorder traversal of the left subtree, do a postorder traversal of the right subtree, and visit the root node

Example: Traversal



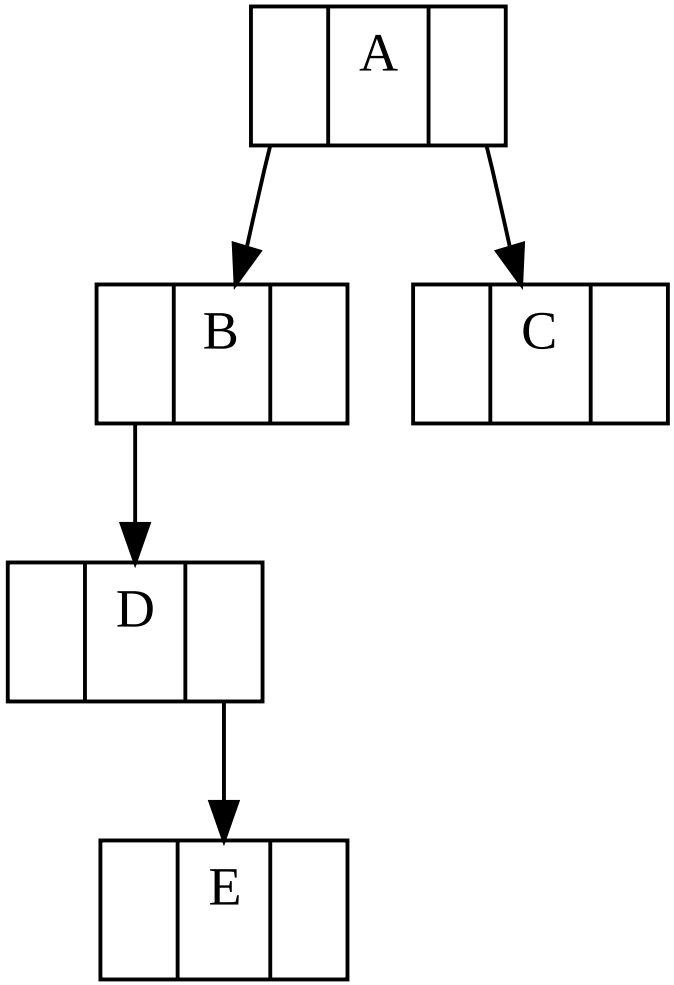
- Preorder:

Example: Traversal



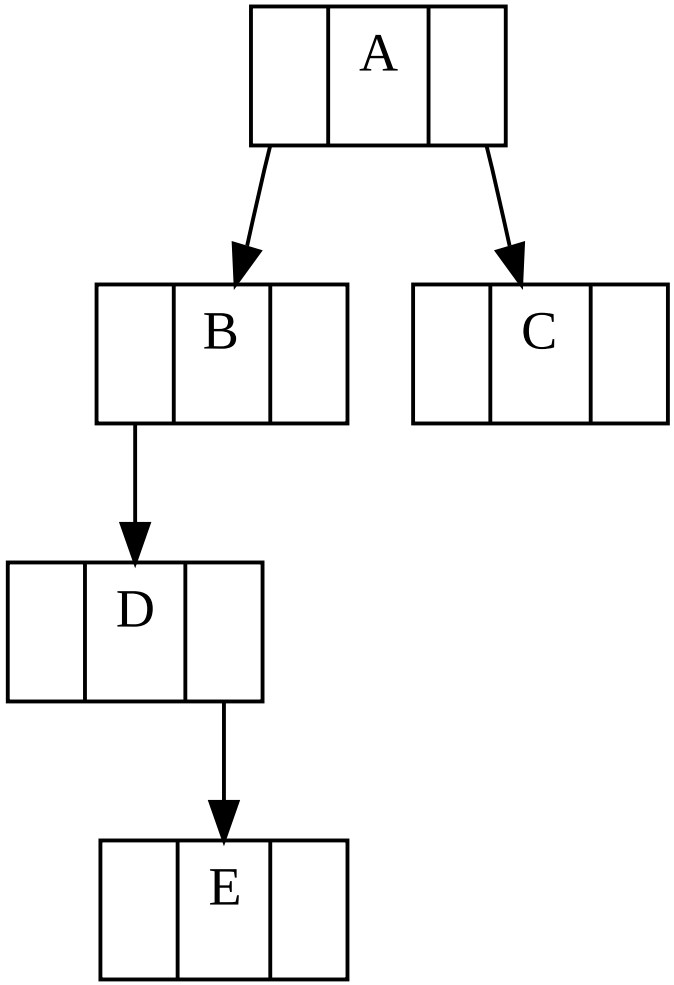
- Preorder: A, B, D, E, C
- Inorder:

Example: Traversal



- Preorder: A, B, D, E, C
- Inorder: D, E, B, A, C
- Postorder:

Example: Traversal



- Preorder: A, B, D, E, C
- Inorder: D, E, B, A, C
- Postorder: E, D, B, C, A

Traversal Code

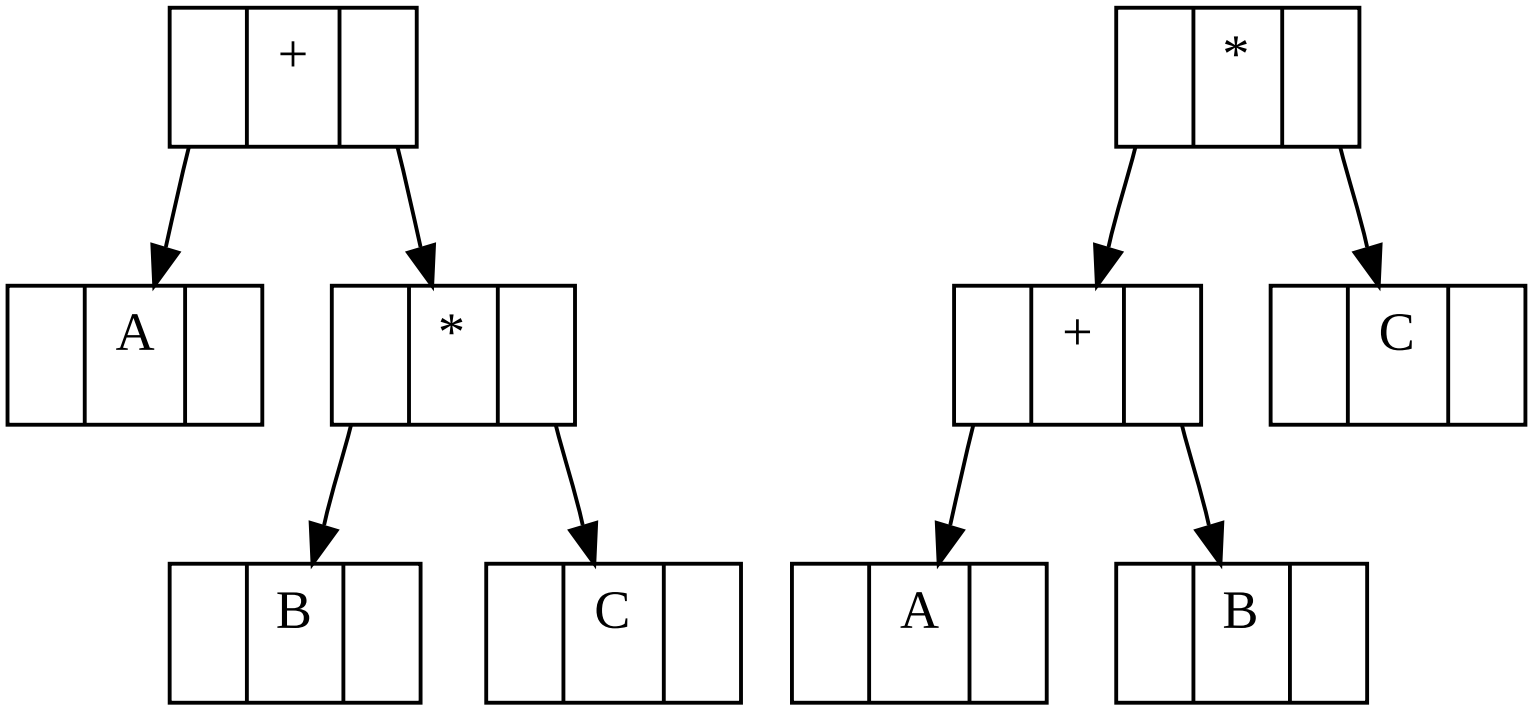
We can directly translate the traversals into code; e.g.:

```
def preorder(tree):
    if tree:
        print tree.getRootValue()
        preorder(tree.getLeftChild())
        preorder(tree.getRightChild())

def inorder(tree):
    if tree:
        inorder(tree.getLeftChild())
        print tree.getRootValue()
        inorder(tree.getRightChild())

def postorder(tree):
    if tree:
        postorder(tree.getLeftChild())
        postorder(tree.getRightChild())
        print tree.getRootValue()
```

Parse Trees



(a) Tree for $A + B * C$ (b) Tree for $(A + B) * C$

Figure: Two parse trees forcing different orders of operation

Uses of Expression Trees

- A slightly modified inorder traversal gives us a fully parenthesized expression corresponding to the tree's order of operations

```
def printexp(tree):
  sVal = ""
  if tree:
    sVal = '(' + printexp(tree.getLeftChild())
    sVal = sVal + str(tree.getRootVal())
    sVal = sVal + printexp(tree.getRightChild())+')'
  return sVal
```

Uses of Expression Trees

- A postorder evaluation gives us a way to evaluate the expression in an expression tree
- Postorder will recursively calculate the value for the left subtree, then the value for the right subtree
- The parent of these two subtrees will be an operator which we can apply to the above results to yield the result for the entire subtree

- Preorder traversal gives us our expression in prefix notation
- Postorder traversal gives us our expression in postfix notation

Prefix, Infix, Postfix

- We usually write mathematical expressions with operators between their operands
- This requires us to use parentheses when the standard precedence rules are not what we want
- For example, in $(a + b) * c$, the parentheses are necessary to force the addition to happen first
- Prefix notation places the operator before its two operands
- Postfix notation places the operator after its two operands
- These formats never require parentheses

Prefix, Infix, Postfix...

- Example: `a+b*c`
  - In prefix: `+a*bc`
  - In postfix: `abc*+`
- Example: `(a+b)*c`
  - In prefix: `*+abc`
  - In postfix: `ab+c*`

Application: Building Parse Trees

- Compilers take source code and convert it into a tree representation for later processing
- This naturally represents the nesting structure of programs (e.g. a class (parent) has methods (children); a function (parent) has parameters and a body (children))
- As we know, expressions can be parsed into trees to clearly show order of operations
- Our expressions will be represented by a Python list, whose components are operands, the operator +, or the operator *
- We will write a procedure, **expr**, for taking such an expression and producing its (binary) parse tree

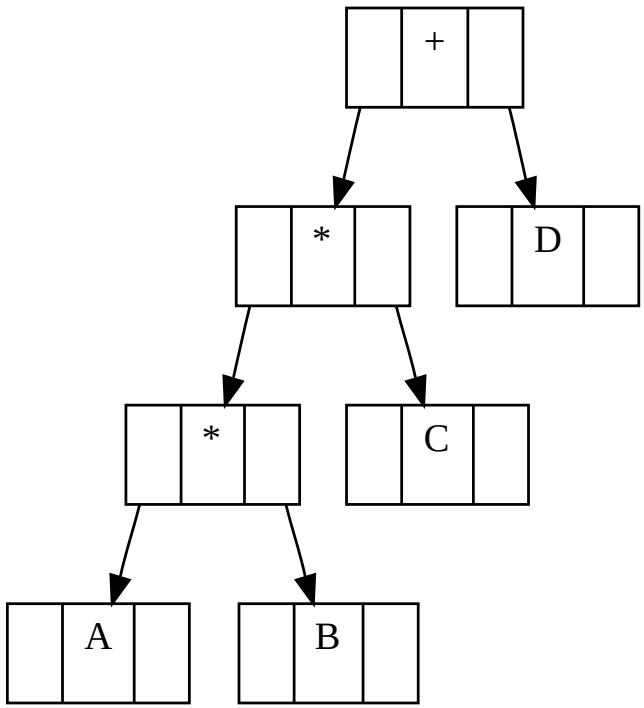
Application: Building Parse Trees...

- The simplest expression to convert is a single operand (we'll call it a **factor**)
- It becomes the root and sole node of its parse tree

```
def factor (l):
  t = BinaryTree (l[0])
  l.pop(0)
  return t
```

Application: Building Parse Trees...

- Consider the expression `a*b*c+d`
- The left subtree of + will represent `a*b*c`, and the right subtree will represent `d`
- We'll therefore write a procedure to create the tree for a chain of $*$ operators (which we'll call a **term**)
- We'll associate multiplication to the left:
  $a * b * c + d = (((a * b) * c) + d)$



Application: Building Parse Trees...

```
def term (l):
  t = factor (l)
  while len(l) > 0 and l[0] == '*':
    l.pop(0)
    t2 = factor (l)
    tNew = BinaryTree ('*')
    tNew.setLeftSubtree (t)
    tNew.setRightSubtree (t2)
    t = tNew
  return t
```

Application: Building Parse Trees...

- We can now write the top-level function for constructing the parse tree for a chain of + operators
- Again, + associates to the left, so each time we read another +, the old chain of + terms becomes a left child

```
def expr (l):
  t = term (l)
  while len(l) > 0 and l[0] == '+':
    l.pop(0)
    t2 = term (l)
    tNew = BinaryTree ('+')
    tNew.setLeftSubtree (t)
    tNew.setRightSubtree (t2)
    t = tNew
  return t
```

Application: Building Parse Trees...

- How can we handle parentheses?
- Idea1 : a parenthesized subexpression is allowed anywhere an operand is allowed (this tells us where to add code)
- Idea 2: a parenthesized subexpression is an expression (this tells us what code to add)
- This results in `expr` calling `term`, calling `factor`, calling `expr`
- This is mutual recursion

Application: Building Parse Trees...

```
def factor (l):
  if l[0] == '(':
    l.pop(0)
    t = expr(l)
    l.pop (0)
  else:
    t = BinaryTree (l[0])
    l.pop(0)
  return t
```