CS416

Introduction to Computer Science II

Spring 2018

8 Trees and Graphs

Chapter 15

Previously in 416

- Abstract Data Types
  - Specification
  - Stacks, Queues, Lists
  - Dictionary
- Concrete Data structures
  - Implementation
  - Lists, arrays, hash tables

Preview

- *Program State*
- Tree Abstract Data Type
- Tree data structures
  - binary trees
  - n-ary trees
- Tree algorithms
  - Tree algorithm complexity
  - Quadtrees
- Graphs

Linear Data Organization

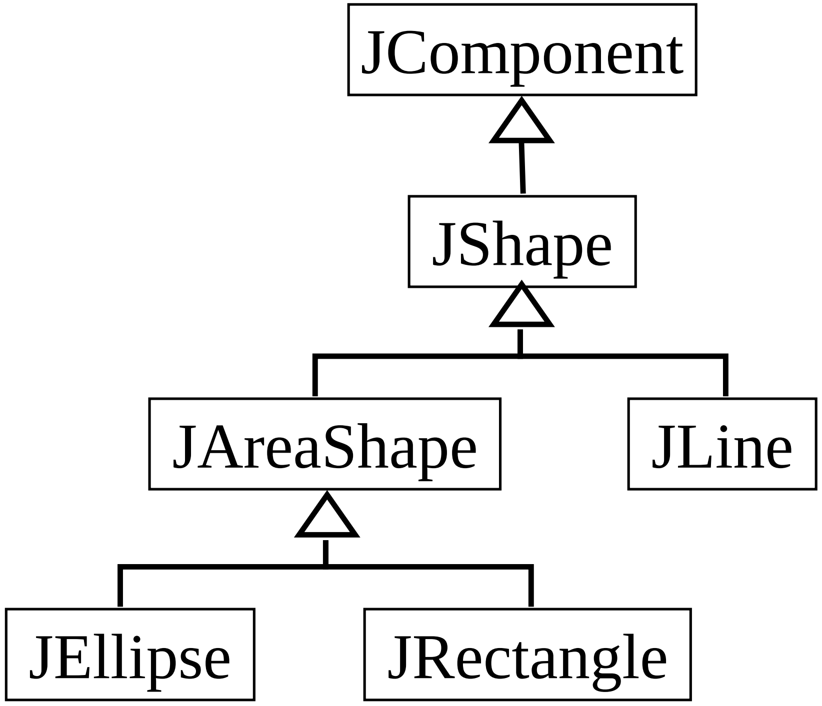
- For the most part we’ve been dealing with data organized in a *linear* fashion:
  - list, stack, queue, array are all essentially linear
- Linear organization is great for lots of things
  - class rank, dictionaries, price lists, etc.
- Lots of data doesn’t map well to a linear structure
  - how do you linearize a company’s organization chart or the classification of biological species?
  - such data is *hierarchical* in nature

Hierarchical Data

- Lots of data can be naturally organized into groups where the groups can be further organized into subgroups and so on
  - organization hierarchies
  - lots of “classification” schemes
    - biology, library holdings, knowledge bases, etc.
  - genealogical information
  - file systems
  - inheritance relations in object-oriented languages
  - lots more

Modeling Hierarchical Data

- Visual modeling
  - Arrange members of a group “under” a representation for the group
- Modeling in the computer
  - Trees



Abstract Trees

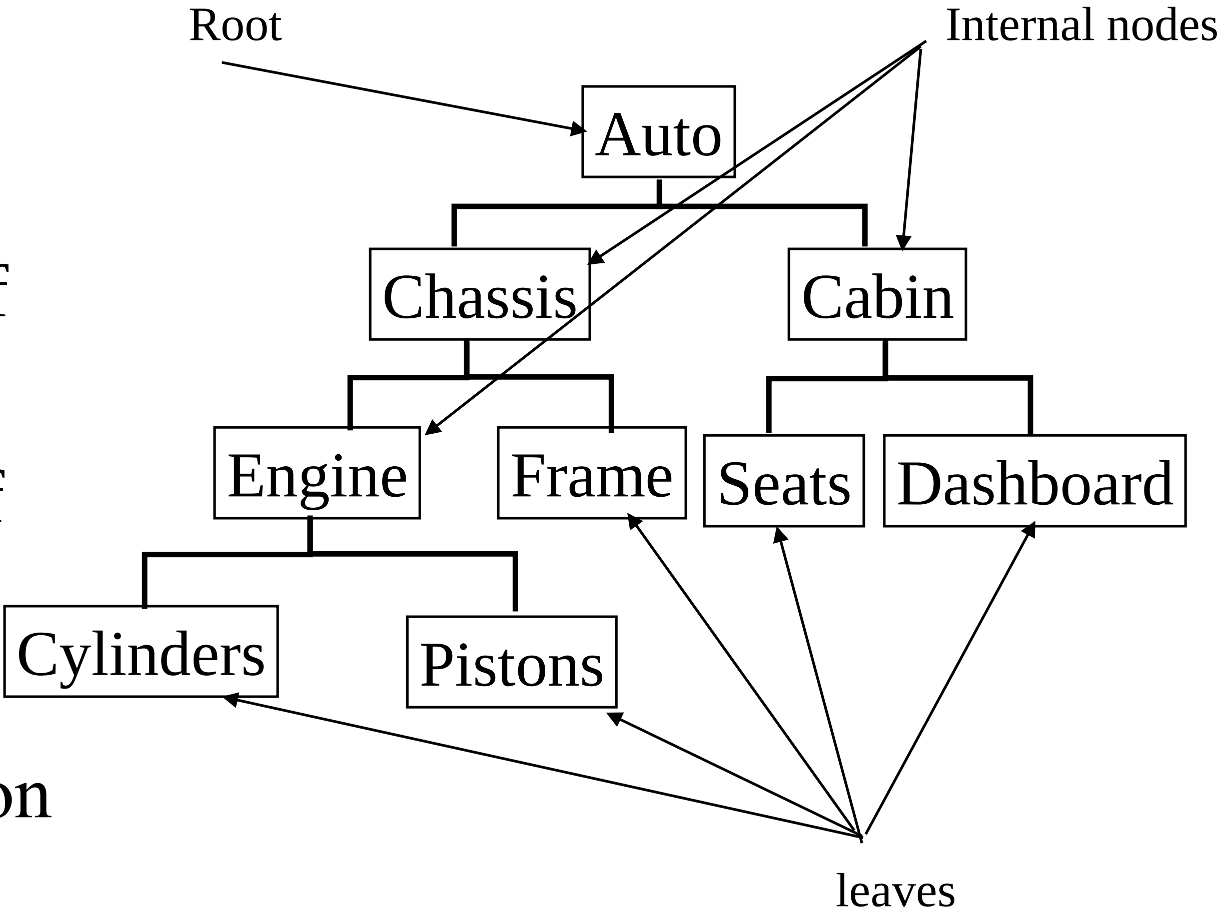
- Start from the *top* and work *down*
  - Top is the most general group; it encompasses everything else in the tree
  - Top of the hierarchy, the CEO, etc.
  - We draw it at the top of the page or screen
  - But, we call it the *root* of the tree

Tree Terminology

- *Root* of the tree
  - encompasses entire tree
  - represented by an information *node*
  - has *children* that are also represented by information *nodes*
- Tree *nodes*
  - Each node is the *root* of its own *subtree*
- *Leaves* -- nodes that have no children
- *Internal nodes* -- nodes with children

Tree Example
Object Hierarchy

- <u>Object</u> hierarchy is a *composition* (*part of*) hierarchy
  - Chassis and Cabin (parts of Auto) are children of Auto
  - Engine and Frame (parts of Chassis) are children of Chassis, etc.
  - “Stopping” point depends on desired level of abstraction



Binary Tree

- In a *binary* tree, every node can have *at most* two children
- This is a pretty severe restriction for representing most hierarchies
- But, it’s great for representing a *search tree*

Binary Search Tree

- Every node contains a *Data* object that contains a *key*.
- Every node also contains two pointers (references) to other nodes
  - *left* references a node that is the root of the *subtree* containing all *Data* objects whose *keys* are *less than* (using some ordering) the *key* at this node
  - *right* references a node that is the root of the *subtree* containing all *Data* objects whose *keys* are *greater than* the *key* at this node

Adding Data to a BST

- Given a binary search tree, add a new entry

```
add( data ):
  if ( root == null )
     root = new Node( data )
  else
     addNode( root, data )
```

```
addNode( Node root, data ):
  if ( data < root.data )
     if root.left == null
        root.left = new Node( data )
     else
        addNode( root.left, data )
  else if ( data > root.data )
     if root.right == null
        root.right = new Node( data )
     else
        addNode( root.right, data )
  else // data == root.data
     error message
```

If tree is empty, make new data the root.

Else, find where it goes and add it.

If new data is “less than” this node it goes in this node’s *left* subtree.

If *left* subtree is empty, a new node becomes this node’s *left*

Else add it to this node’s *left* subtree

If new data is “greater than” this node it goes in this node’s *right* subtree.

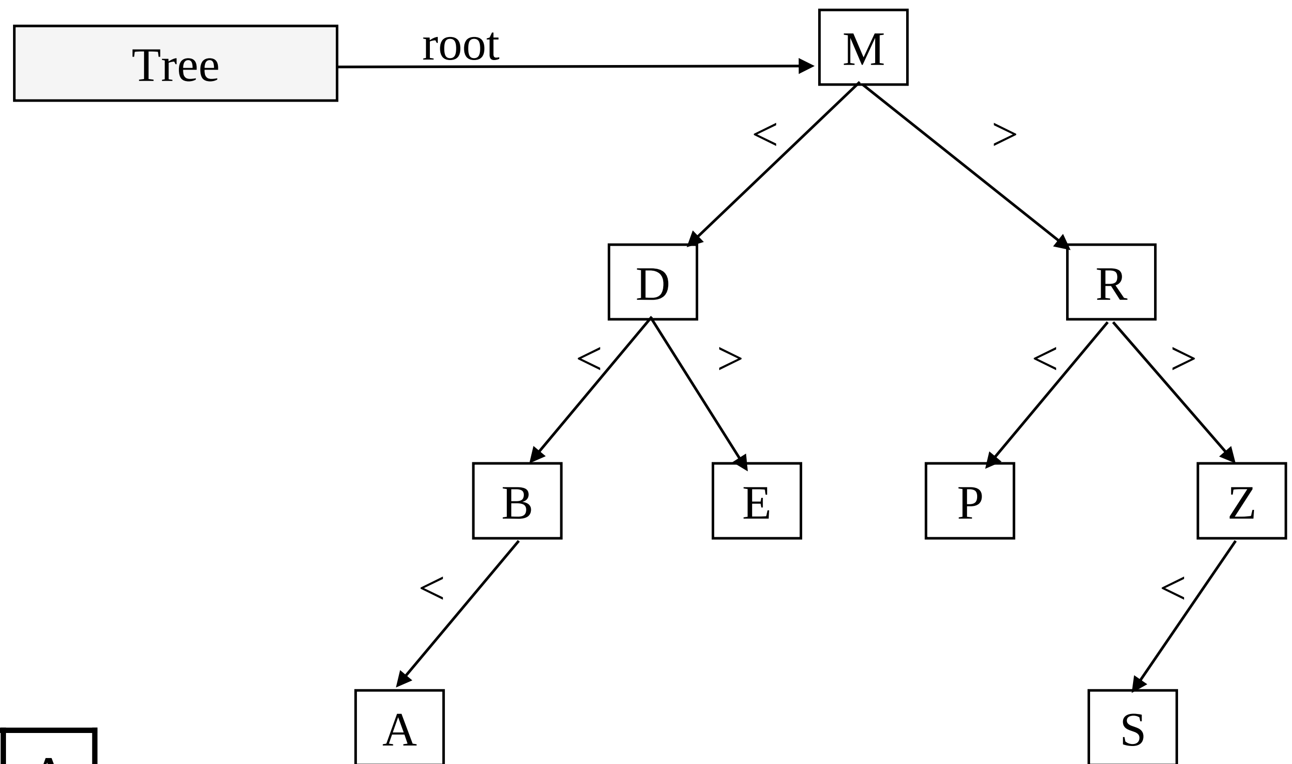
Empty *right* subtree means new data becomes *right* subtree

Else add it to this node’s *right* subtree

Else data is already in the tree

Binary Search Tree Example

- *Tree* has a reference to the root, that is initially *null*
- Input stream: a sequence of *Data* objects to be added to the tree.

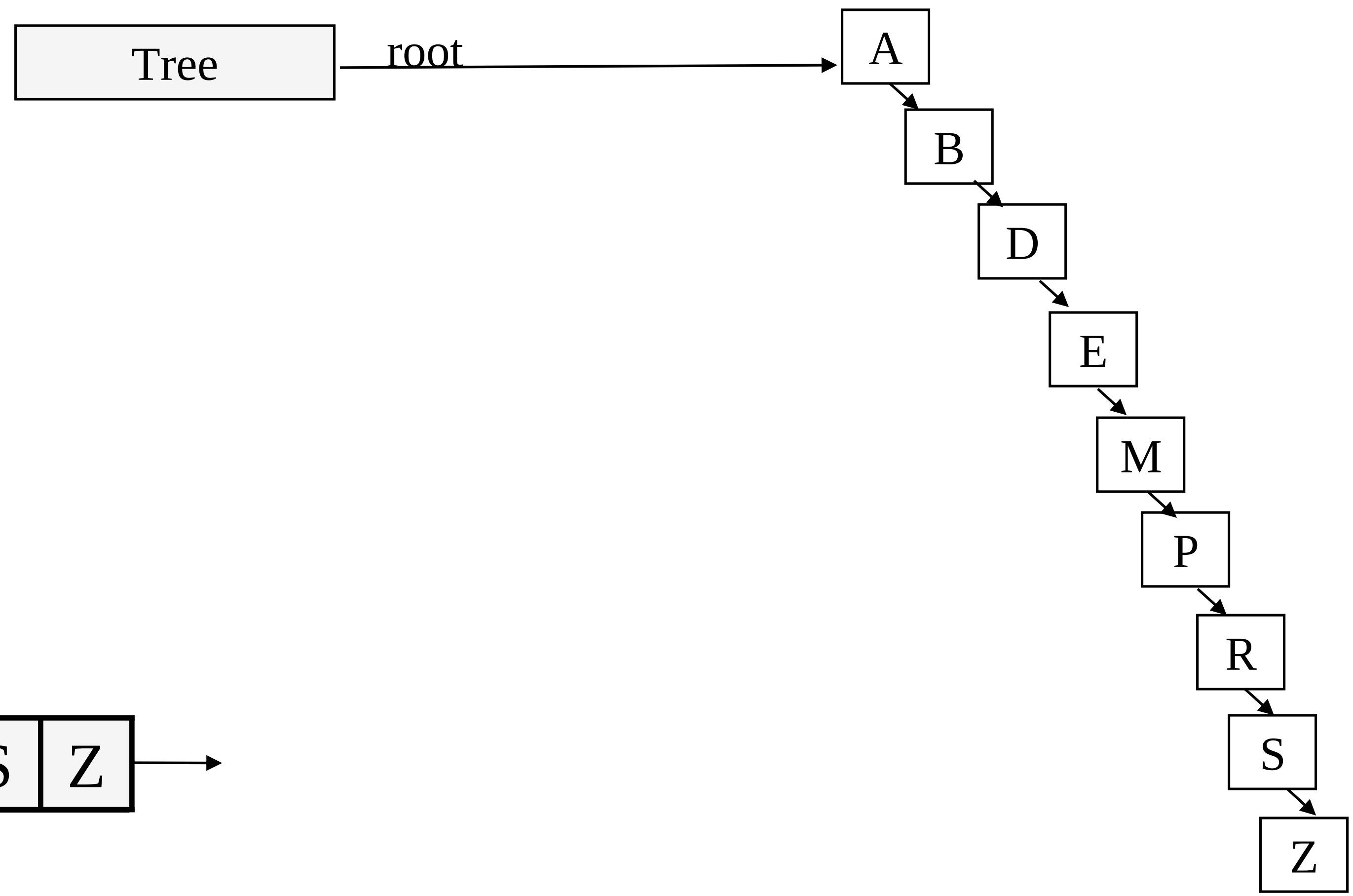


M	R	D	P	Z	B	E	S	A

Add M to Root	Add R to M.right	Add D to M.left	Add P to R.left	Add Z to R.right	Add B to D.left	Add E to D.right	Add S to Z.left	Add A to B.left

Binary Search Tree Example 2

- What about a different input order?
- Get very different "tree"!



A	B	D	E	M	P	R	S	Z

Add A to Root	Add B to A.right	Add D to B.right	Add E to D.right	Add M to E.right	Add P to M.right	Add R to P.right	Add S to R.right	Add Z to S.right

BinarySearchTree.find

```
// start recursion from here
public Data find( String str ){
 Node ret = findNode( _root, str );
  if ( ret == null )
    return null;
  else
    return ret.data;
}
```

```
// find "s" in subtree rooted at "r"
private Node findNode( Node r, String s ){
  if ( r == null )
    return null;
  int cmp = s.compareTo( r.data.key );
  if ( cmp < 0 ) // left branch
    return findNode( r.left, s );
  else if ( cmp > 0 )// right branch
    return findNode( r.right, s );
  else // found it!
    return r;
}
```

Iterative BST.find

```
// iterative find isn't a lot different
public Data find( String s )
{
  Node cur   = _root;
  Data found = null;
  while ( cur != null && found == null )
  {
    int cmp = s.compareTo( cur.data.key );
    if ( cmp < 0 )  // check left branch
       cur = cur.left;
    else if ( cmp > 0 ) // check right branch
       cur = cur.right;
    else  // found it
      found = cur.data;
  }
  return found;
}
```

Iterator-based find

It looks pretty simple

```
// iterative find with iterator
public Data find( Data d )
{
  Node cur   = _root;
  Data found = null;
  Iterator<Data> iter = this.iterator();
  while ( iter.hasNext() )
  {
    Data test = iter.next();
    if ( test.equals( d ) )
      return test;
  }
  return null;
}
```

BUT, it's very inefficient compared to the binary search approach, since the iterator just returns every element in the tree (in some order). No matter what order that is, it makes the tree look like a list or array.

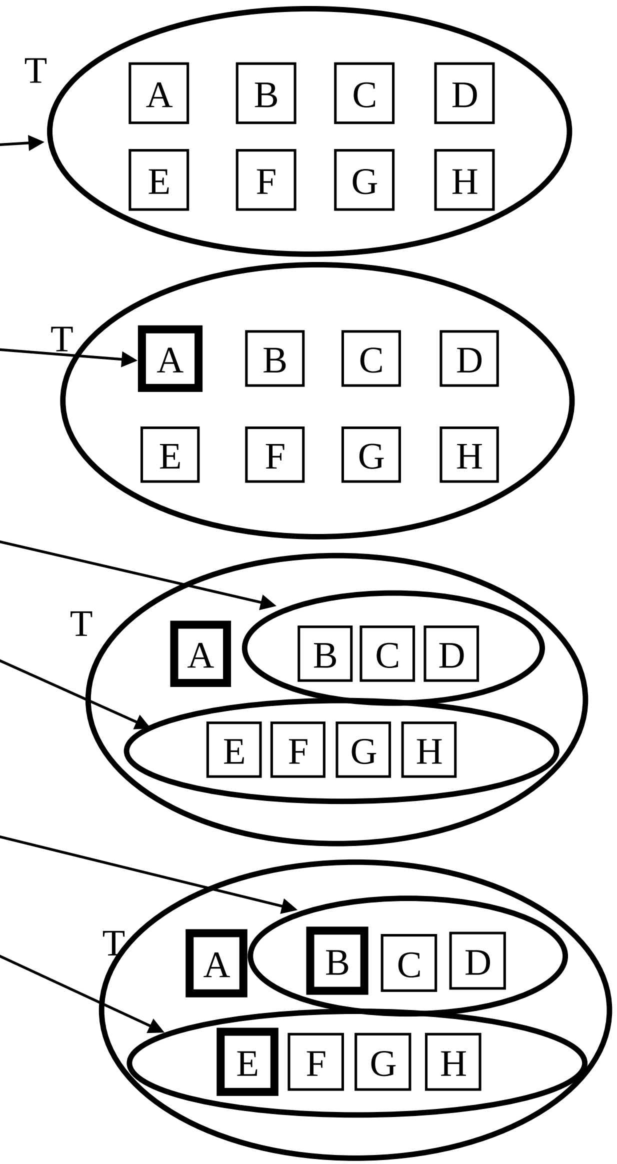
Formal Tree Definition

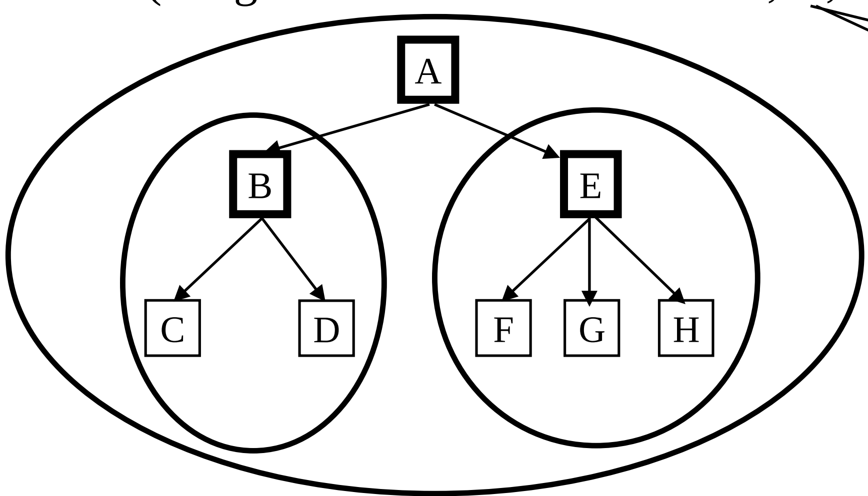
- There is a rigorous “mathematical” tree definition
- A tree is a finite set, T, of one or more nodes such that:
  - Exactly 1 node has no *predecessor*; this is the root
  - Remaining nodes can be partitioned into *disjoint* sets, $T_1, T_2, \ldots, T_m$ for some $m >= 0$
  - $T_1, T_2, \ldots, T_m$ are all trees; they are called subtrees
- This is a recursive definition: a tree is defined as a *root* plus 0 or more trees (*subtrees*).
- Note: $m <= 2$ for a binary tree

Example Tree Using Formal Definition

- T is a set of nodes
- 1 node is the *root*
- Other nodes are partitioned into disjoint sets
- The partitions are subtrees

(imagine circles around nodes C, D, F, G, H)



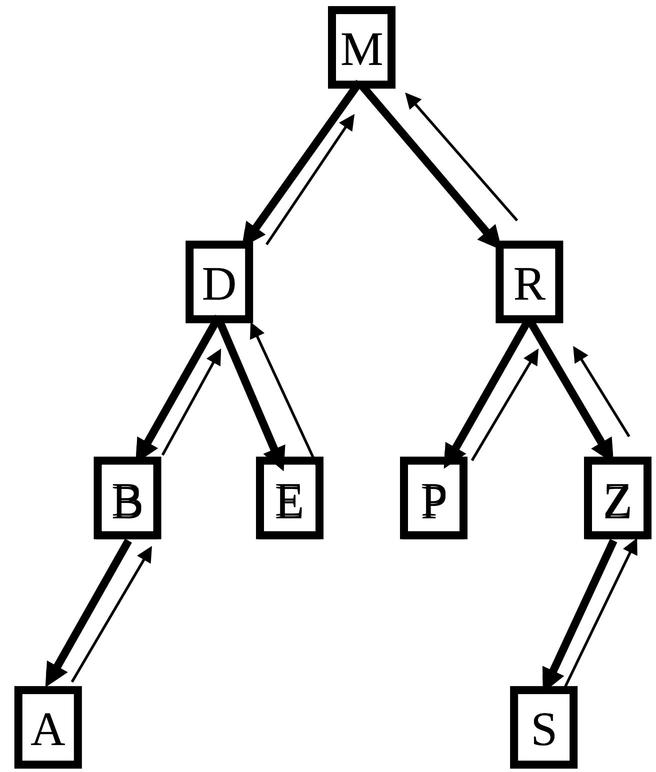


More Terminology

- Node, root, child, parent, leaf, internal node
- **Edge**: the “link” from parent to child
- **Depth** of a node: # edges to follow to reach root
- **Height** of a tree: depth of deepest node
  - some define it as depth of deepest node +1
- **Degree** of a node: # of its children
  - leaf degree is always 0

Printing the Binary Tree

- Suppose you want to print an alphabetic list of all nodes in a binary search tree
- Need to traverse tree from root
- At each node, must
  1. Print all nodes in the left subtree
     - All come before this node in the alphabet
  2. Print the node
  3. Print all nodes in the right subtree
     - All come after this node in the alphabet



Output

A	B	D	E	M	P	R	S	Z

Tree Traversal

- *In-order* tree traversal (ex. printing the binary tree)
  - Traverse left subtree, “process” node, traverse right subtree
- *Pre-order* tree traversal
  - “Process” node, traverse left subtree, traverse right subtree
- *Post-order* tree traversal
  - Traverse left subtree, traverse right subtree, “process” node

Tree Traversal Algorithms

- Tree traversal algorithms are naturally expressed with recursion
- High-level algorithms map directly to code

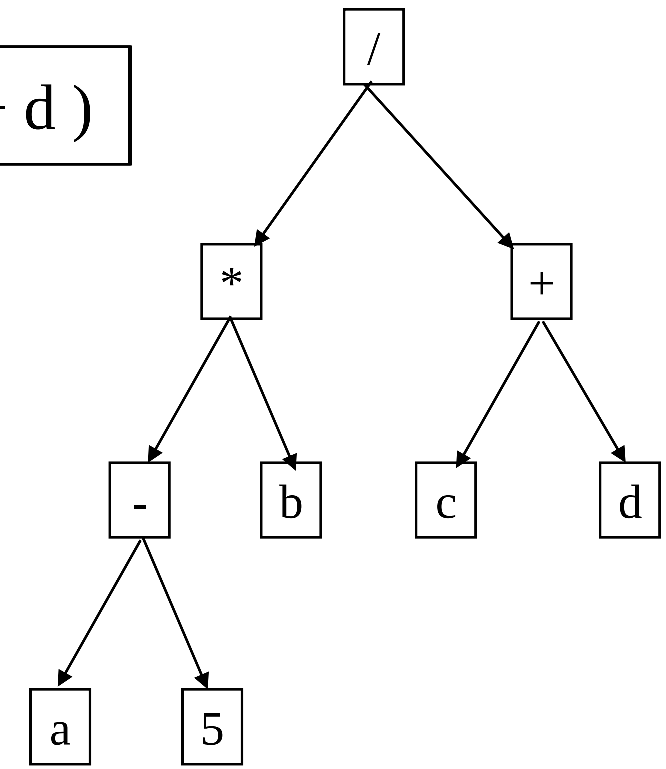
```
inOrder( node ):
  if ( node is null )
     return
  else
     inOrder( node.leftTree )
     process node.name
     inOrder( node.rightTree )
```

```
preOrder( node ):
  if ( node is null )
     return
  else
     process node.name
     preOrder( node.leftTree )
     preOrder( node.rightTree )
```

```
postOrder( node ):
  if ( node is null )
     return
  else
     postOrder( node.leftTree )
     postOrder( node.rightTree )
     process node.name
```

Expression Trees

(a - 5)*b / (c + d)



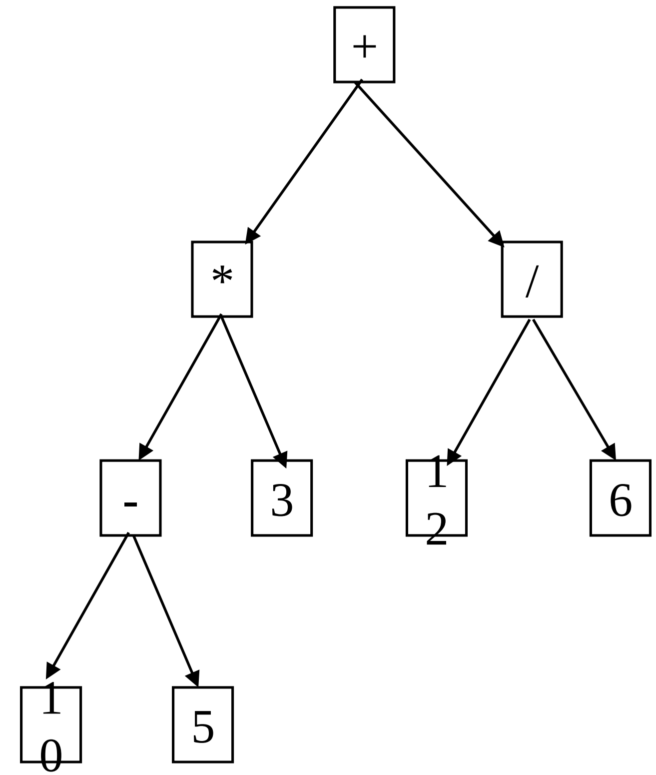
- Arithmetic expressions map well to trees
  - most operators are binary and have two operands
  - internal nodes are operators, leaves are operands
  - operators that need to be executed early are “deep”
  - don’t need parentheses in the tree

pre-order print:	/*-a5b+cd
in-order print:	a-5*b/c+d
post-order print:	a5-b*cd+/

Evaluate Expression Tree

- High level algorithm

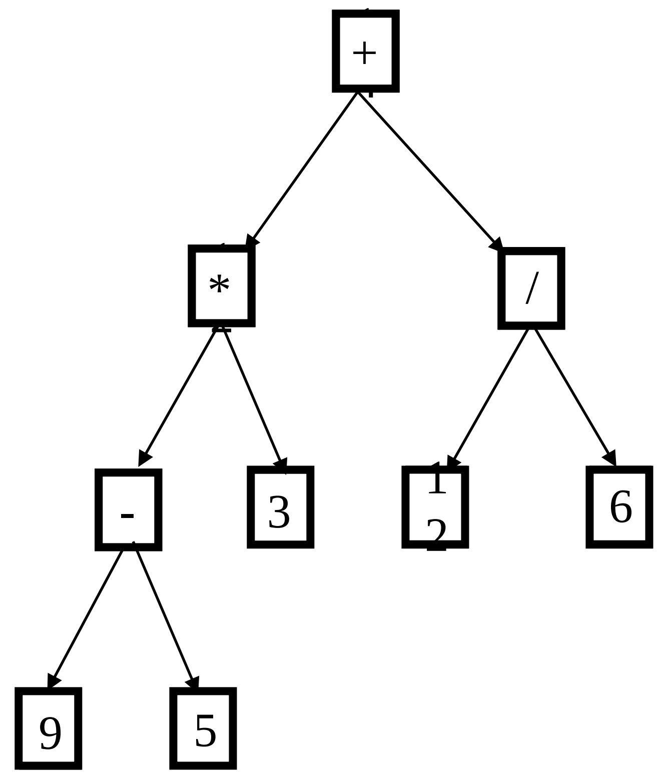
```
float eval( node ):
if ( node is operand )
  return operand.value
else
  lVal = eval( node.left )
  rVal = eval( node.right )
  replaceNode( op( lVal, rVal ))
```



replaceNode applies the operator using *lVal* and *rVal* and replaces the operator node with an operand node containing the result of the operation.

Expression Tree Evaluation

```
float eval( node ):
if ( node is operand )
  return operand.value
else
  lVal = eval( node.left )
  rVal = eval( node.right )
  replaceNode(lVal op rVal)
```



eval(+)
- eval(*)
  - eval(-)
    - eval(9)
    - eval(5)
    - replace(9-5)
  - eval(3)
  - replace(4*3)

eval(+) (cont)
- eval(/)
  - eval(12)
  - eval(6)
  - replace(12/6)
- replace(12*2)

Postfix Expression to Expression Tree

Review the algorithm for evaluating postfix

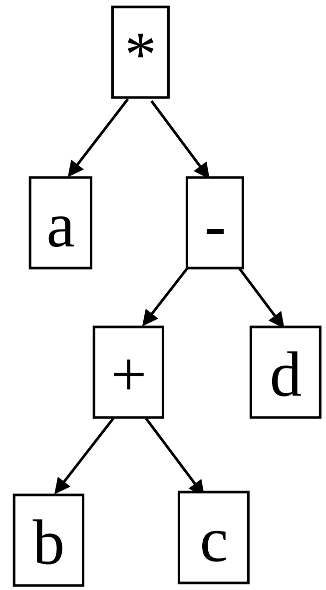
```
stack = empty
for each token in expression
  if token is operand
    push operand onto stack
  else // operator
    pop right operand from stack
    pop left operand from stack
    calculate left op right
    push result
stack.top is final result
```

Creating an expression tree is essentially identical

```
stack = empty
for each token in expression
  if token is operand
    stack.push( new operandNode )
  else // operator
    opNode = new operatorNode
    opNode.right = stack.pop()
    opNode.left  = stack.pop()
    push opNode
stack.top is root of expression tree
```

Postfix to Expression Tree Example

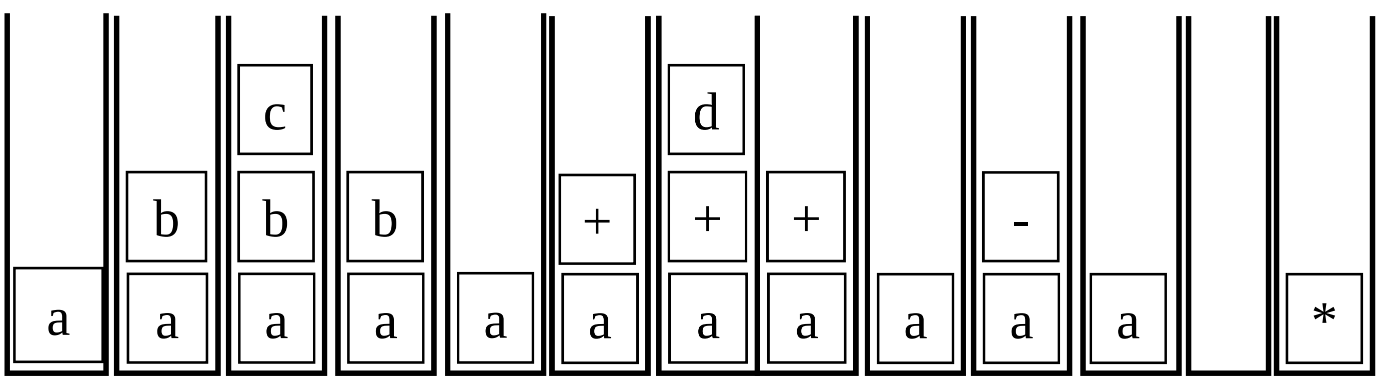
```
stack = empty
for each token in expression
  if token is operand
    stack.push(new operandNode)
  else // operator
    opNode = new operatorNode
    opNode.right = stack.pop()
    opNode.left  = stack.pop()
    push opNode
stack.top is root of tree
```



Expr:

a	b	c	+	d	-	*

In	Action	L	R
a	push node(a)		
b	push node(b)		
c	push node(c)		
+	pop to R, pop to L	b	c
	push +(b,c)		
d	push d		
-	pop to R, pop to L	+	d
	push -(+,d)		
*	pop to R, Pop to L	a	-
	push *(a,-)		



Stack history

Infix to Tree Algorithm

- Could generate infix to postfix and then postfix to tree, but why not generate tree directly from infix?
- Minor variation of infix to postfix algorithm
  - need an *operand stack* (**randStack**) as well as an *operator stack* (**opStack**)
  - when we get an *operand* need to create an *operand* <u>*node*</u> and push it onto *randStack*
  - When we pop an *operator* from *opStack*:
    - create an operator *node*
    - pop 2 operand nodes as right/left subtrees of the operator node
    - push the operator node onto *randStack*
    - this step is the equivalent of *evaluating* the expression

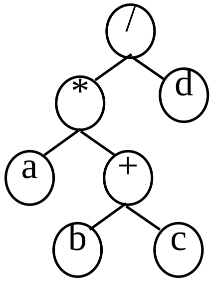
Infix to Tree Algorithm 2

```
Infix to postfix:
create empty opStack and postfix list
for each token in infix expression
  if token is operand
    add token to postfix
  else if token is "("
    opStack.push( token )
  else if token is ")"
    while opStack.top() != "("
      add opStack.pop() to postfix list
    opStack.pop() // pop "("
  else // it is an operator
    while ( opStack is not empty
          and prec(top) >= prec(token))
      add opStack.pop() to postfix list
    opStack.push( token )
// copy remaining operators to output
while ( opStack !empty )
  add opStack.pop() to postfix list
```

```
Infix to tree:
create empty opStack, randStack
for each token in infix expression
  if token is operand
    randStack.push( new operandNode )
  else if token is "("
    opStack.push( token )
  else if token is ")"
    while opStack.top() != "("
      pushOpNode( opStack.pop());
    opStack.pop() // pop "("
  else // it is an operator
    while ( opStack is not empty
          and prec(top) >= prec(token))
      pushOpNode( opStack.pop() )
    opStack.push( token )
// copy remaining operators to output
while ( opStack !empty )
  pushOpNode( opStack.pop());
```

Remember: "(" in opStack has **lowest** precedence of all, so it cannot be popped in this step.

```
pushOpNode( op ):
  opNode = new operatorNode( op );
  opNode.right = randStack.pop();
  opNode.left  = randStack.pop();
  randStack.push( opNode );
```



Infix to Tree Example

Example: a * (b + c) / d

```
Infix to tree:
create empty opStack, randStack
for each token in infix expression
  if token is operand
    randStack.push( new operandNode )
  else if token is "("
    opStack.push( token )
  else if token is ")"
    while opStack.top() != "("
      pushOpNode( opStack.pop())
    opStack.pop() // pop "("
  else // it is an operator
    while ( opStack is not empty
          and prec(top) >= prec(token))
      pushOpNode( opStack.pop())
    opStack.push( token )
// copy remaining operators to output
while ( opStack !empty )
   pushOpNode( opStack.pop())
```

```
pushOpNode( op ):
   opNode = new operatorNode( op );
   opNode.right = randStack.pop();
   opNode.left  = randStack.pop();
   randStack.push( opNode );
```

Input	Step	OpStack	RandStack	L	R
a * (b + c) / d	1		a		
a * (b + c) / d	6	*	a		
a * (b + c) / d	2	(*	a		
a * (b + c) / d	1	(*	b a		
a * (b + c) / d	6	+ (*	b a		
a * (b + c) / d	1	+ (*	c b a		
a * (b + c) / d	3a	+ (*	a	b	c
	3b,4	*	$\oplus$ a		
a * (b + c) / d	5a			a	$\oplus$
	5b		$\otimes$		
	6	/	$\otimes$		
a * (b + c) / d	1	/	d $\otimes$		
a * (b + c) / d	7a	/		$\otimes$	d
	7b		$\oslash$		

The circled operators in the RandStack identify operator **nodes.**