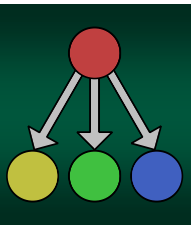
Trees

Section 1.5

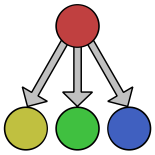


Introduction to Trees

Let the data grow

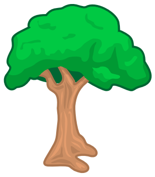
Introduction to Trees

- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relationship to zero *or more* nodes

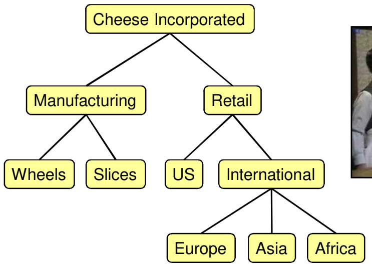


Some Applications

- Organizational charts
- Class hierarchy
- Disk directory and subdirectories
- Structure of a program



Tree Example



Cheese Incorporated

Manufacturing

Retail

Wheels

Slices

US

International

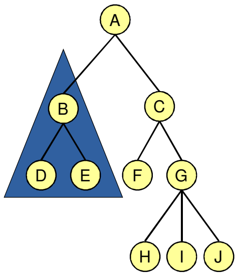
Europe

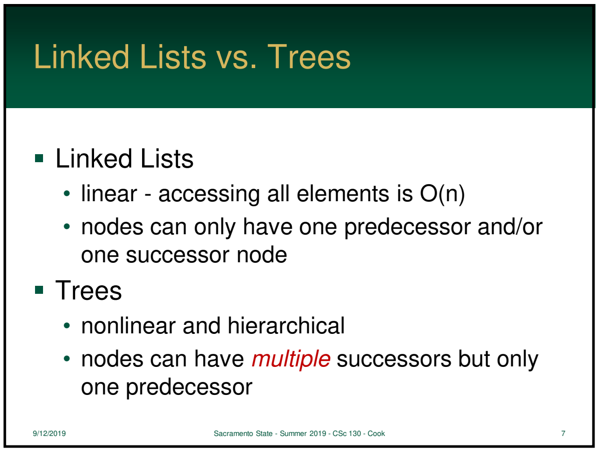
Asia

Africa

Trees are Recursive

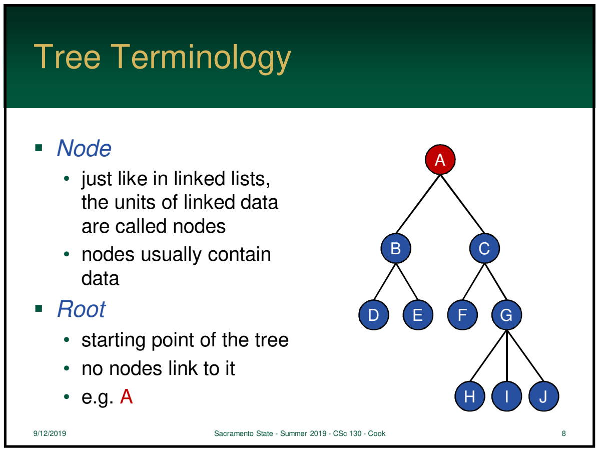
- Trees are recursive data structures
- They can be defines as smaller and smaller instances of trees
- So, using recursion is a natural approach to using them





Linked Lists vs. Trees

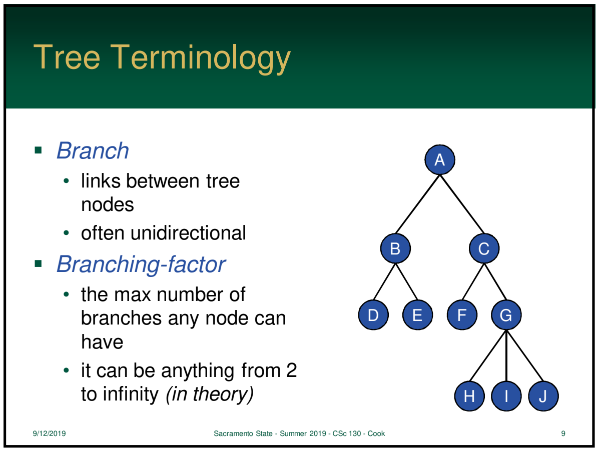
- Linked Lists
  - linear - accessing all elements is O(n)
  - nodes can only have one predecessor and/or one successor node
- Trees
  - nonlinear and hierarchical
  - nodes can have *multiple* successors but only one predecessor



Tree Terminology

- *Node*
  - just like in linked lists, the units of linked data are called nodes
  - nodes usually contain data
- *Root*
  - starting point of the tree
  - no nodes link to it
  - e.g. A

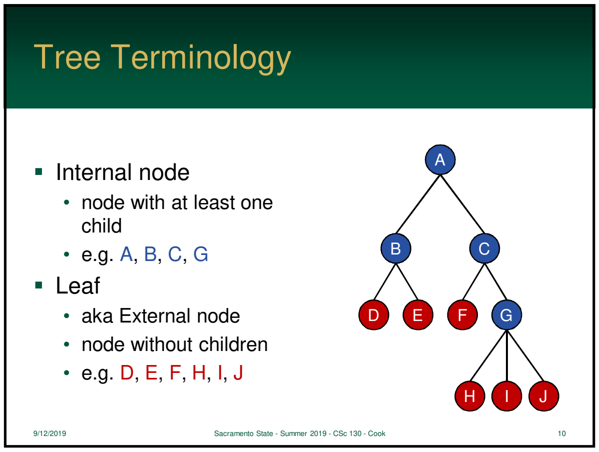
A
B
C
D
E
F
G
H
I
J



Tree Terminology

- *Branch*
  - links between tree nodes
  - often unidirectional
- *Branching-factor*
  - the max number of branches any node can have
  - it can be anything from 2 to infinity *(in theory)*

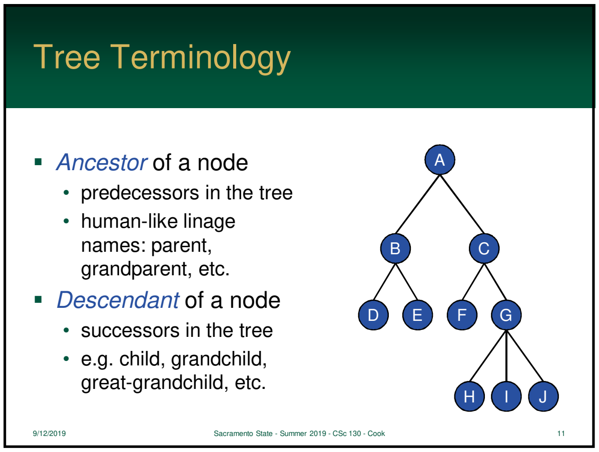
A
B
C
D
E
F
G
H
I
J



Tree Terminology

- Internal node
  - node with at least one child
  - e.g. A, B, C, G
- Leaf
  - aka External node
  - node without children
  - e.g. D, E, F, H, I, J

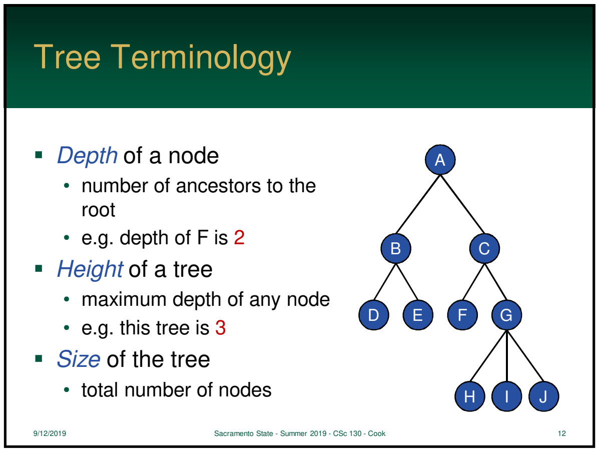
A
B
C
D
E
F
G
H
I
J



Tree Terminology

- *Ancestor* of a node
  - predecessors in the tree
  - human-like linage names: parent, grandparent, etc.
- *Descendant* of a node
  - successors in the tree
  - e.g. child, grandchild, great-grandchild, etc.

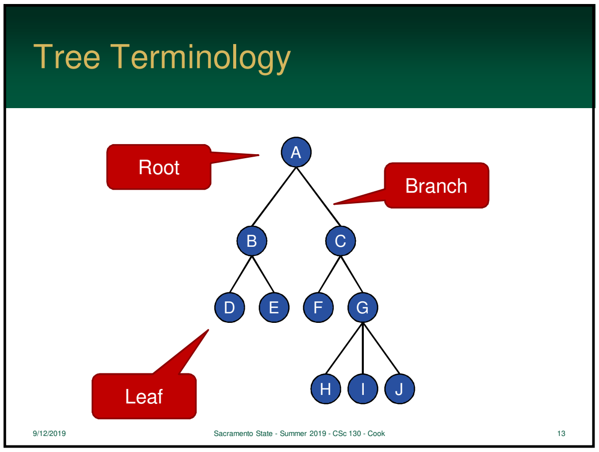
A
B
C
D
E
F
G
H
I
J



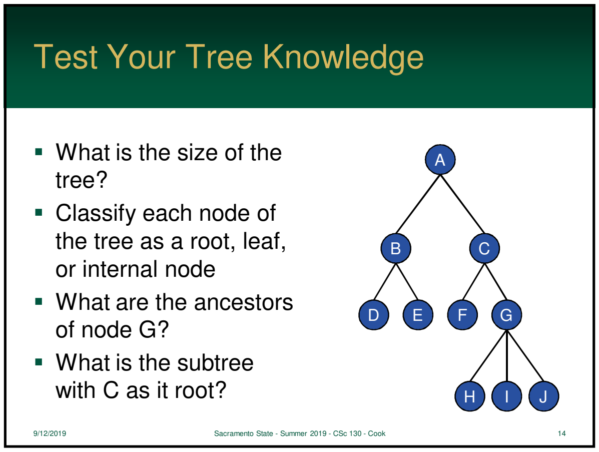
Tree Terminology

- *Depth* of a node
  - number of ancestors to the root
  - e.g. depth of F is 2
- *Height* of a tree
  - maximum depth of any node
  - e.g. this tree is 3
- *Size* of the tree
  - total number of nodes

A
B
C
D
E
F
G
H
I
J

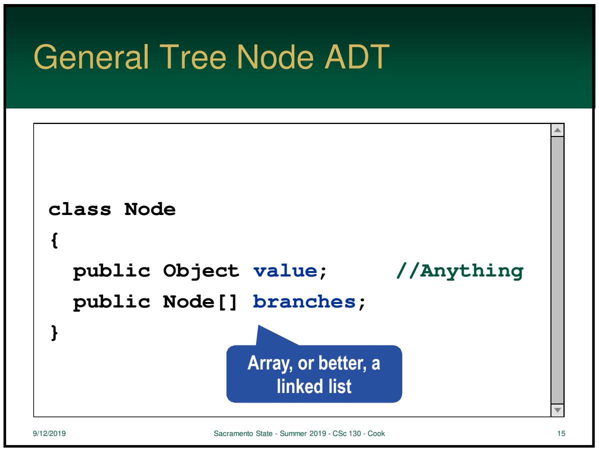


Tree Terminology



Test Your Tree Knowledge

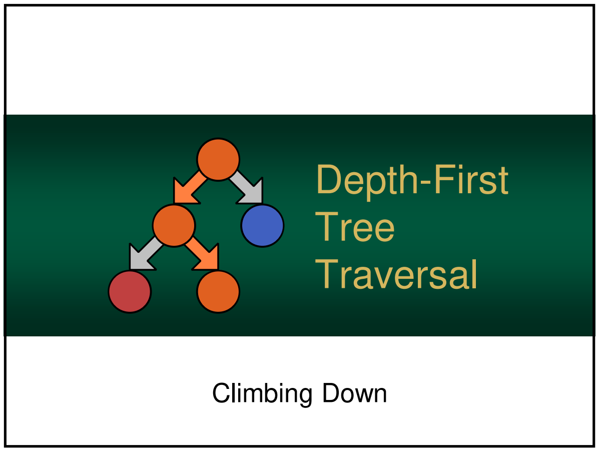
- What is the size of the tree?
- Classify each node of the tree as a root, leaf, or internal node
- What are the ancestors of node G?
- What is the subtree with C as it root?



General Tree Node ADT

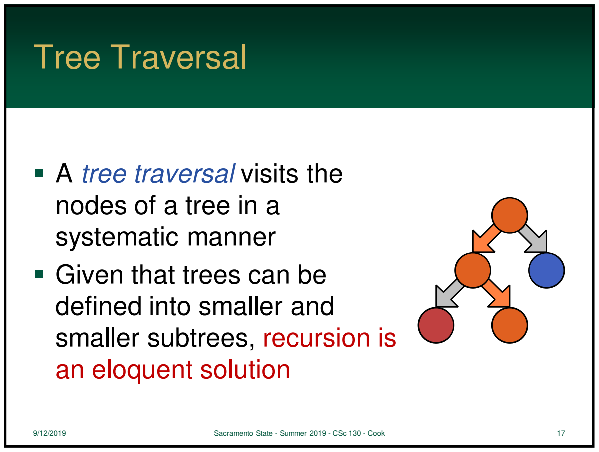
```
class Node
{
  public Object value;      //Anything
  public Node[] branches;
}
```

Array, or better, a linked list



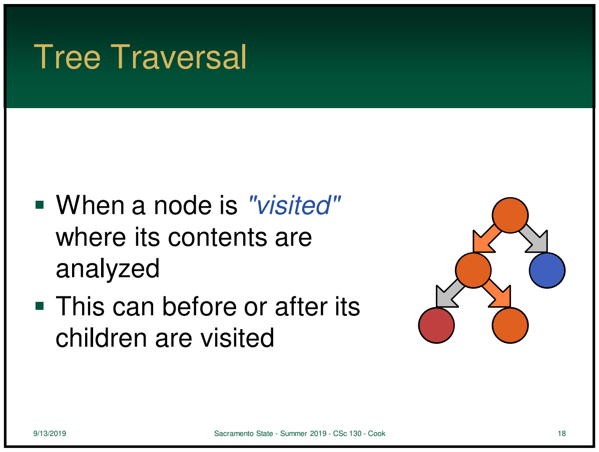
Depth-First Tree Traversal

Climbing Down



Tree Traversal

- A *tree traversal* visits the nodes of a tree in a systematic manner
- Given that trees can be defined into smaller and smaller subtrees, recursion is an eloquent solution



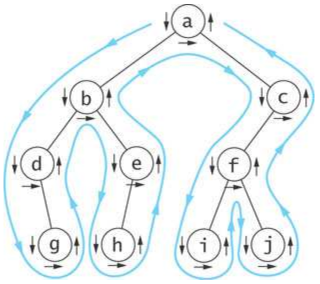
Tree Traversal

- When a node is *"visited"* where its contents are analyzed
- This can before or after its children are visited

Depth-First Transversal

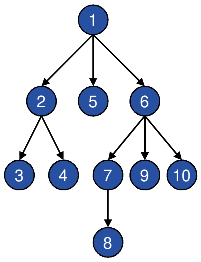
- In depth-first transversal, the algorithm travels down the tree
- So, the algorithm looks at a child and it *then* looks at its children
- This approach lends itself to recursion
- There are several approaches of when a node is "visited"

Depth First Traversal



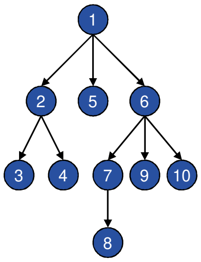
Depth-first: Preorder

- In a *preorder traversal*, a node is visited before its descendants
- Nodes will be visited in the order depicted on tree to the right

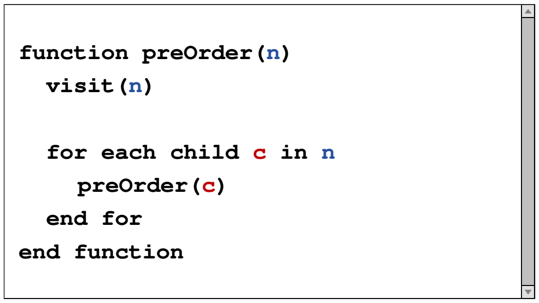


Depth-first: Preorder

- Notice that each child was visited after its parent
- Some applications…
  - print a tree document
  - e.g. XML export



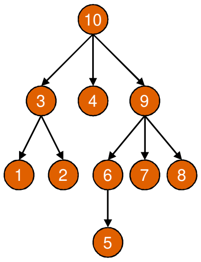
Preorder Traversal Logic

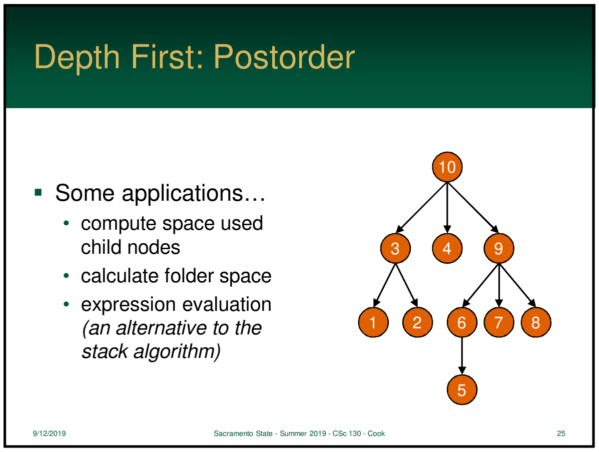


```
function preOrder(n)
  visit(n)

  for each child c in n
    preOrder(c)
  end for
end function
```

Depth First: Postorder

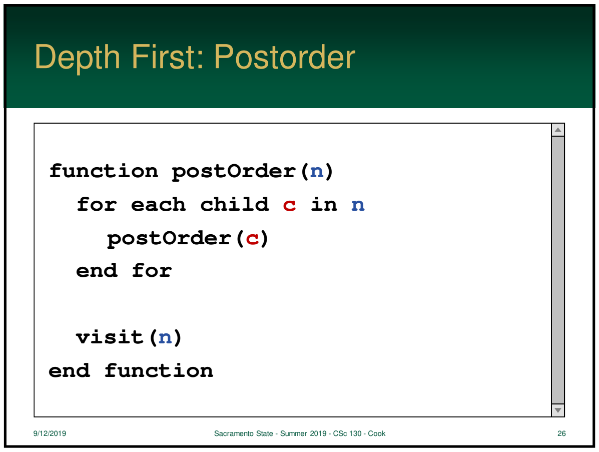
- In a *postorder traversal*, a node is visited after its descendants
- Notice that each child was visited before its parent





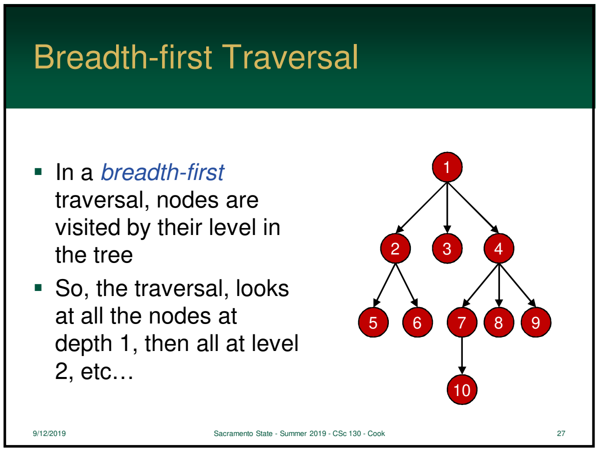
Depth First: Postorder

- Some applications…
  - compute space used child nodes
  - calculate folder space
  - expression evaluation *(an alternative to the stack algorithm)*



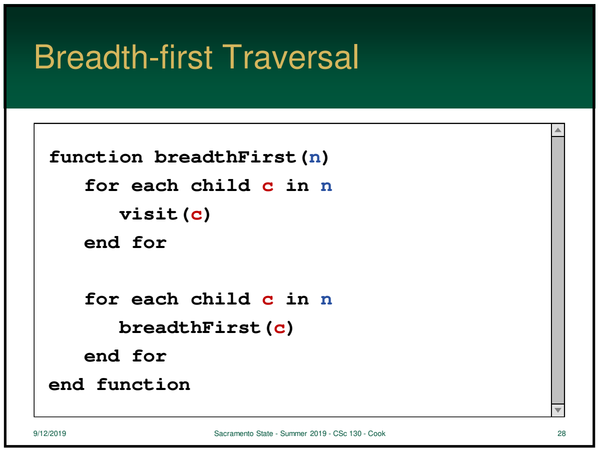
Depth First: Postorder

```
function postOrder(n)
  for each child c in n
     postOrder(c)
  end for

  visit(n)
end function
```



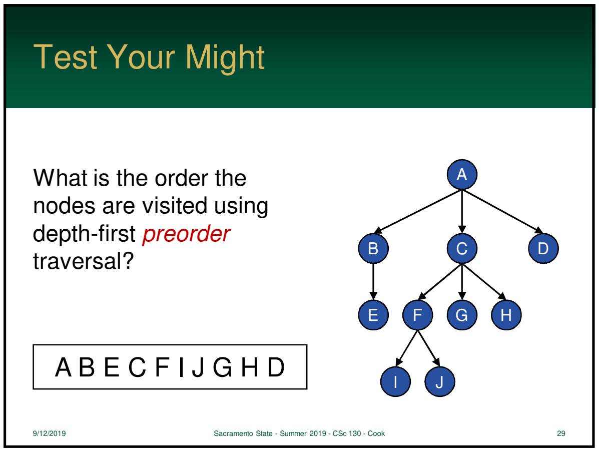
Breadth-first Traversal

- In a *breadth-first* traversal, nodes are visited by their level in the tree
- So, the traversal, looks at all the nodes at depth 1, then all at level 2, etc…



Breadth-first Traversal

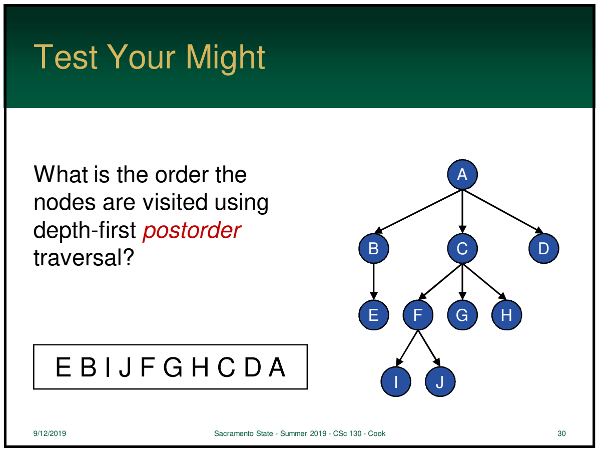
```
function breadthFirst(n)
   for each child c in n
      visit(c)
   end for

   for each child c in n
      breadthFirst(c)
   end for
end function
```



Test Your Might

What is the order the nodes are visited using depth-first *preorder* traversal?

A B E C F I J G H D



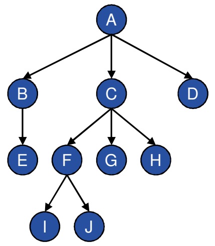
Test Your Might

What is the order the nodes are visited using depth-first *postorder* traversal?

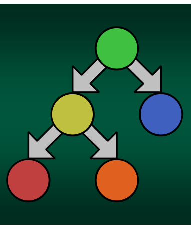
E B I J F G H C D A

Test Your Might

What is the order the nodes are visited using *breadth-first* traversal?



A B C D E F G H I J

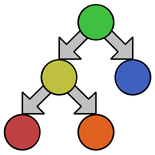


Binary Trees

The Power of Two!

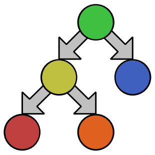
Binary Trees

- The most common tree used in data structures is in the style of the binary tree
- As the name implies, nodes in a binary tree only have two successors



Binary Trees

- We call the children of an internal node *left* child and *right* child
- Binary trees can be represented by arrays and linked data structures

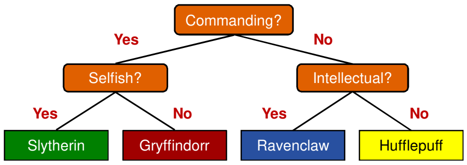


Binary Trees

- Binary Trees are extremely useful in data structures
- The two branches allow for efficient branching and is ideal for binary operations
- Applications:
  - storing arithmetic expressions
  - decision processes
  - searching
  - sorting

Boolean Decision Tree

- Binary tree can be used for decision branching
- internal nodes: questions with yes/no answer
- leaves: decisions



Commanding?

Yes

No

Selfish?

Intellectual?

Yes

No

Yes

No

Slytherin

Gryffindorr

Ravenclaw

Hufflepuff

Binary Tree Node

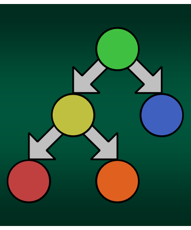
```
class Node
{
  public Object value; //Can be anything
  public Node left;
  public Node right;
}
```

Branches are much simpler

Attributes of a Binary Tree

- $v = i + 1$
- $n = 2v - 1$
- $h \le i$
- $h \le (n - 1) / 2$
- $v \le 2h$
- $h \ge \log_2 v$
- $h \ge \log_2 (n + 1) - 1$

n	number of nodes
i	number of internal nodes
v	number of leaves
h	height of the tree

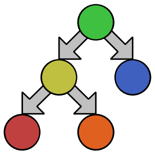


Depth-First Traversing Binary Trees

With simplicity, we have power!

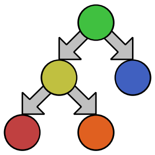
Depth-First Traversing

- Because of the simplicity of binary trees, we have a very useful structure for tree traversal
- We can only traverse left and right
- This gives three possibilities for a depth first search



Post-order Depth-first Traversal

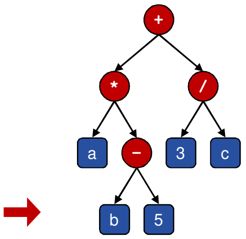
- In an *post-order traversal* a node is evaluated after its left branch and after its right branch
- In other words: recurse left, recurse right, then do something

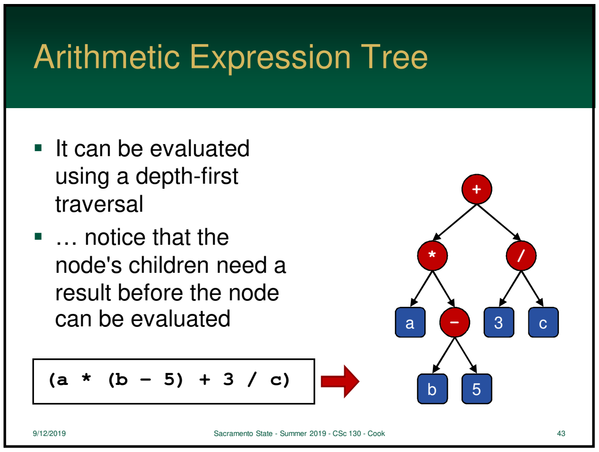


Arithmetic Expression Tree

- Expressions can be represented with a tree
- How?
  - internal nodes: operators
  - leaves: operand

```
(a * (b − 5) + 3 / c)
```

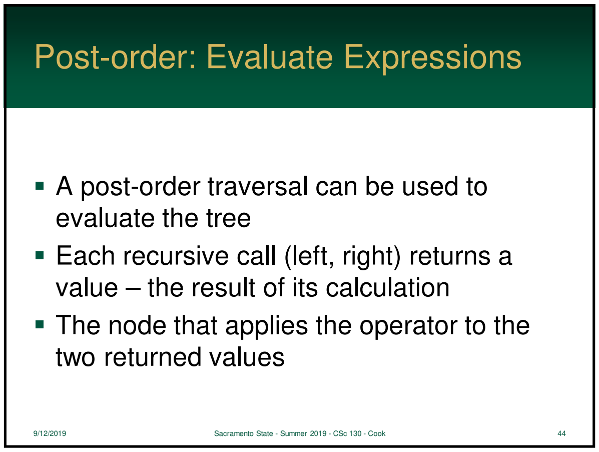




Arithmetic Expression Tree

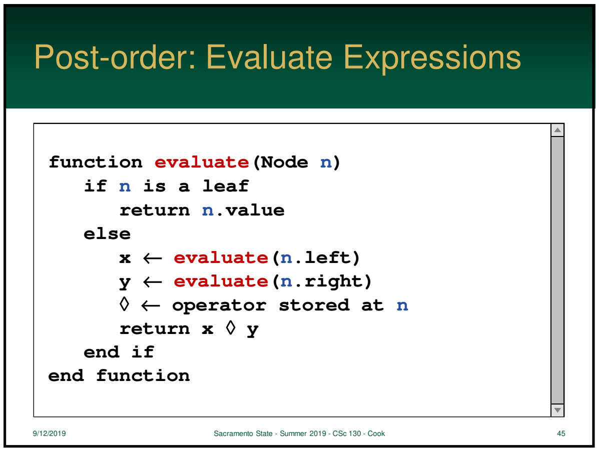
- It can be evaluated using a depth-first traversal
- … notice that the node's children need a result before the node can be evaluated

```
(a * (b – 5) + 3 / c)
```



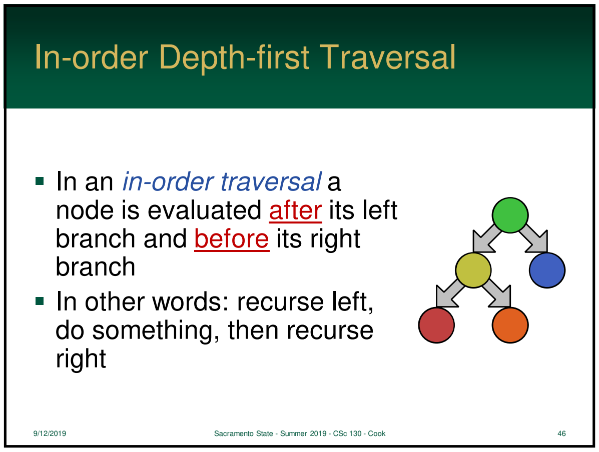
Post-order: Evaluate Expressions

- A post-order traversal can be used to evaluate the tree
- Each recursive call (left, right) returns a value – the result of its calculation
- The node that applies the operator to the two returned values



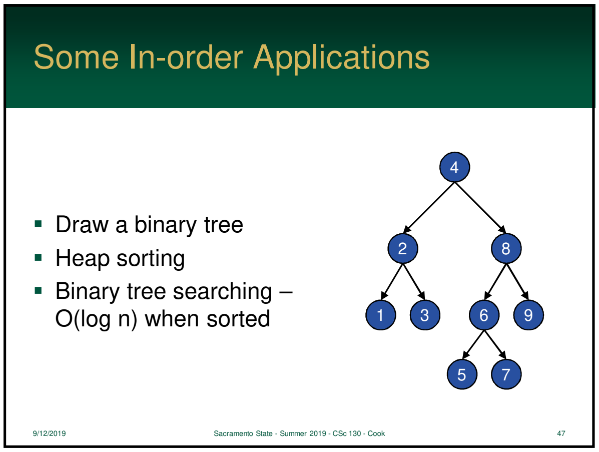
Post-order: Evaluate Expressions

```
function evaluate(Node n)
   if n is a leaf
      return n.value
   else
      x ← evaluate(n.left)
      y ← evaluate(n.right)
      ◊ ← operator stored at n
      return x ◊ y
   end if
end function
```



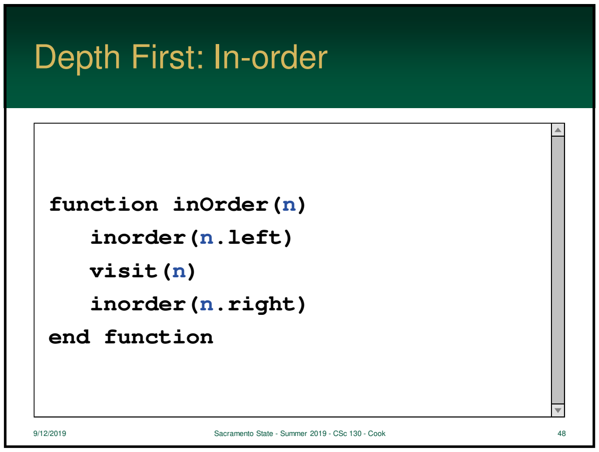
In-order Depth-first Traversal

- In an *in-order traversal* a node is evaluated after its left branch and before its right branch
- In other words: recurse left, do something, then recurse right



Some In-order Applications

- Draw a binary tree
- Heap sorting
- Binary tree searching – O(log n) when sorted

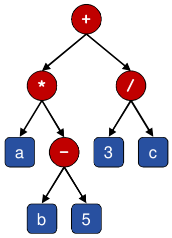


Depth First: In-order

```
function inOrder(n)
   inorder(n.left)
   visit(n)
   inorder(n.right)
end function
```

In-order: Print Expressions

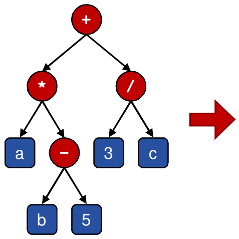
- Inorder can be used to easily print an expression stored in a tree
- Print….
  - "(" before traversing left
  - the node's operator
  - ")" after traversing right



In-order: Print Expressions

```
function print(Node n)
   if n is a leaf
      write n.value
   else
      write "("
      print(n.left)
      write n.operator
      print(n.right)
      write ")"
   end if
end function
```

In-order: Print Expressions



```
((a * (b – 5)) + (3 / c))
```

Pre-order Depth-first Traversal

- When a *pre-order* depth-first traversal is performed, the node is evaluated before the right or left child
- This is useful for copying a tree, but not much more