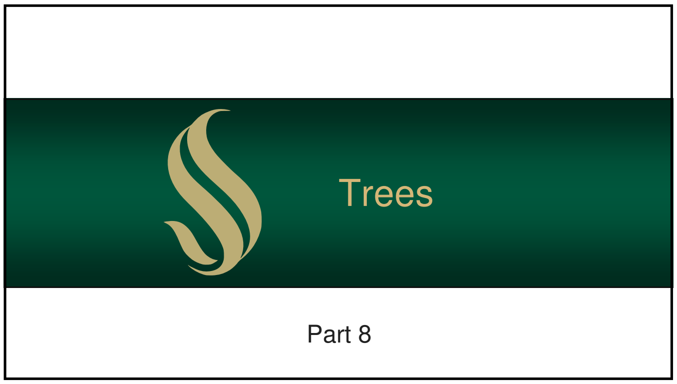
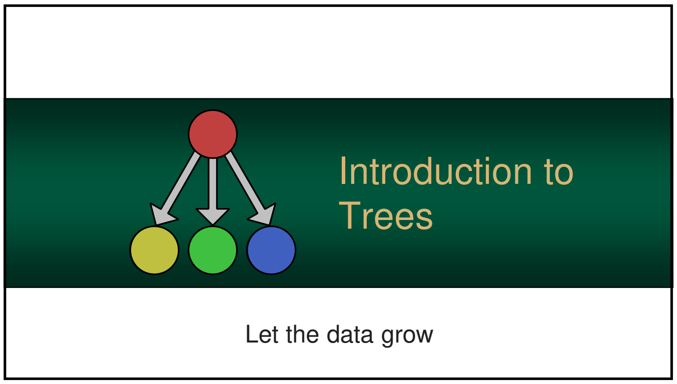


Trees

Part 8

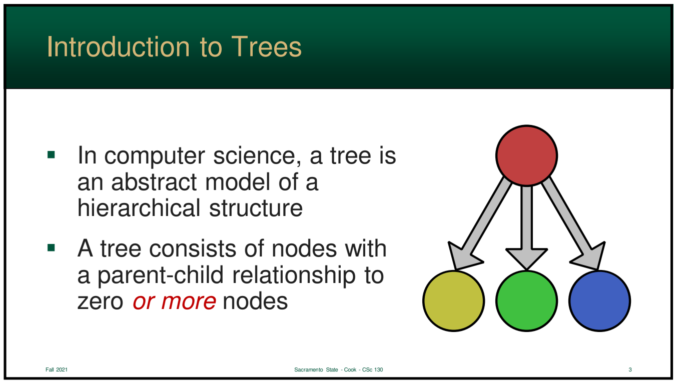
1



Introduction to Trees

Let the data grow

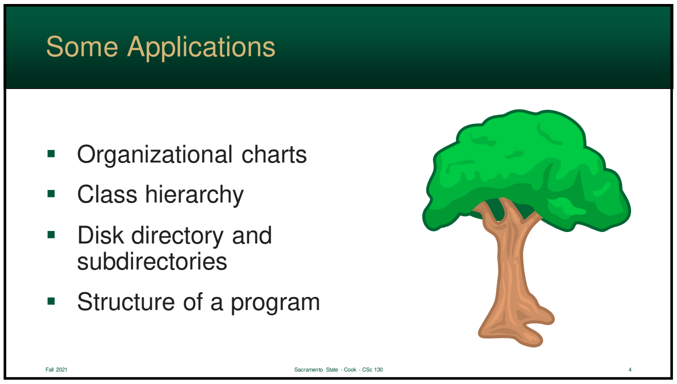
2



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

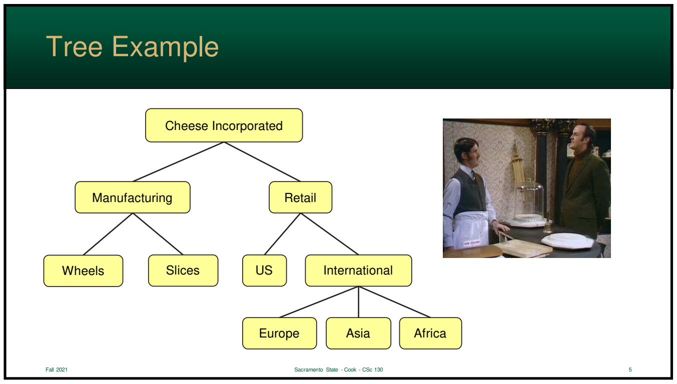
3



Some Applications

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

4



Tree Example

Cheese Incorporated

Manufacturing

Retail

Wheels

Slices

US

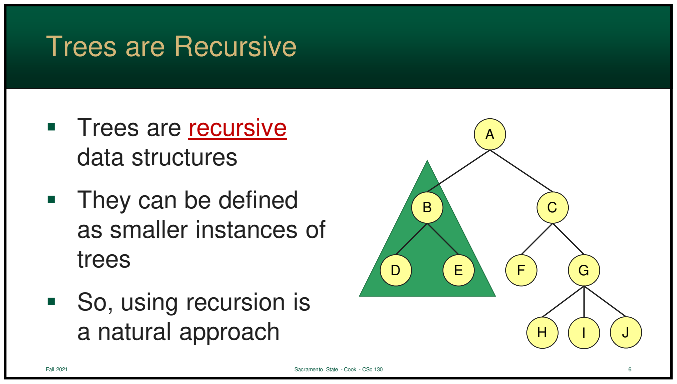
International

Europe

Asia

Africa

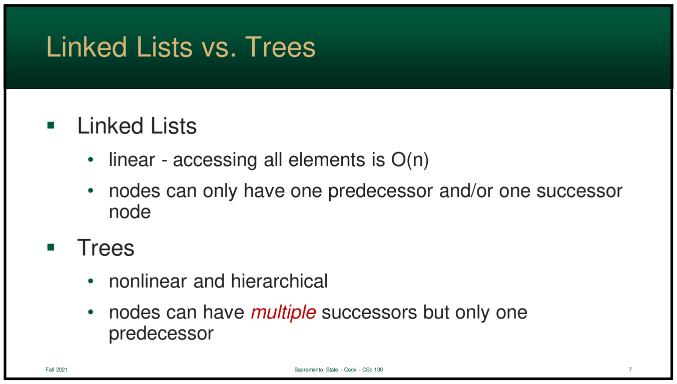
5



Trees are Recursive

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

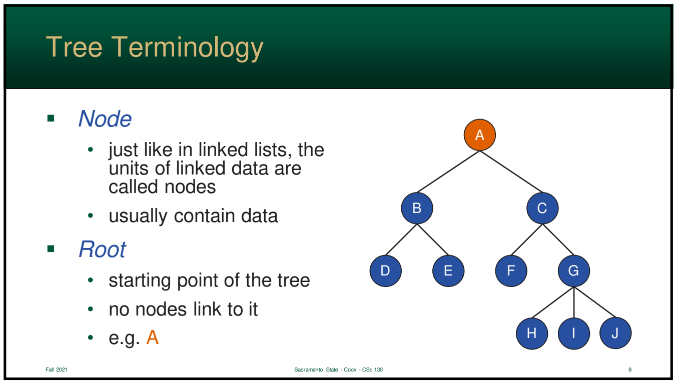
6



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

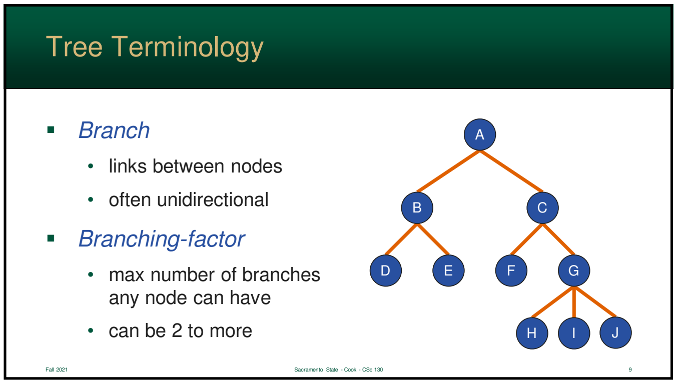
7



Tree Terminology

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

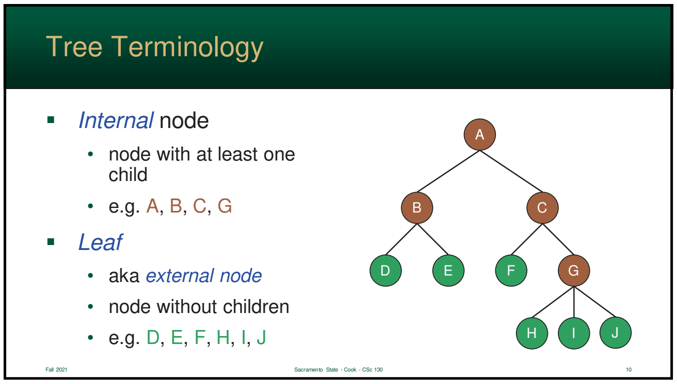
8



Tree Terminology

- *Branch*
  - links between nodes
  - often unidirectional
- *Branching-factor*
  - max number of branches any node can have
  - can be 2 to more

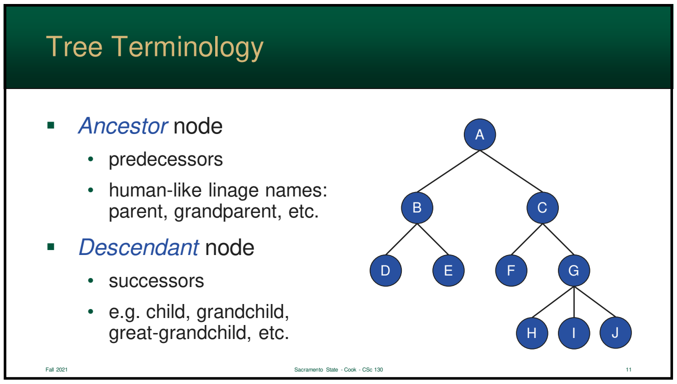
9



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

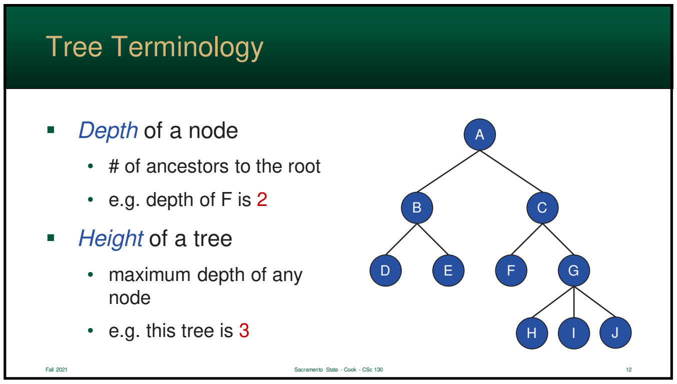
10



Tree Terminology

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

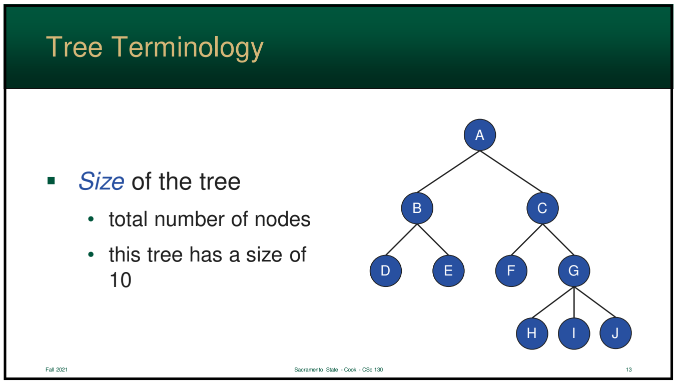
11



Tree Terminology

- *Depth* of a node
  - # 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

12

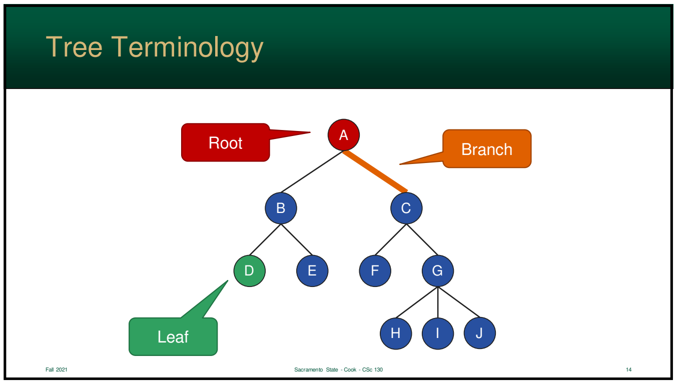


Tree Terminology

- *Size* of the tree
  - total number of nodes
  - this tree has a size of 10

A
B
C
D
E
F
G
H
I
J

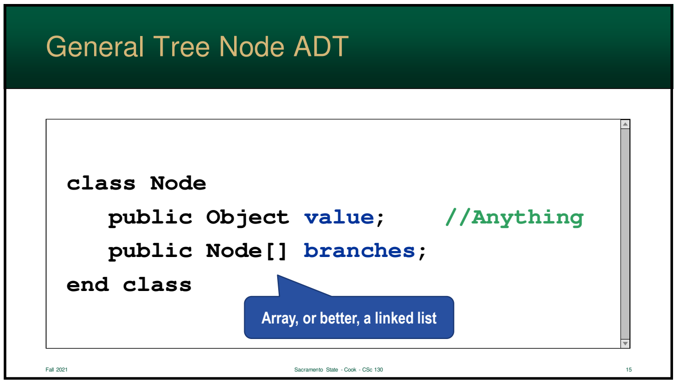
13



Tree Terminology

Root
A
Branch
B
C
D
E
F
G
Leaf
H
I
J

14

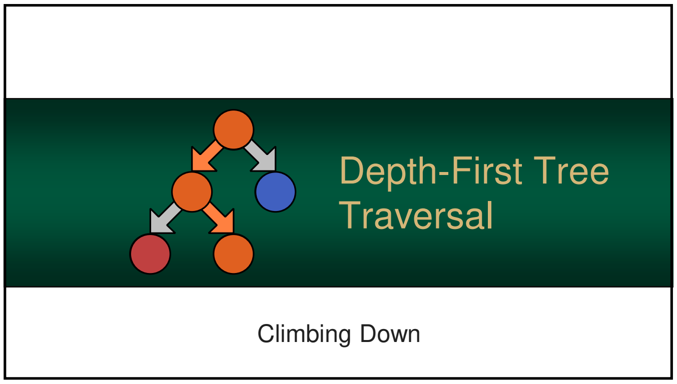


General Tree Node ADT

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

Array, or better, a linked list

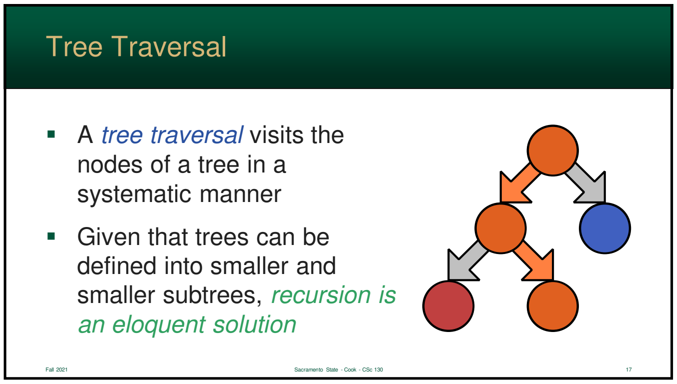
15



Depth-First Tree Traversal

Climbing Down

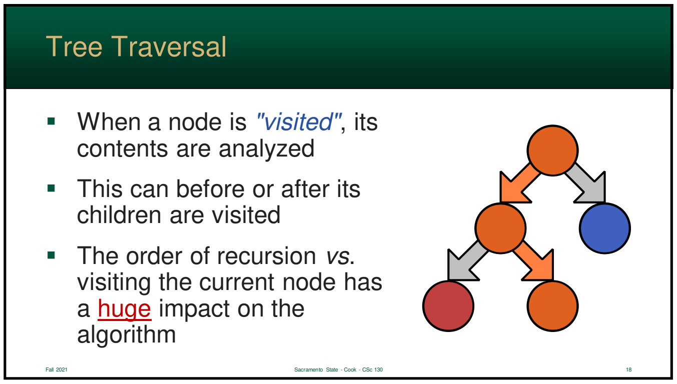
16



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*

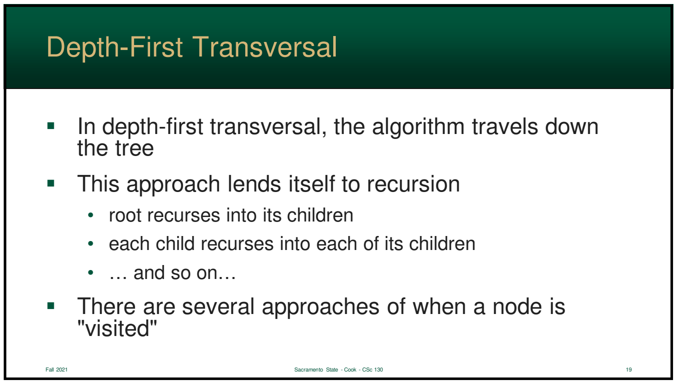
17



Tree Traversal

- When a node is *"visited"*, its contents are analyzed
- This can before or after its children are visited
- The order of recursion *vs.* visiting the current node has a huge impact on the algorithm

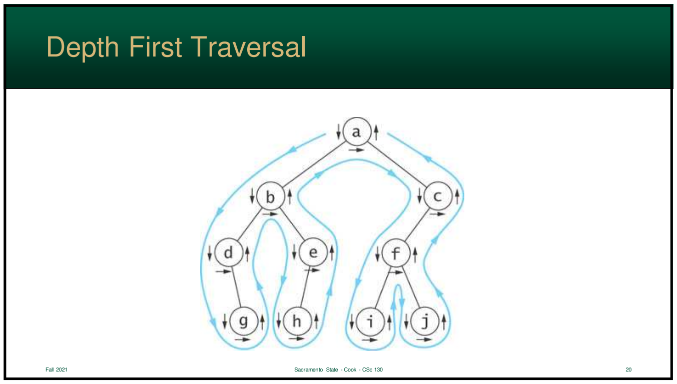
18



Depth-First Transversal

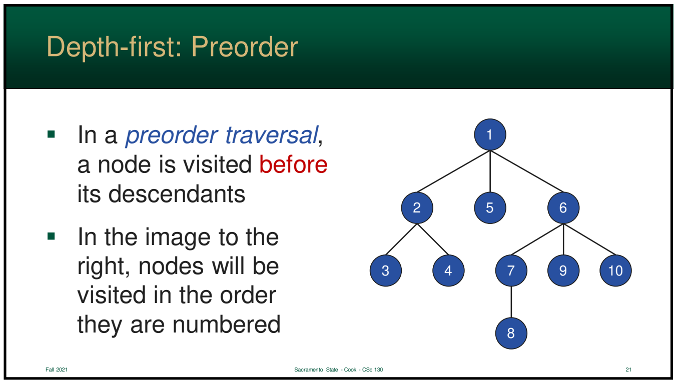
- In depth-first transversal, the algorithm travels down the tree
- This approach lends itself to recursion
  - root recurses into its children
  - each child recurses into each of its children
  - … and so on…
- There are several approaches of when a node is "visited"

19



Depth First Traversal

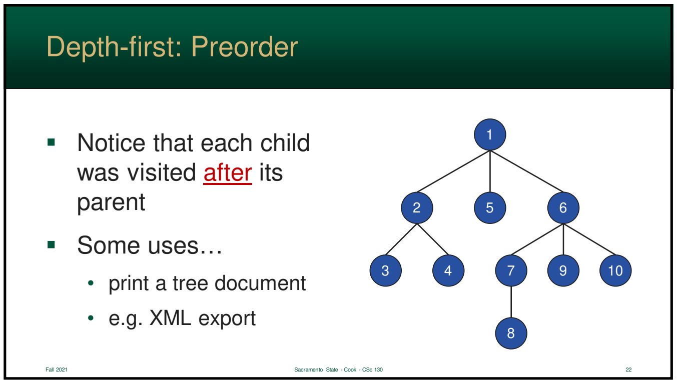
20



Depth-first: Preorder

- In a *preorder traversal*, a node is visited before its descendants
- In the image to the right, nodes will be visited in the order they are numbered

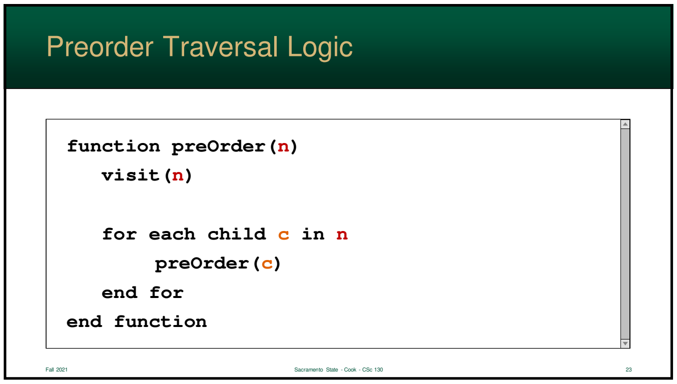
21



Depth-first: Preorder

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

22

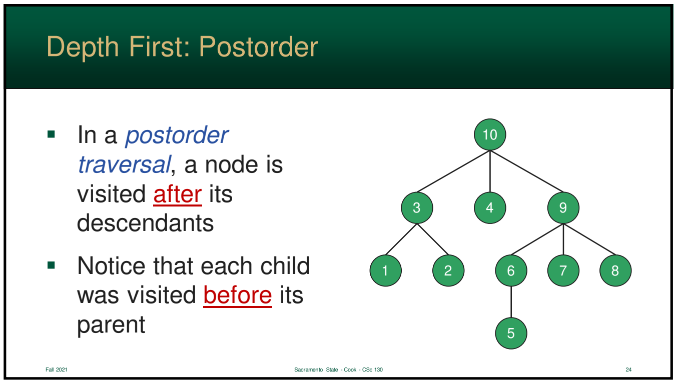


Preorder Traversal Logic

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

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

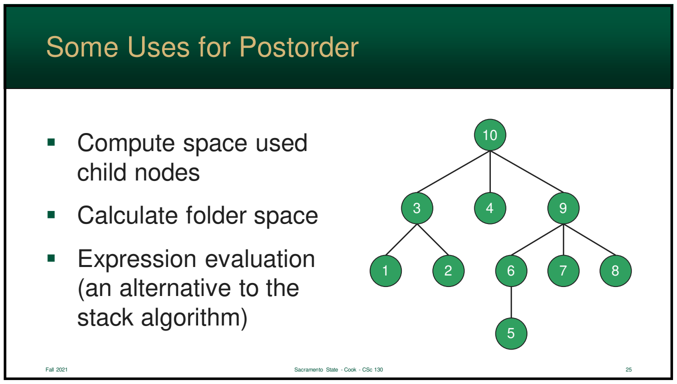
23



Depth First: Postorder

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

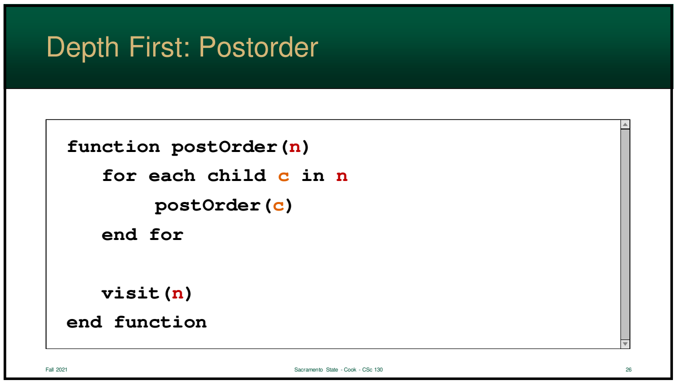
24



Some Uses for Postorder

- Compute space used child nodes
- Calculate folder space
- Expression evaluation (an alternative to the stack algorithm)

25

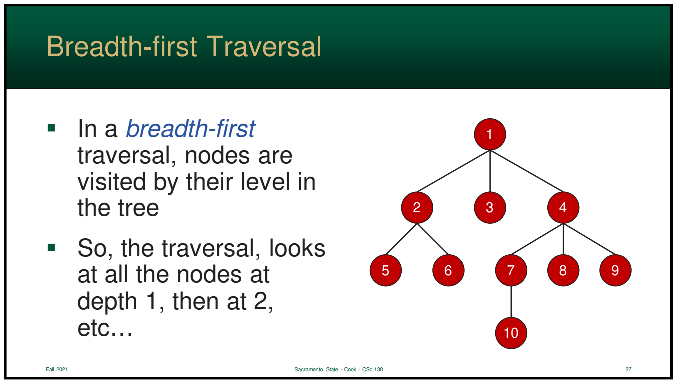


Depth First: Postorder

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

   visit(n)
end function
```

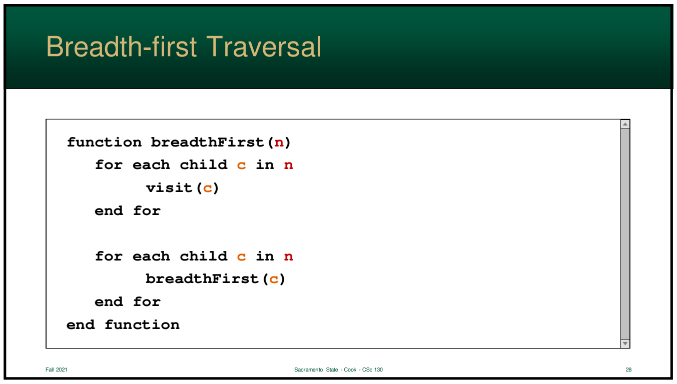
26



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 at 2, etc…

27

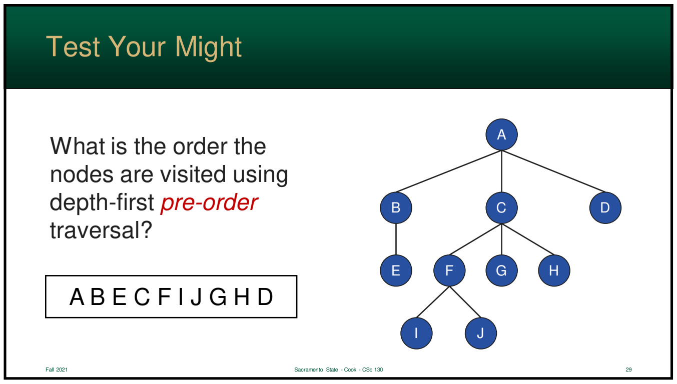


Breadth-first Traversal

```
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
```

28

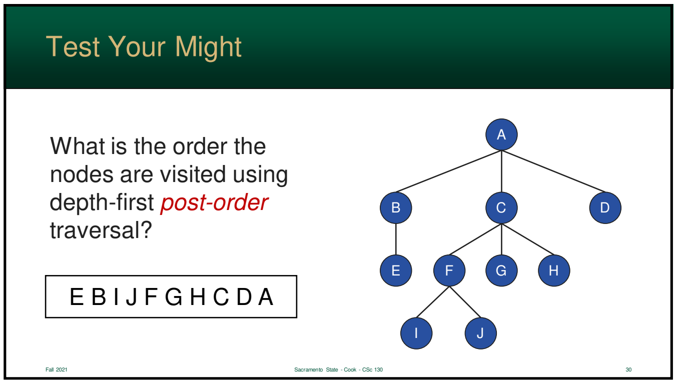


Test Your Might

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

A B E C F I J G H D

29

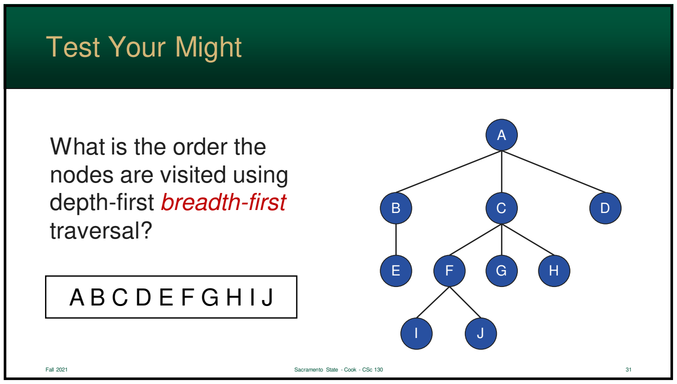


Test Your Might

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

E B I J F G H C D A

30

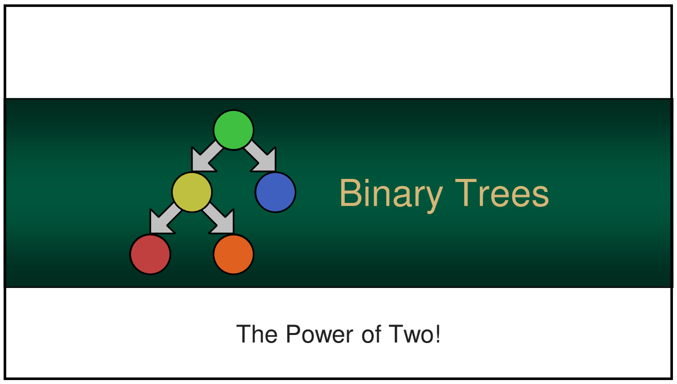


Test Your Might

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

A B C D E F G H I J

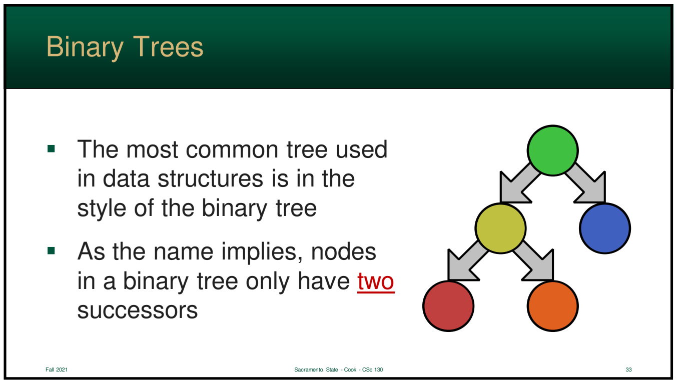
31



Binary Trees

The Power of Two!

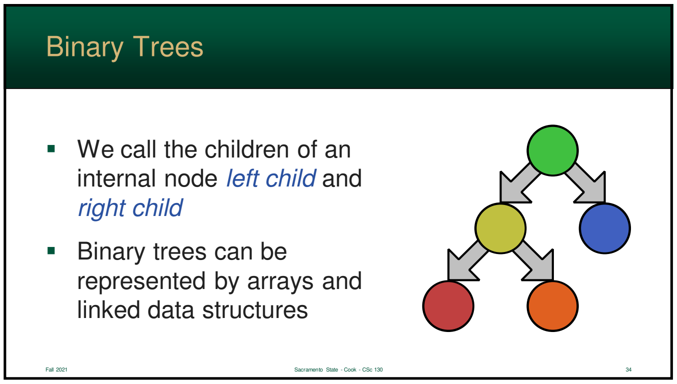
32



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

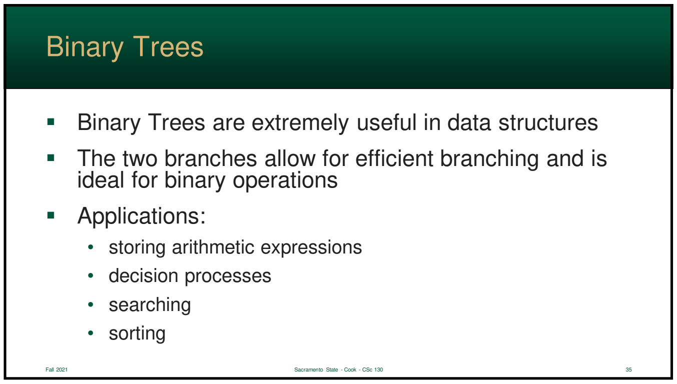
33



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

34



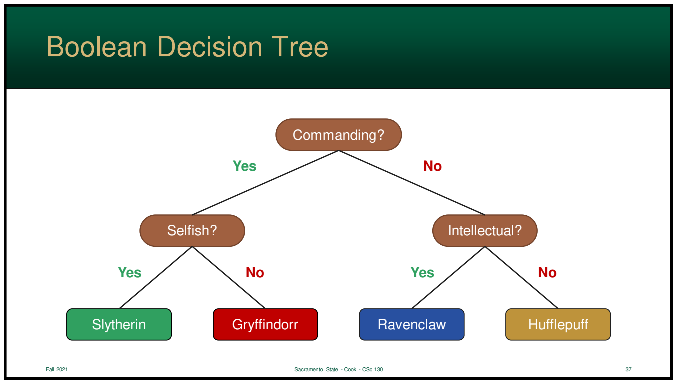
Binary Trees

- 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

35

Boolean Decision Tree

36



Boolean Decision Tree

Commanding?

Yes

No

Selfish?

Intellectual?

Yes

No

Yes

No

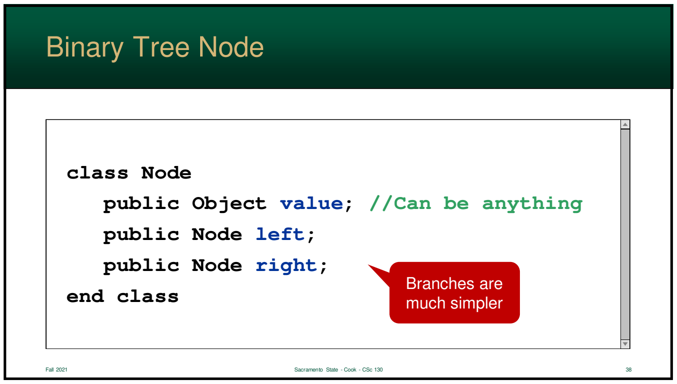
Slytherin

Gryffindorr

Ravenclaw

Hufflepuff

37

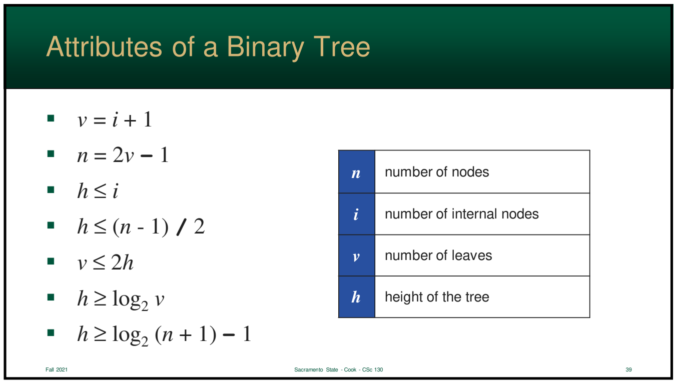


Binary Tree Node

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

Branches are much simpler

38

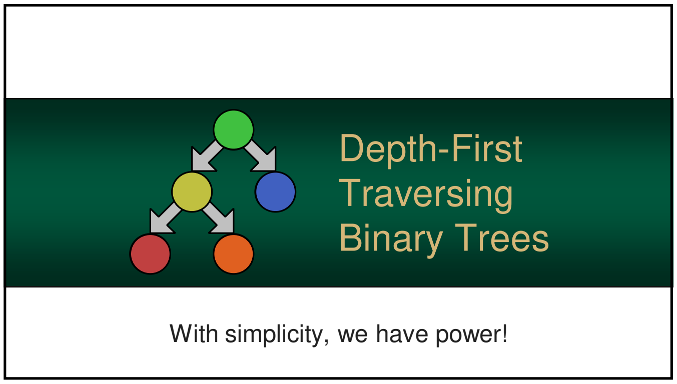


Attributes of a Binary Tree

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

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

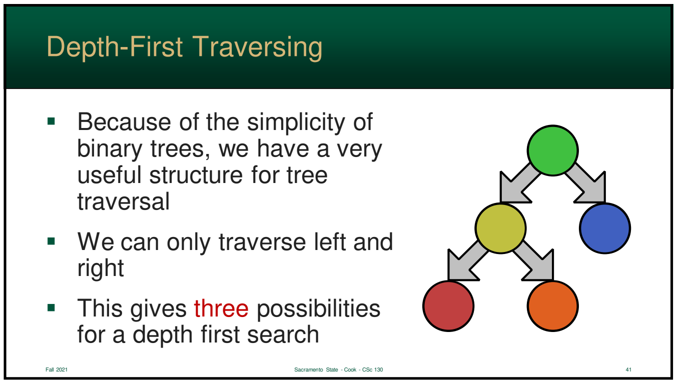
39



Depth-First Traversing Binary Trees

With simplicity, we have power!

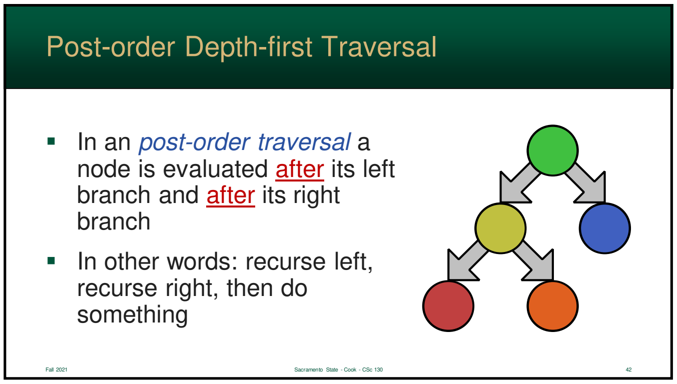
40



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

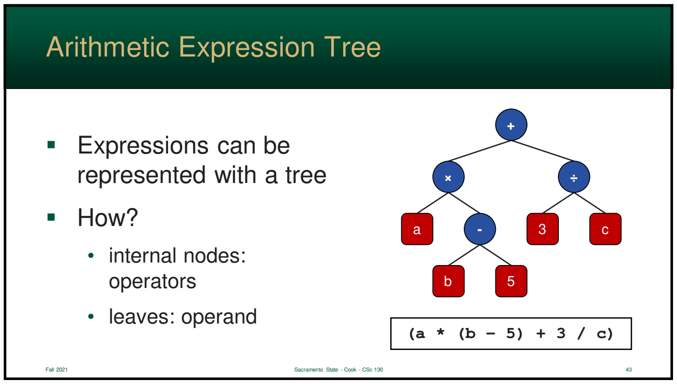
41



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

42

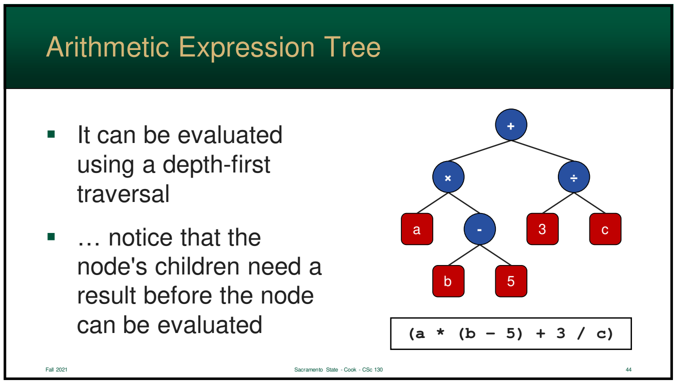


Arithmetic Expression Tree

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

(a * (b – 5) + 3 / c)

43

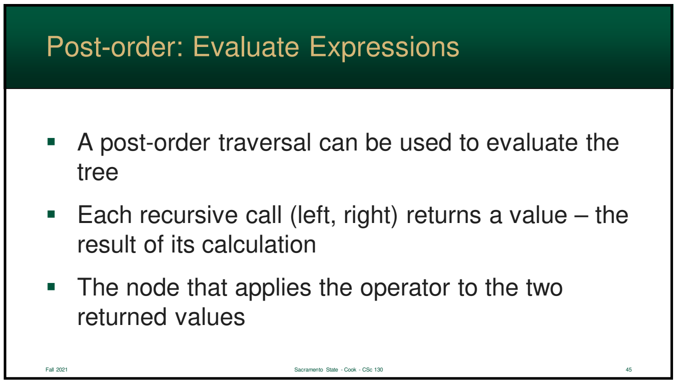


Arithmetic Expression Tree

- 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)

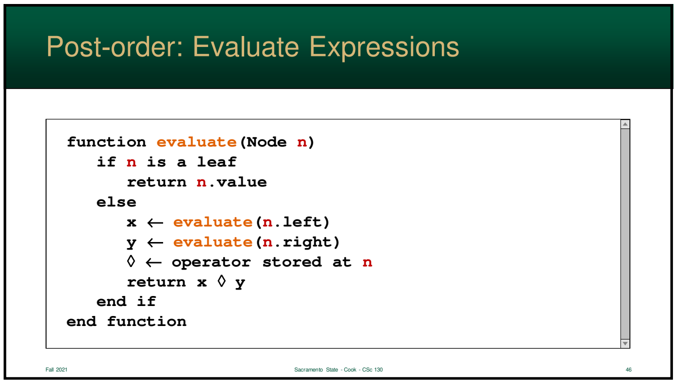
44



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

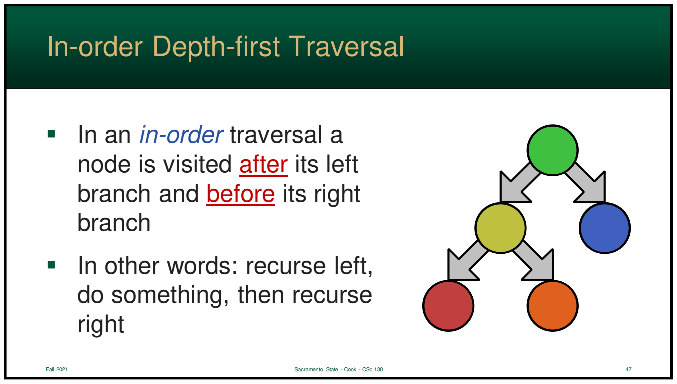
45



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
```

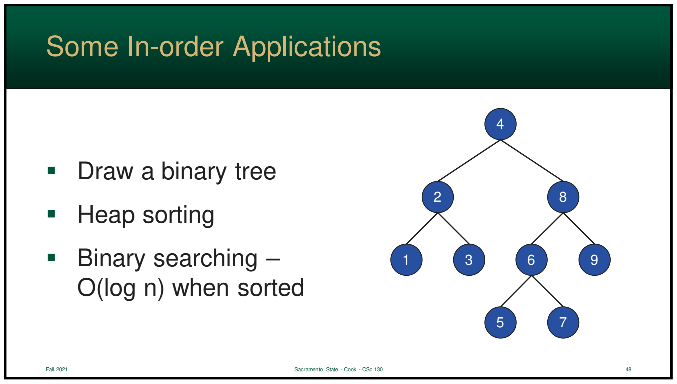
46



In-order Depth-first Traversal

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

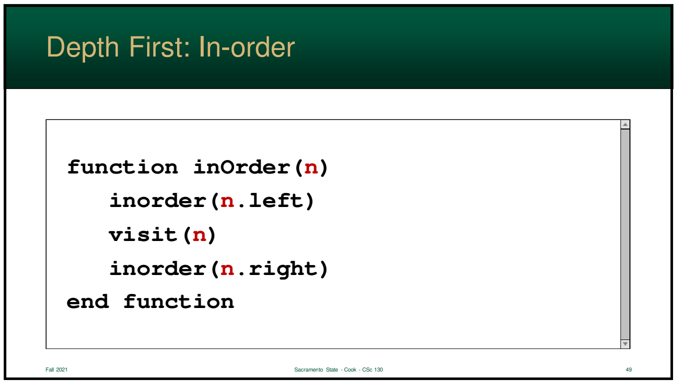
47



Some In-order Applications

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

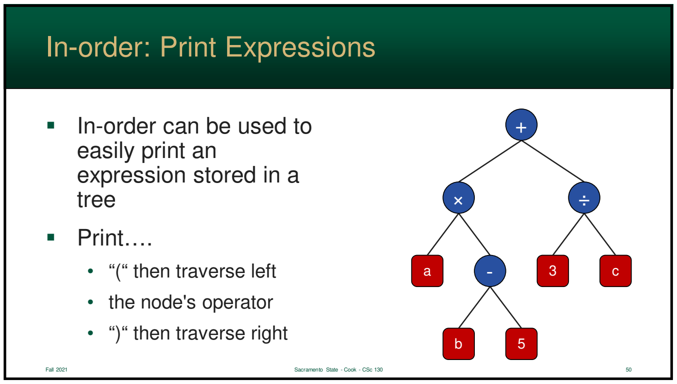
48



Depth First: In-order

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

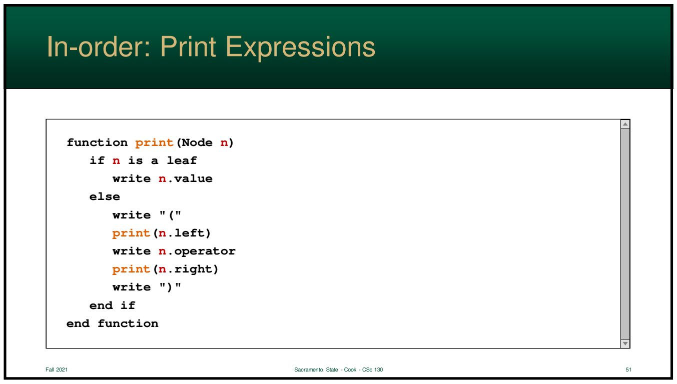
49



In-order: Print Expressions

- In-order can be used to easily print an expression stored in a tree
- Print….
  - "(" then traverse left
  - the node's operator
  - ")" then traverse right

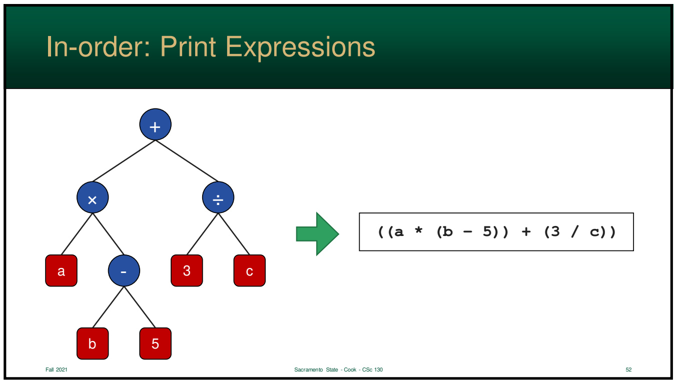
50



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
```

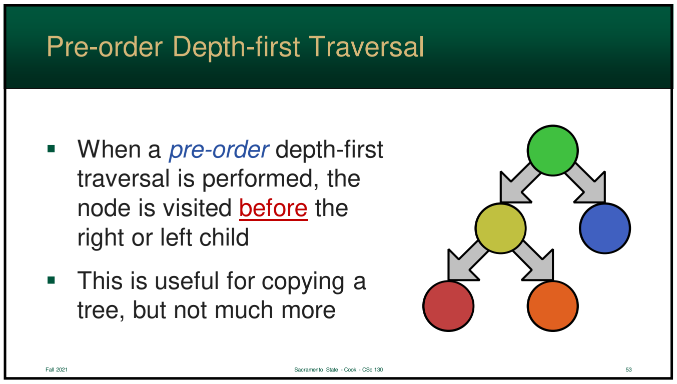
51



In-order: Print Expressions

```
((a * (b – 5)) + (3 / c))
```

52



Pre-order Depth-first Traversal

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

53