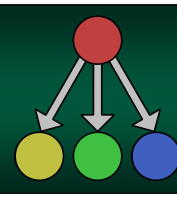
Trees

Part 8

1



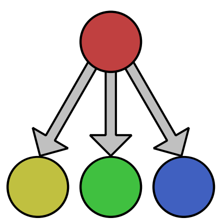
Introduction to Trees

Let the data grow

2

Introduction to Trees

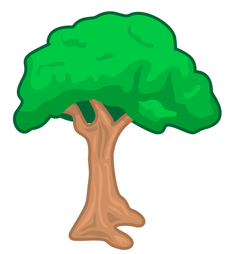
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relationship to zero *or more* nodes



3

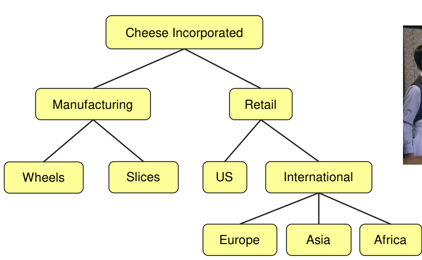
Some Applications

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



4

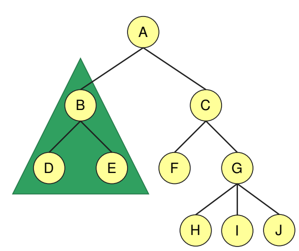
Tree Example



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

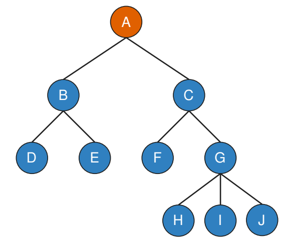
Linked Lists vs. Trees

- 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

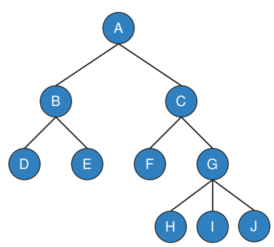
- *Node*
  - just like in linked lists, the units of linked data are called nodes
  - usually contain data
- *Root*
  - starting point of the tree
  - no nodes link to it
  - e.g. A



8

Tree Terminology

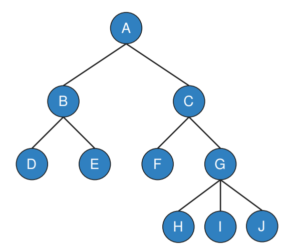
- *Ancestor* node
  - predecessors
  - human-like linage names: parent, grandparent, etc.
- *Descendant* node
  - successors
  - e.g. child, grandchild, great-grandchild, etc.



9

Tree Terminology

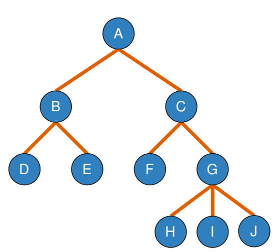
- *Depth* of a node
  - # of ancestors to the root
  - e.g. depth of F is 2
- *Height* of a tree
  - maximum depth of any node
  - e.g. this tree is 3



10

Tree Terminology

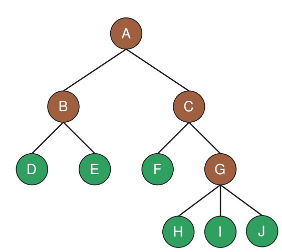
- *Branch*
  - links between nodes
  - often unidirectional
- *Branching-factor*
  - max number of branches any node can have
  - can be 2 to more



11

Tree Terminology

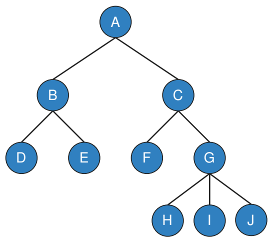
- *Internal* node
  - node with at least one child
  - e.g. A, B, C, G
- *Leaf*
  - aka *external node*
  - node without children
  - e.g. D, E, F, H, I, J



12

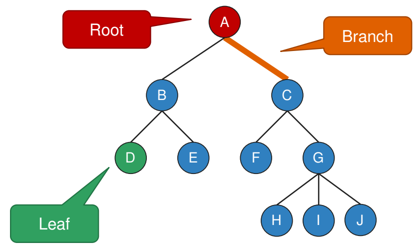
Tree Terminology

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



13

Tree Terminology



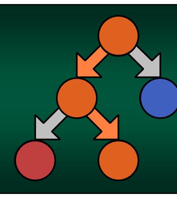
14

General Tree Node ADT

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

Array, or better, a linked list

15



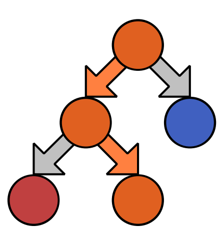
Tree Traversals

Climbing Down

16

Tree Traversal

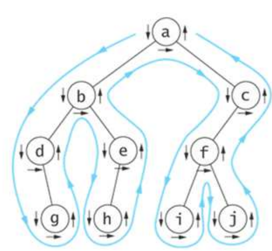
- A *tree traversal* visits the nodes of a tree in a systematic manner
- Given that trees can be defined into smaller and smaller subtrees, *recursion is an eloquent solution*



17

Depth First Traversal

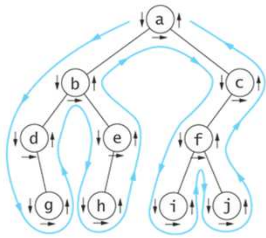
- If we continuously follow the tree to the left – this will result in *Euler Tour*
- We traverse the tree and pass through each node



18

Depth First Traversal

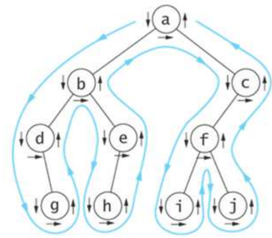
- Notice, in this case, that we tend to go do the bottom first
- This is also known *depth-first traversal*



19

Depth First Traversal

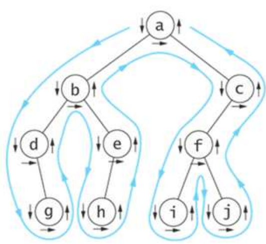
- A node is *visited* when its contents are analyzed
- Notice that we pass by each node going down and going up
- On either of these passes, we can visit the node



20

Depth First Traversal

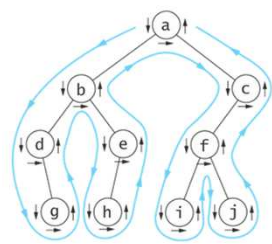
- This approach lends itself to recursion
- How?
  - root recurses into its children
  - each child recurses into each of its children



21

Depth First Traversal

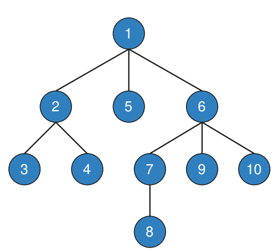
- This can before or after its children are visited
- When the node is visited, when recursing the tree, has a huge impact on the algorithm



22

Depth-first: Preorder

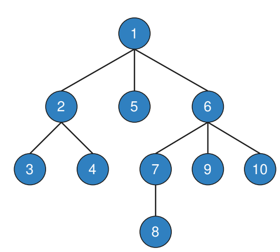
- In a *preorder traversal*, a node is visited before its descendants
- In the image to the right, nodes will be visited in the order they are numbered



23

Depth-first: Preorder

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



24

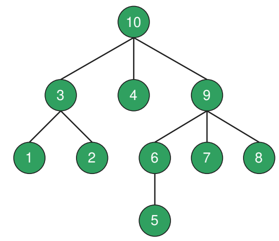
Preorder Traversal Logic

```
function preOrder
   this.visit()

   for each child c in this node
       c.preOrder()
   end for
end function
```


25

Depth First: Postorder

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



26

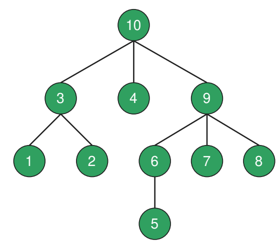
Depth First: Postorder

```
function postOrder
   for each child c in this node
       c.postOrder()
   end for

   this.visit()
end function
```


27

Some Uses for Postorder

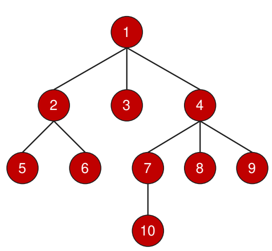
- Compute space used child nodes
- Calculate folder space
- Expression evaluation (an alternative to the stack algorithm)



28

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…

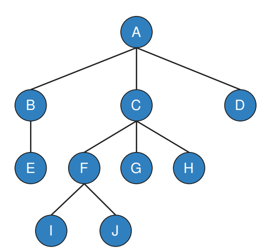


29

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

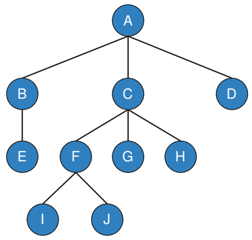


30

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

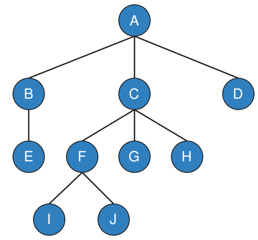


31

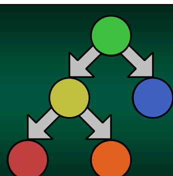
Test Your Might

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

A B C D E F G H I J



32



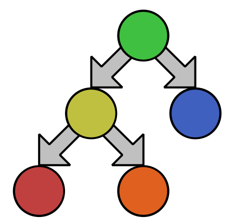
Binary Trees

The Power of Two!

33

Binary Trees

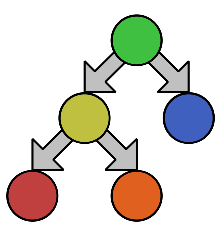
- The most common tree used in data structures is in the style of the binary tree
- As the name implies, nodes in a binary tree only have two successors



34

Binary Trees

- We call the children of an internal node *left child* and *right child*



35

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

36

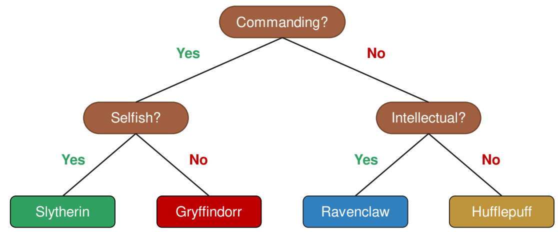
Binary Tree Node

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

Branches are much simpler

37

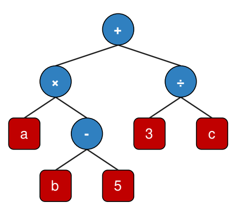
Boolean Decision Tree



38

Arithmetic Expression Tree

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

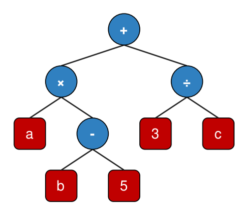



```
(a * (b − 5) + 3 / c)
```


39

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)
```

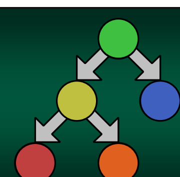

40

Attributes of a Binary Tree

- $v = i + 1$
- $n = 2v - 1$
- $h \leq i$
- $h \leq (n - 1) / 2$
- $v \leq 2h$
- $h \geq \log_2 v$
- $h \geq \log_2 (n + 1) - 1$

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

41



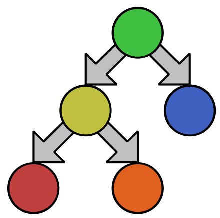
Depth-First Traversing Binary Trees

With simplicity, we have power!

42

Depth-First Traversing

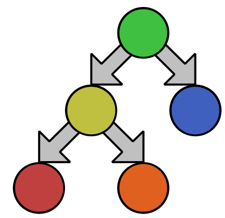
- Because of the simplicity of binary trees, we have a very useful structure for tree traversal
- We can only traverse left and right
- This gives three possibilities for a depth first search



43

Pre-order Depth-first Traversal

- When a *pre-order* depth-first traversal is performed, the node is visited before the right or left child
- Useful for copying a tree and printing trees



44

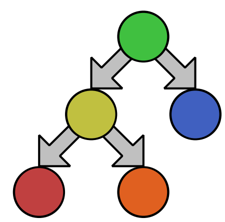
Binary Pre-order Traversal Logic

```
function preOrder
    this.visit()

    if left  isn't null then left.preOrder()
    if right isn't null then right.preOrder()
end function
```


45

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, visit, then recurse right



46

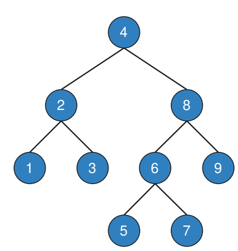
Binary In-order Traversal Logic

```
function inOrder
    if left  isn't null then left.inOrder()

    this.visit()

    if right isn't null then right.inOrder()
end function
```


47

Some In-order Applications

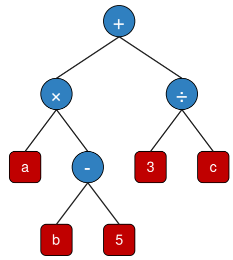
- Draw a binary tree
- Heap sorting
- Binary searching – O(log n) when sorted



48

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
  - traverse right then)



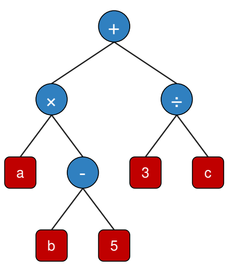
49

In-order: Print Expressions

```
function print()
    if this is a leaf
        write this.value
    else
        write "("
        left.print()
        write this.operator    …can be stored in this.value
        right.print()
        write ")"
    end if
end function
```


50

In-order: Print Expressions

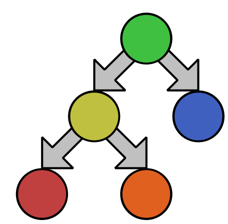



```
((a * (b – 5)) + (3 / c))
```


51

Post-order Depth-first Traversal

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



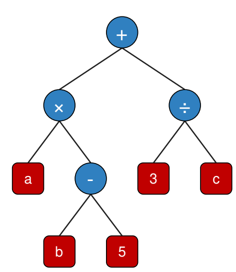
52

Binary Post-order Traversal Logic

```
function postOrder
    if left  isn't null then left.postOrder()
    if right isn't null then right.postOrder()

    this.visit()
 end function
```


53

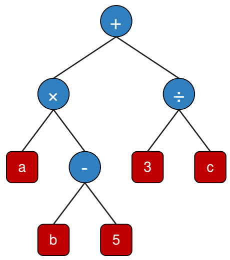
Post-order: Evaluate Expressions



- A post-order traversal can be used to evaluate the tree
- Each recursive call (left, right) returns a value – the result of its calculation

54

Post-order: Evaluate Expressions



- The node then applies the operator to the two returned values (left, right)
- … and then returns that value to its caller

55

Post-order: Evaluate Expressions

```
function evaluate()
   if this is a leaf
      return this.value
   else
      x ← left.evaluate()
      y ← right.evaluate()
      ◊ ← this.operator   …can be stored in this.value
      return x ◊ y
   end if
end function
```


56