How does OS calculate size of directories?
Outline

- Tree & Binary Tree ADT
- Tree Traversals
  - Breadth-First Traversal
  - Depth-First Traversal
- Recursive DFT
  - pre-order, post-order, in-order
- Euler Tour Traversal
- Traversal Problems
- Analysis on perfect binary trees
What is a Tree?

- Abstraction of hierarchy
- Tree consists of
  - nodes with parent/child relationship
- Examples
  - Files/folders (Windows, MacOSX, …, CS33)
  - Merkle Trees (Bitcoin, CS166)
  - Encrypted Data Structures (CS2950-v)
  - Datacenter Networks (Azure, AWS, Google, CS168)
  - Distributed Systems (Distributed Storage, Cluster computing, CS138)
  - AI & Machine Learning (Decision trees, CS141, CS142)
Tree “Anatomy”
Tree Terminology

- **Root**: node without a parent (A)
- **Internal node**: node with at least one child (A, B, C, F)
- **Leaf (external node)**: node without children (E, I, J, K, G, H, D)
- **Parent node**: node immediately above a given node (parent of C is A)
- **Child node**: node(s) immediately below a given node (children of C are G and H)
- **Ancestors of a node**:
  - parent, grandparent, grand-grandparent, etc. (ancestors of G are C, A)
- **Descendant of a node**: child, grandchild, grand-grandchild, etc.
- **Depth of a node**: number of ancestors (I has depth 3)
- **Height of a tree**:
  - maximum depth of any node (tree with just a root has height 0, this tree has height 3)
- **Subtree**: tree consisting of a node and its descendants
Tree ADT

- Tree methods:
  - int size(): returns the number of nodes
  - boolean isEmpty(): returns true if the tree is empty
  - Node root(): returns the root of the tree

- Node methods:
  - Node parent(): returns the parent of the node
  - Node[] children(): returns the children of the node
  - boolean isInternal(): returns true if the node has children
  - boolean isExternal(): returns true if the node is a leaf
  - boolean isRoot(): returns true if the node is the root
Binary Trees

- Internal nodes have at most 2 children
  - left and right
  - if only 1 child, still need to specify if left or right
- Recursive definition of a Binary Tree
  - a single node
  - or a root node with at most 2 children
    - each of which is a binary tree
Binary Tree ADT

- In addition to Tree methods *binary* trees have:
  - Node `left()`: returns the left child if it exists, else NULL
  - Node `right()`: returns the right child if it exists, else NULL
  - Node `hasLeft()`: returns TRUE if node has left child
  - Node `hasRight()`: returns TRUE if node has right child
Perfection

- A binary tree is **perfect** if
  - every level is completely full
A binary tree is **left-complete** if

- every level is completely full, possibly excluding the lowest level
- all nodes are as far left as possible

![Left-complete tree](image1)

![Not left-complete tree](image2)
Aside: Decorations

- Decorating a node
  - associating a value to it

- Two approaches
  - Add new attribute to each node
    - ex: `node.numDescendants = 5`
  - Maintain dictionary that maps nodes to decoration
    - do this if you can’t modify tree
    - ex: `descendantDict[node] = 5`
Outline

- Tree ADT
- Binary Tree ADT
- **Tree Traversals**
  - Breadth-First Traversal
  - Depth-First Traversal
- Recursive DFT
  - pre-order, post-order, in-order
- Euler Tour Traversal
- Traversal Problems
- Analysis on perfect binary trees
Tree Traversals

- How would you enumerate every item in an array?
  - use a for loop from \(i\) to \(n\) and read \(\text{Arr}[i]\)

- How would you enumerate every item in a (linked) Tree?
  - not obvious…
  - because Trees don’t have an “obvious” order like arrays

- Tree traversal
  - algorithm that visits every node of a tree

- Many possible tree traversals
  - each kind of traversal visits nodes in different order
Breadth- vs. Depth-First Traversals
Traversal Strategy

- Why can we use a `for` loop to enumerate items in an array?
- Can we use a `for` loop to visit nodes in a `linked` Tree?
  - Why not?
  - we usually don’t know how many nodes
  - not clear what we should do at every iteration
- For tree traversals we’ll use a `while` loop
function **traversal**(root):  
  Store root in S  
  while S is not empty  
    get node from S  
    do something with node  
    store children in S
Traversing Strategy

function **traversal**(root):
    Store root in S
    while S is not empty
        get node from S
        do something with node
        store children in S

- What is **S** exactly?
  - A place we store nodes until we can process them
- Which node of **S** should we process next?
  - the first? the last?
Traversing Strategy — Get First Node

function traversals(root):
    Store root in S
    while S is not empty
        get node from S
        do something with node
        store children in S
Traversals Strategy — Get First Node

function \textit{traversal}(\textit{root}):  
  Store root in \textit{S}  
  while \textit{S} is not empty  
    get node from \textit{S}  
    do something with node  
    store children in \textit{S}

Does \textit{S} remind you of something?
Traversing Strategy — Get First Node

- If we get first node in S
  - we're doing FIFO…
  - so S is a queue!
- Traversal w/ Queue gives breadth-first traversal
- Why?
  - Queue guarantees a node is processed before its children
  - Children can be inserted in any order

function bft(root):
  Q = new Queue()
enqueue root
  while Q is not empty
    node = Q.dequeue()
    visit(node)
enqueue node’s left & right children
Breadth-First Traversal

- Start at root
  - Visit both of its children first,
    - Then all of its grandchildren,
      - Then great-grandchildren
        - etc…
  - Also known as
    - level-order traversal
Depth-First Traversal

‣ What if we process youngest/last element in S?
  ‣ we’re doing LIFO…
  ‣ so S is a stack!
  ‣ Traversal w/ Stack gives us…
‣ Depth-first search
  ‣ start from root
  ‣ traverse each branch before backtracking
  ‣ can produce different orders
Depth-First Traversal

function **dft** (root):
    S = new Stack()
push root
while S is not empty
    node = S.pop()
    visit(node)
push node’s left & right children

- Why does Stack give DFT?
  - Stack guarantees entire branch will be visited before visiting another branch
- Children can be pushed on stack in any order
Outline

- Tree ADT
- Binary Tree ADT
- Tree Traversals
  - Breadth-First Traversal
  - Depth-First Traversal
- **Recursive DFT**
  - pre-order, post-order, in-order
- Euler Tour Traversal
- Traversal Problems
- Analysis on perfect binary trees
Recursive Depth-First Traversal

- DFT can be implemented recursively
- With recursion we can have 3 different orders
  - **pre-order**: visits node before visiting left and right children
  - **post-order**: visits each child before visiting node
  - **in-order**: visits left child, node and then right child
Depth-First Visualizations

Starting Pre Order Traversal...

Starting Inorder Traversal

Starting Post Order Traversal...
Pre-order Traversal

function `preorder(node):`
  visit(node)
  if node has left child
    preorder(node.left)
  if node has right child
    preorder(node.right)

Note: like iterative DFT
function **postorder**(node):
  if node has left child
    postorder(node.left)
  if node has right child
    postorder(node.right)
  visit(node)
In-order Traversal

function `inorder(node):`
  if node has left child
    `inorder(node.left)`
  `visit(node)`
  if node has right child
    `inorder(node.right)`
Outline

- Tree ADT
- Binary Tree ADT
- Tree Traversals
  - Breadth-First Traversal
  - Depth-First Traversal
- Recursive DFT
  - pre-order, post-order, in-order
- **Traversal Problems**
- Analysis on perfect binary trees
When to Use What Traversal?

- How do you know which traversal to use?
- Sometimes it doesn’t matter
- Often one traversal makes solving problem easier
Tree Traversals Problems

Which traversal should be used to decorate nodes with # of descendants?

Activity #1
Which traversal should be used to decorate nodes with # of descendants?
Which traversal should be used to decorate nodes with # of descendants?
Tree Traversals Problems

- Decorating with number of descendants?
- **Post-order**
  - visits both children before node
  - easy to calculate # of descendants if you know # of descendants of both children
  - try writing pseudo-code for this
Given root, which traversal should be used to test if tree is perfect?
Given root, which traversal should be used to test if tree is perfect?
Tree Traversals Problems

Activity #2

Given root, which traversal should be used to test if tree is perfect?
Tree Traversals Problems

- Testing if tree is perfect
- **Breadth-first**
  - traverses tree level by level
  - keep track of how many nodes at level
  - each level should have twice as many as previous level
Tree Traversals Problems

- Evaluate arithmetic expression tree

\[(7 - (4 + 3)) + (9 / 3) \equiv\]

- Best traversal?
  - **post-order**: to evaluate operation, you first need to evaluate sub-expression on each side
  - What should you do when you get to a leaf?
Tree Traversals Problems

- Best traversal?
  - **post-order**: need to know size of subfolders before you can compute size of a folder
Euler Tour Traversal

- Generic traversal of binary tree
  - pre-order, post-order and in-order are special cases
- Each node visited 3 times
  - left, bottom, right
Euler Tour Traversal

- Visit node on the
  - **left** $\rightarrow$ pre-order traversal
  - **bottom** $\rightarrow$ in-order traversal
  - **right** $\rightarrow$ post-order traversal
function eulerTour(node):
    # pre-order
    visitLeft(node)

    if node has left child:
        eulerTour(node.left)

    # in-order
    visitBelow(node)

    if node has right child:
        eulerTour(node.right)

    # post-order
    visitRight(node)
Tree Traversals Problems

- Given tree, print out expression w/ parentheses

\[ (7 - (4 + 3)) + (9 / 3) \]

- Best traversal?

- Euler tour
Tree Traversals Problems

- Best traversal?
  - **Euler tour**
    - Internal nodes
      - For pre-order/left visit, print "(" 
      - For in-order/bottom visit, print operator
      - For post-order/right visit, print ")"
    - Leaves
      - Don’t do anything for pre-order and post-order visits
      - For in-order visit, print number
Outline

- Tree ADT
- Binary Tree ADT
- Tree Traversals
  - Breadth-First Traversal
  - Depth-First Traversal
- Recursive DFT
  - pre-order, post-order, in-order
- Traversal Problems

**Analysis on perfect binary trees**
Analyzing Binary Trees

- Many things can be modeled as binary trees
  - ex: Fibonacci recursive tree

\[ F(n) = F(n - 1) + F(n - 2) \]
Analyzing Binary Trees

- Knowing facts about binary trees can help with runtime analysis
  - ex: how many recursive calls are made by a binary recursive tree of height $n$?
- Perfect binary trees are easier to analyze…
  - …so often we use them to estimate analysis of general trees
Analyzing *Perfect* Binary Trees

- Number of nodes in PBT of height $h$:
  - $2^{h+1} - 1$

- Height of a PBT with $n$ nodes:
  - $\log_2(n+1) - 1$

- Number of leaves in PBT of height $h$:
  - $2^h$

- Number of nodes in PBT with $L$ leaves:
  - $2L - 1$
Induction on Perfect Binary Trees

- Can use induction to prove things about PBTs
- Using recursive definition of perfect binary trees
- Tree $T$ is a perfect binary tree if
  - it has only one node
  - has root with left and right subtrees which are both perfect binary trees of same height
  - (if subtrees have height $h$, then $T$ has height $h+1$)
Example Inductive Proof on PBTs

- Prove $P(n)$:
  - number of nodes in a PBT of height $n$ is $2^{n+1} - 1$
- Base case $P(0)$:
  - Number of nodes in PBT of height $0$ is $1$, because tree only has a root by definition
  - $2^{0+1} - 1 = 2 - 1 = 1$
- Inductive hypothesis:
  - assume $P(k)$ is true (for some $k \geq 0$)
  - in other words: the number of nodes in PBT of height $k$ is $2^{k+1} - 1$
Example Inductive Proof on PBTs

› Then prove that \( P(k+1) \) is true:
  
  › Let \( T \) be any PBT of height \( k+1 \)
  
  › By definition, \( T \) consists of root with two subtrees, \( L \) and \( R \), which are both PBTs of height \( k \)
  
  › By inductive hypothesis, \( L \) and \( R \) both have \( 2^{k+1} - 1 \) nodes
  
  › Number of nodes in \( T \) is therefore:
    
    › \( 1 + 2 \times (2^{k+1} - 1) = 1 + 2^{k+2} - 2 = 2^{(k+1)+1} - 1 \)
  
› Since we’ve proved
  
  › \( P(0) \) is true
  
  › \( P(k) \) implies \( P(k+1) \) for any \( k \geq 0 \)
  
  › It follows by induction that \( P(n) \) is true for all \( n \geq 0 \)
Tree ADT vs. Data Structure

- Is a Tree an ADT or a data structure?
  - It’s both
  - The answer depends on the context
- Trees are useful and interesting *abstract* objects
  - that capture parent/child relationships
  - they can be implemented using different data structures
    - some trees can be implemented using arrays
    - they can also be implemented using dictionaries
- But when computer scientists talk about Trees they often mean
  - the “linked tree” data structure
  - trees that are implemented using nodes and pointers