Tree Traversals

Types of Traversals

Breadth-first traversal can be implemented with a queue (a FIFO data structure)

Depth-first traversal requires a stack (LIFO)

Huh?? Didn't we already see some depth-first examples?

We didn't use a stack — did we??

Actually, we did use a stack — quite hidden, but we did use the stack of the processor !! (each instance of the recursive function corresponds to a node, and the stack pointer keeps track of which instance it has to return to)

Types of Traversals

- The breadth-first traversal visits all nodes at depth k before proceeding onto depth $k + 1$
- Easy to implement using a queue

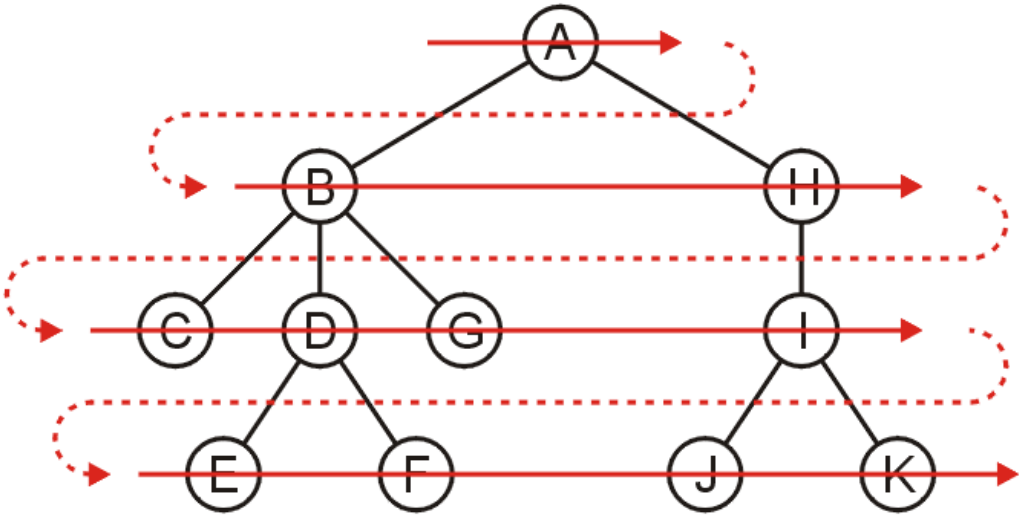
Another approach is to visit always go as deep as possible before visiting other siblings: *depth-first traversals*

Breadth First Traversal
(a.k.a. Level Order)

Breadth-First Traversal

Breadth-first traversals visit all nodes at a given depth

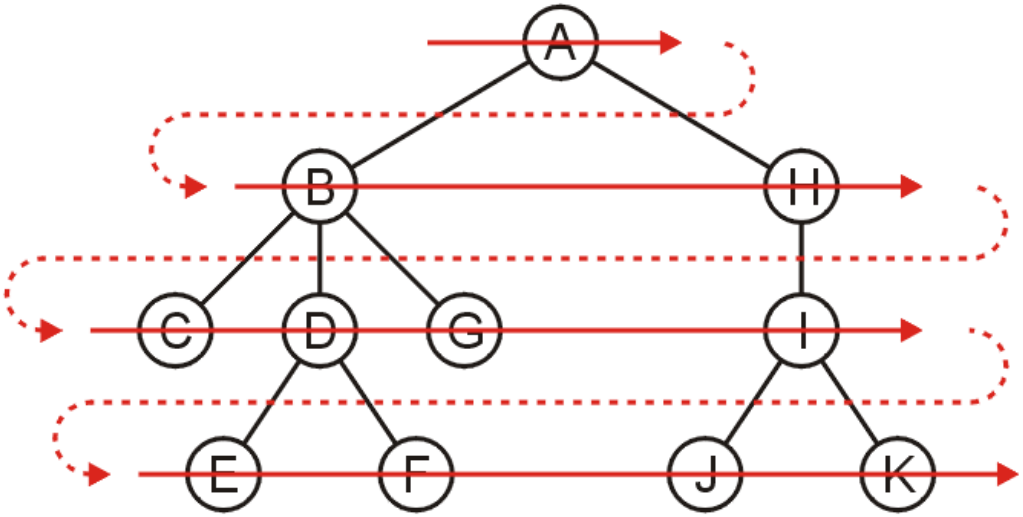
- Can be implemented using a queue
- Run time is $\Theta(n)$
- Memory is potentially expensive: maximum nodes at a given depth
- Order: A B H C D G I E F J K



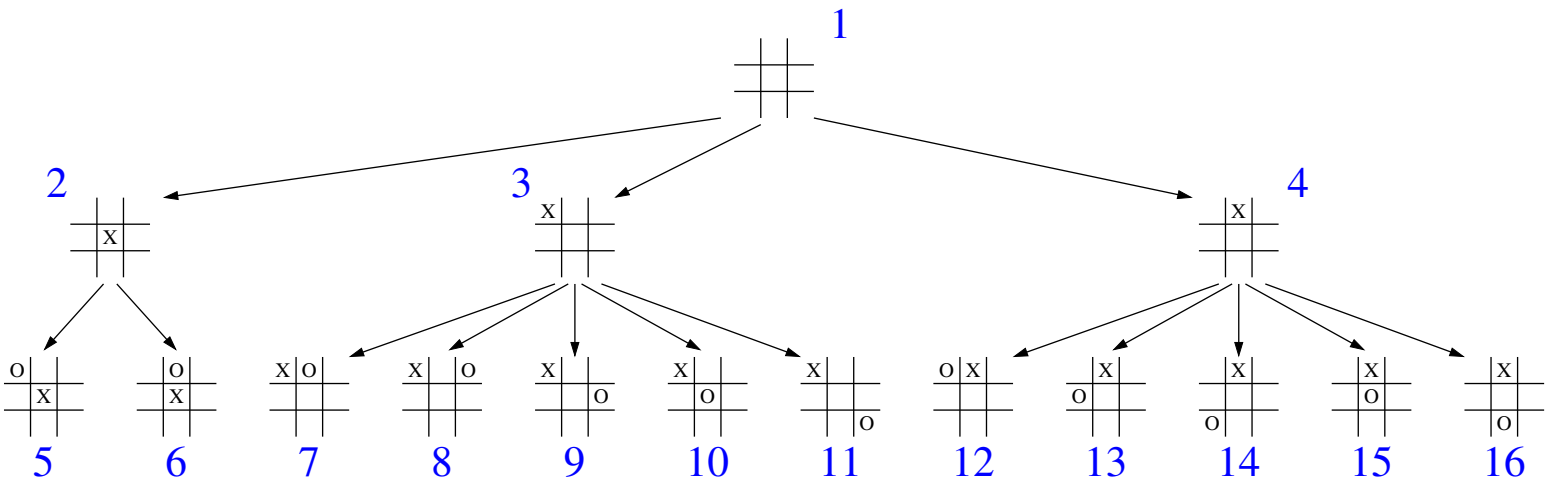
Breadth-First Traversal

The implementation was already discussed:

- Create a queue and push the root node onto the queue
- While the queue is not empty:
  - Push all of its children of the front node onto the queue
  - Pop the front node



Breadth-First Traversal

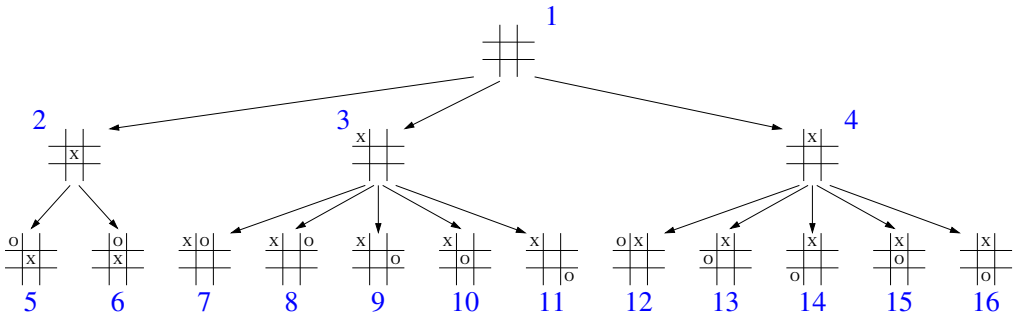


Partial game tree for Tic-Tac-Toe, with annotations displaying the order in which positions are visited in a breadth-first traversal.

Breadth-First Traversal

- Recall that the breadth-first traversal **algorithm is not recursive**; it relies on a queue of positions to manage the traversal process.
- Our implementation uses the **LinkedQueue** class from earlier lectures.

Breadth-First Traversal



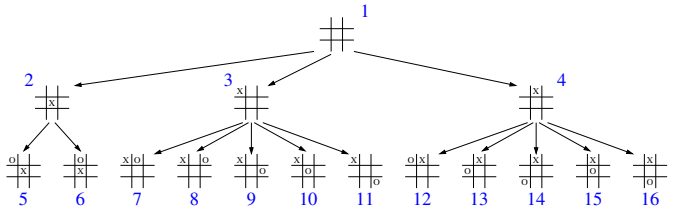
The process is not recursive.

Since we are not traversing the entire subtree at once.

We use a queue to produce FIFO semantics for the order in which we visit the nodes.

The overall running time is O(n), due to n calls to enqueue and n calls to dequeue.

Breadth-First Traversal



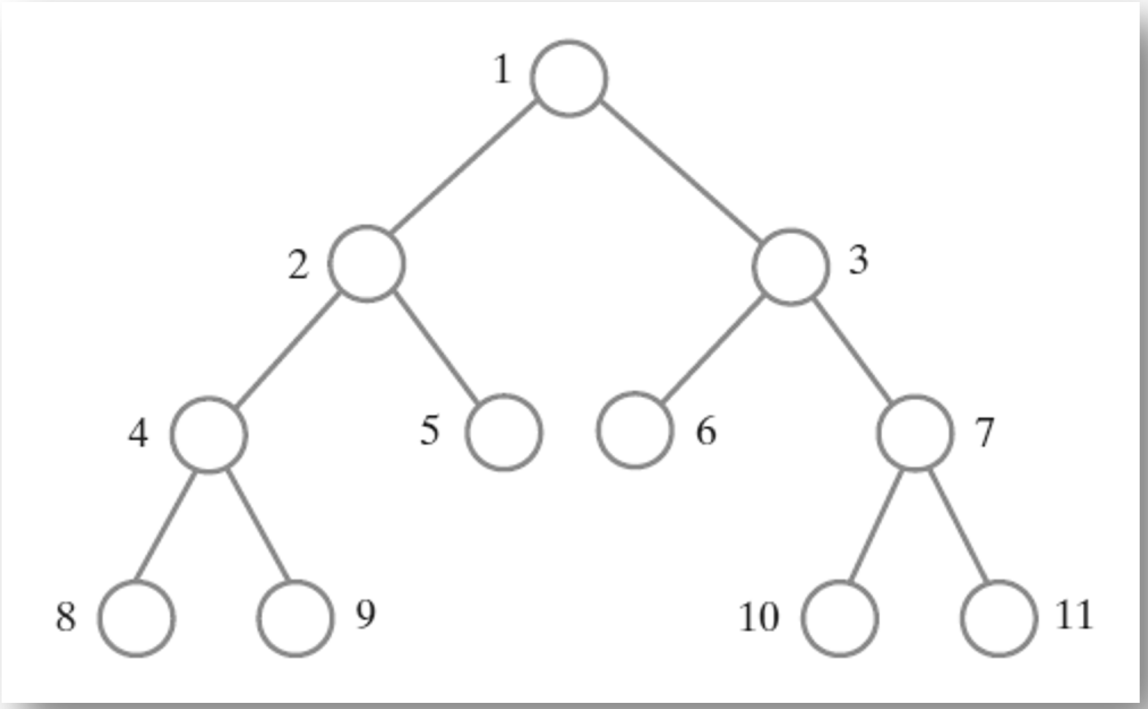
Algorithm breadthfirst(T):
Initialize queue Q to contain T.root()
while Q not empty **do**
    p = Q.dequeue() {p is the oldest entry in the queue}
    perform the "visit" action for position p
    for each child c in T.children(p) **do**
        Q.enqueue(c) {add p's children to the end of the queue for later visits}

Breadth-First Traversal

```
def breadthfirst(self):
  """Generate a breadth-first iteration of the positions of the tree."""
  if not self.is_empty():
    fringe = LinkedQueue()              # known positions not yet yielded
    fringe.enqueue(self.root())         # starting with the root
    while not fringe.is_empty():
      p = fringe.dequeue()              # remove from front of the queue
      yield p                           # report this position
      for c in self.children(p):
        fringe.enqueue(c)               # add children to back of queue
```

This code is located under Tree.py

Traversals of a Binary Tree



The visitation order of a **level-order** traversal

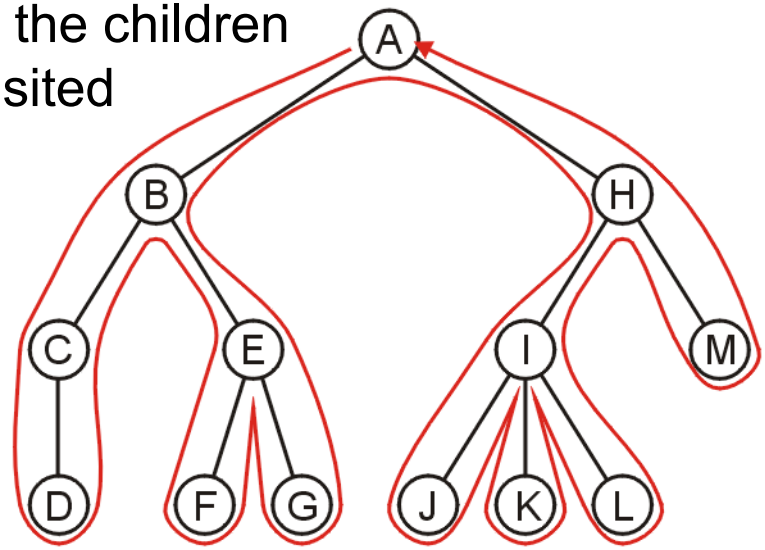
Depth First Traversal

Backtracking

To discuss **depth-first traversals**, we will define a backtracking algorithm for stepping through a tree:

- At any node, we proceed to the first child that has not yet been visited
- Or, if we have visited all the children (of which a leaf node is a special case), we backtrack to the parent and repeat this decision making process

We end once all the children of the root are visited

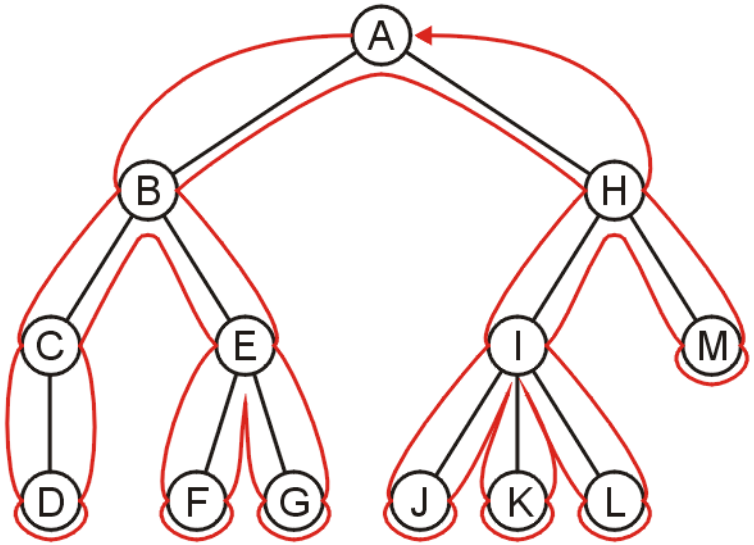


Depth-first Traversal

We define such a path as a *depth-first traversal*

We note that each node could be visited twice in such a scheme

- The first time the node is approached (before any children)
- The last time it is approached (after all children)



Pre Order,
Post Order Traversals are
Depth-First Traversals

Investigated traversal strategies:

Breadth-first (visit all siblings before descending)

Depth-first (go as deep as possible before moving to the next sibling)

Pre-order traversal:(process current node, then children)

Post-order traversal:(process all children, then current node)

Visiting a node (What does it mean ?)

“ Visiting a node” means “ processing the data within a node.”

It is an action that we perform during a traversal of a tree.

Visiting a node

A traversal can pass through a node without visiting it at that moment.

Realize that traversals of a tree are based on the **positions of its nodes**, but not on the nodes' data values.

Traversals of a Binary Tree

- We use recursion
- To visit all the nodes in a binary tree, we must
- Visit the root
- Visit all the nodes in the root's left subtree
- Visit all the nodes in the root's right subtree

Traversals of a Binary Tree

- **Pre**order traversal
  - Visit root before we visit root's subtrees(root,left,right)
- **Post**order traversal
  - Visit root of a binary tree after visiting nodes in root's subtrees (left,right,root)
- **Level**-order traversal (a.k.a Breadth-First)
  - Begin at root and visit nodes one level at a time

Pre-Order Traversal

Implementation: Preorder Traversal of Trees

In a preorder traversal of a tree T ,

- the **root of T is visited first** and
- then the sub- trees rooted at its children are traversed recursively.

If the tree is ordered, then the sub-trees are traversed according to the order of the children.

The pseudo-code for the preorder traversal of the subtree rooted at a position p is shown in next slide.

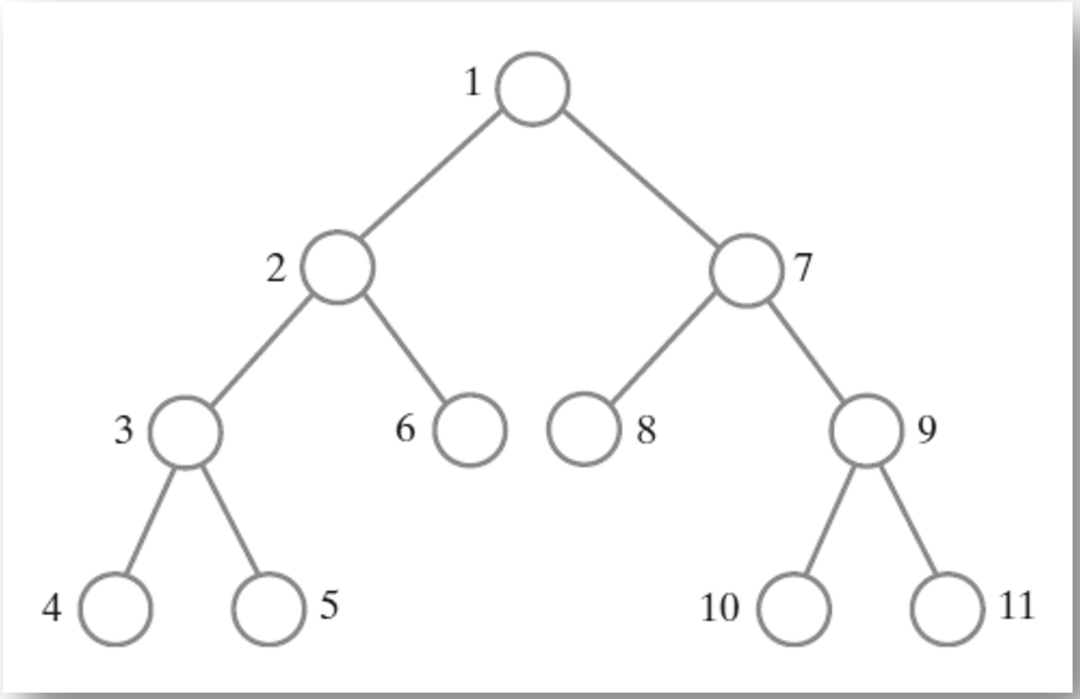
Implementation: Preorder Traversal of Trees

Algorithm preorder(T, p):
    perform the "**visit**" action for position p
    for each child c in T.children(p) **do**
        preorder(T, c)   {recursively traverse the subtree rooted at c}

Algorithm preorder for performing the preorder traversal of a subtree rooted at position p of a tree T.

Algorithm ***preOrder***(***v***)
    visit(***v***)
    for each child ***w*** of ***v***
        preorder (***w***)

Traversals of a Binary Tree

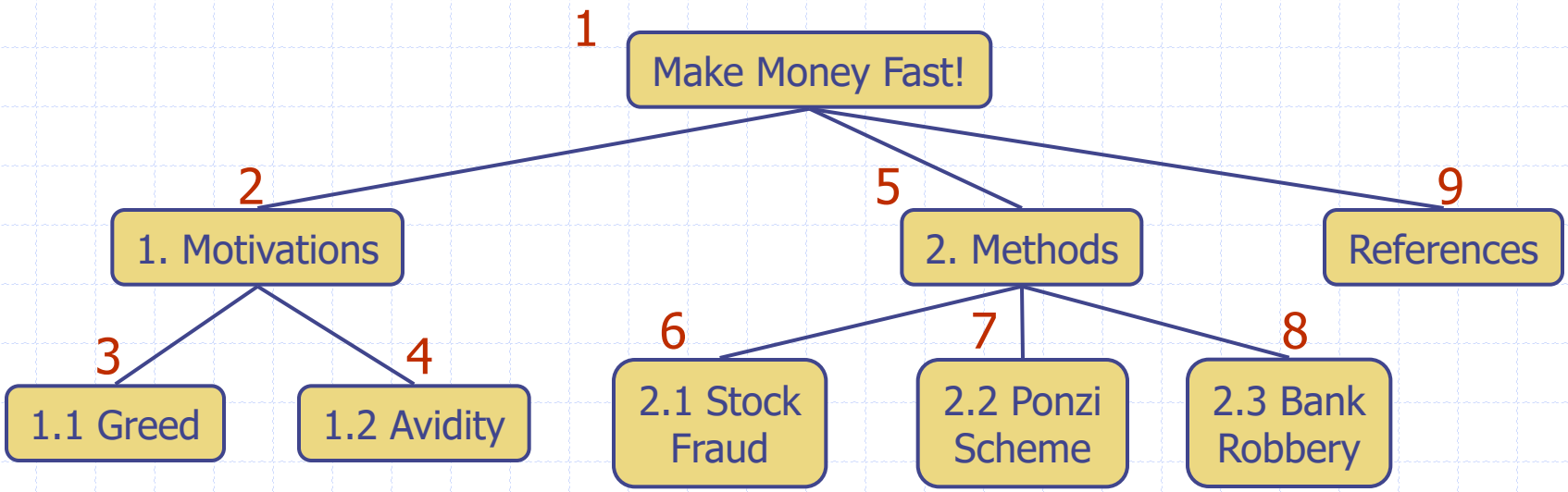


The visitation order of a **preorder** traversal

Preorder Traversal

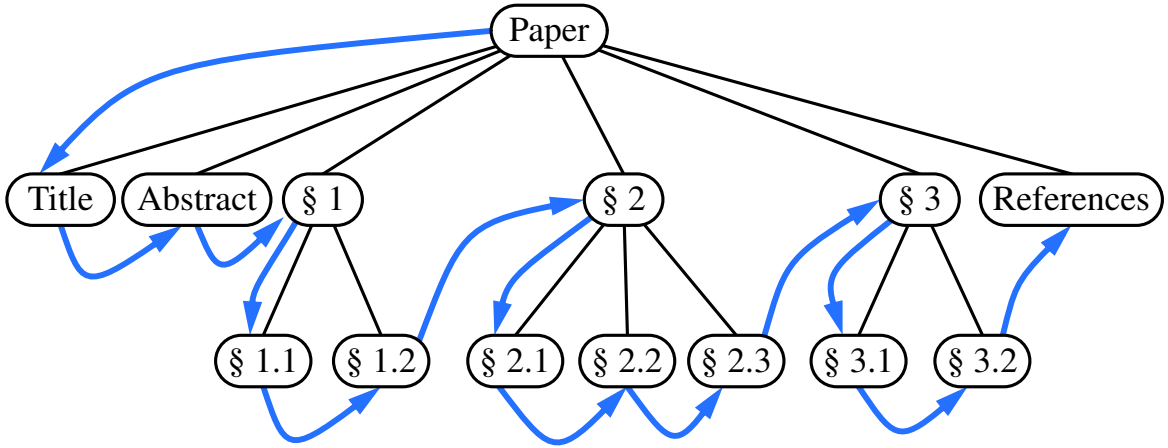
- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)



Implementation:
Preorder Traversal of Trees

```
Algorithm preorder(T, p):
    perform the "visit" action for position p
    for each child c in T.children(p) do
        preorder(T, c)          {recursively traverse the subtree rooted at c}
```



Preorder traversal of an ordered tree, where the children of each position are **ordered from left to right**.

Implementation in Python: Preorder Traversal of Trees

Encapsulation
Using wrapper

```
def preorder(self):
  """Generate a preorder iteration of positions in the tree."""
  if not self.is_empty():
    for p in self._subtree_preorder(self.root()):   # start recursion
      yield p

def _subtree_preorder(self, p):
  """Generate a preorder iteration of positions in subtree rooted at p."""
  yield p                                          # ROOT: visit p before its subtrees
  for c in self.children(p):                       # CHILDREN:for each child c
    for other in self._subtree_preorder(c):        # do preorder of c's subtree
      yield other                                  # ORDER LEFT RIGHT
                                                   yielding each to our caller
```

Support for performing a preorder traversal of a tree. This code should be included in the body of the Tree class.

Implementation in Python: Preorder Traversal of Trees

```
def _subtree_preorder(self, p):
  """Generate a preorder iteration of positions in subtree rooted
at p."""
  yield p                                  # ROOT: visit p before its
subtrees
  for c in self.children(p):               # CHILDREN:for each
child c
    for other in self._subtree_preorder(c):        # do preorder of
c's subtree
      yield other                          # ORDER LEFT RIGHT,
                                           yielding each to our caller
```

The subtree preorder method is the recursive one.
However, because we are relying on generators rather than traditional functions, the recursion has a slightly different form.
In order to yield all positions within the subtree of child c,

- we loop over the positions yielded by the recursive call **self. subtree preorder(c),** and re-yield each position in the outer context.

- Note that if p is a leaf, the for loop over self.children(p) is trivial (this is the base case for our recursion).

Post-Order Traversal

Implementation: Postorder Traversal of Trees

- Another important tree traversal algorithm is the postorder traversal.
- In some sense, this algorithm can be viewed as the **opposite of the preorder traversal.**

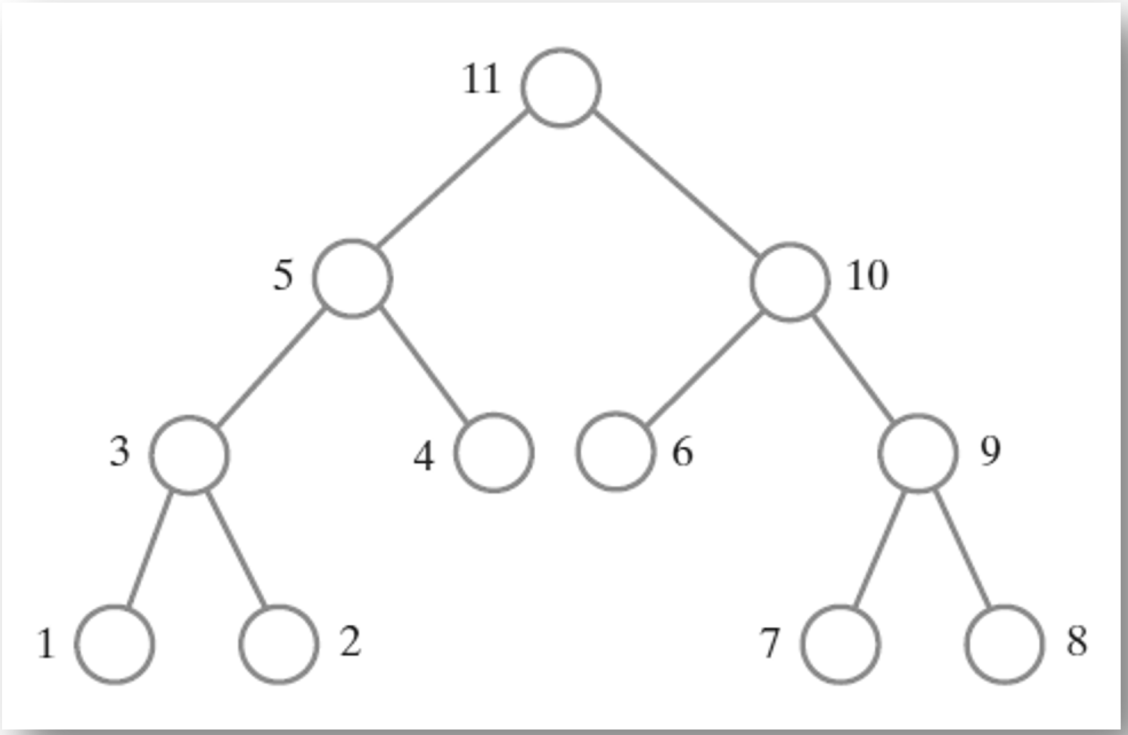
Implementation: Postorder Traversal of Trees

- Because it recursively traverses the sub-trees rooted at the children of the root first, and then …
- visits the root (hence, the name “postorder”)

Implementation: Postorder Traversal of Trees

Algorithm postorder(T, p):
for each child c in T.children(p) do
postorder(T, c) {recursively traverse the subtree rooted at c}
perform the "visit" action for position p

Traversals of a Binary Tree



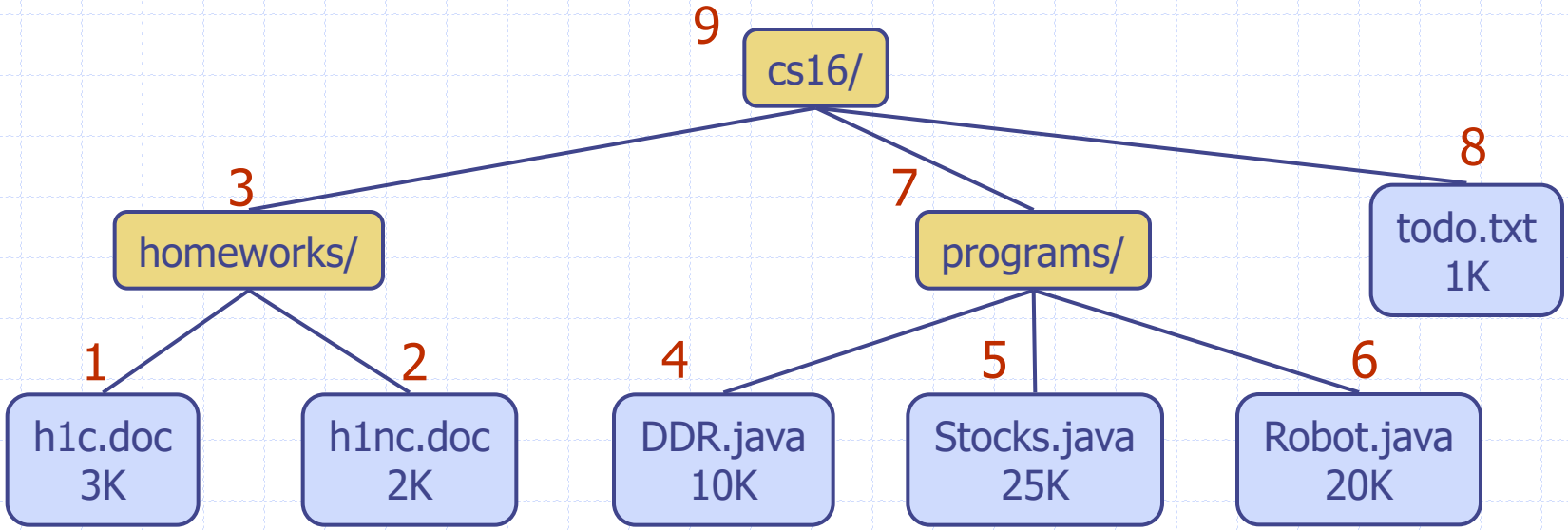
The visitation order of a **postorder** traversal

Sebnem Onsay

Postorder Traversal

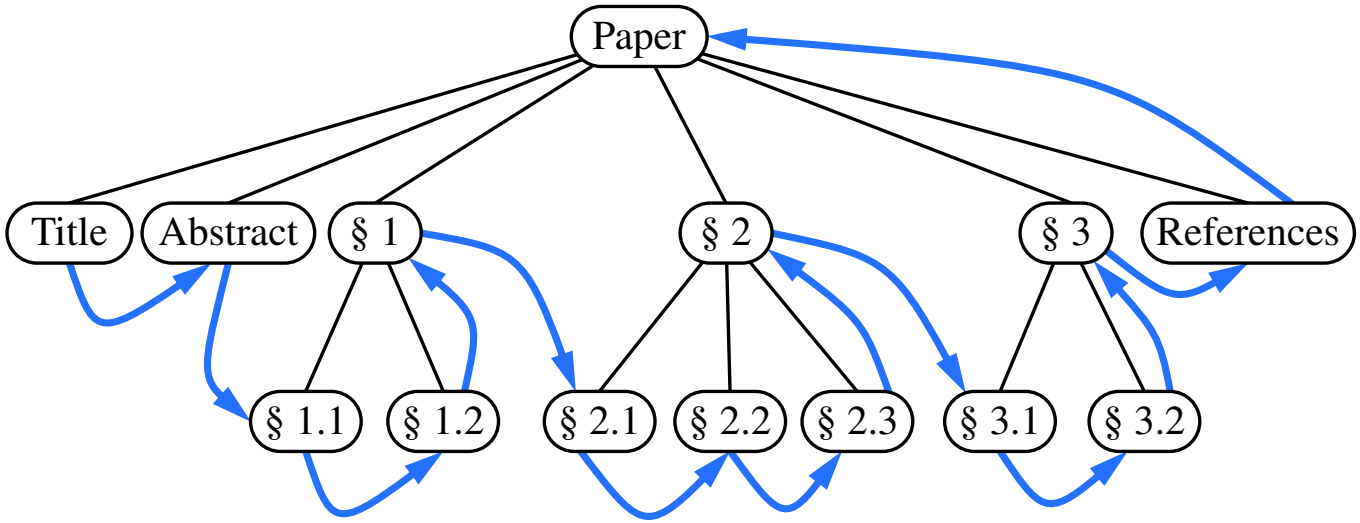
- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

Algorithm *postOrder*(*v*)
  for each child *w* of *v*
    postOrder (*w*)
  visit(*v*)



Implementation: Postorder Traversal of Trees

Algorithm postorder(T, p):
for each child c in T.children(p) do
postorder(T, c) {recursively traverse the subtree rooted at c}
perform the "visit" action for position p



Postorder traversal of the ordered tree

Implementation: Postorder Traversal of Trees

- We can implement a postorder traversal using very similar technique as with a **preorder** traversal.
- **The only difference is that** within the recursive utility for a post- order **we wait to yield position p** until after we have recursively yield the positions in its subtrees.

Implementation: Postorder Traversal of Trees

```
def postorder(self):
  """Generate a postorder iteration of positions in the tree."""
  if not self.is_empty():
    for p in self._subtree_postorder(self.root()):   # start recursion
      yield p

def _subtree_postorder(self, p):
  """Generate a postorder iteration of positions in subtree rooted at p."""
  for c in self.children(p):                    # CHILDREN:for each child c
    for other in self._subtree_postorder(c):        # do postorder of c's subtree
      yield other                                   # yielding each to our caller
  yield p                                       # ROOT: visit p after its subtrees
```

This code should be included in the body of the Tree class.

Running-Time Analysis for Pre and Postorder

- At each position p, the nonrecursive part of the traversal algorithm requires time $O(c_p + 1)$, where **c_p is the number of children of p**, under the assumption that the "visit" itself takes O(1) time.
- The overall running time for the traversal of tree T is O(n), where **n is the number of positions** in the tree.
- This running time is asymptotically optimal since the traversal must visit all the n positions of the tree

In-Order Traversal

Inorder Traversal of a Binary Tree

- The standard preorder, **postorder, and breadth-first traversals that were introduced for general trees**, can be directly applied to binary trees.
- In this section, we talk about another common traversal algorithm **specifically for a BST→ Inorder Traversal**

Inorder Traversal of a Binary Tree

- During an in-order traversal, we visit:
  - a position between the recursive traversals of its left and right subtrees.
  - The inorder traversal of a binary tree T can be informally viewed as visiting the nodes of T "**from left to right**."
  - Indeed, for every position p, the **inorder** traversal visits p after all the positions in the left subtree of p and before all the positions in the right subtree of p.

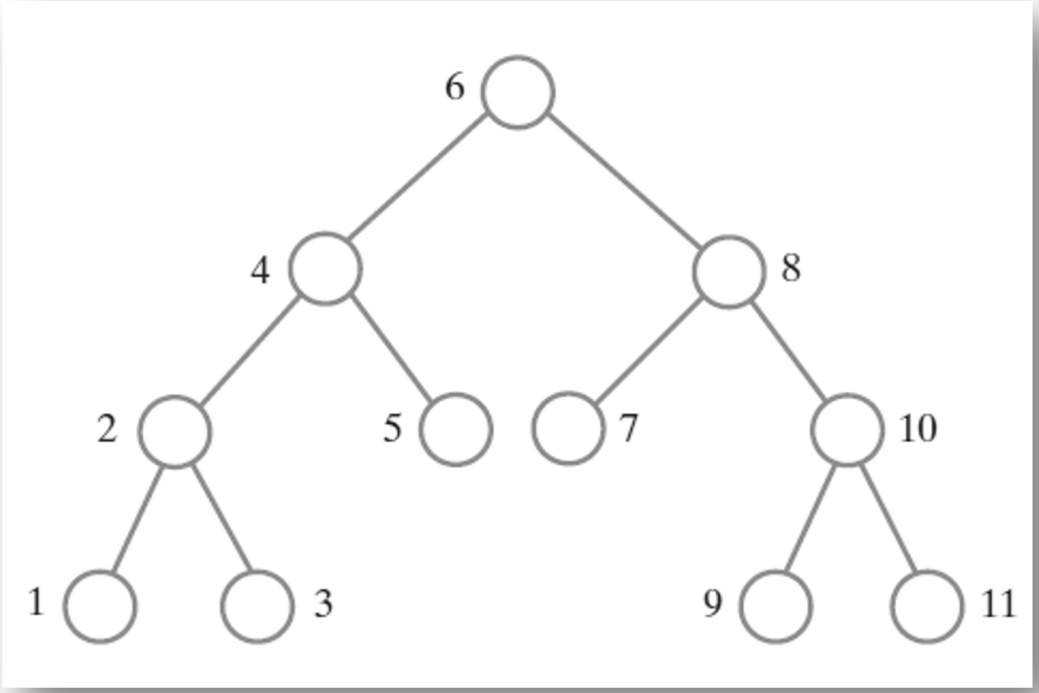
Inorder Traversal of a Binary Tree

Pseudo-code for the inorder traversal

```
Algorithm inorder(p):
    if p has a left child lc then
        inorder(lc)          {recursively traverse the left subtree of p}
    perform the "visit" action for position p
    if p has a right child rc then
        inorder(rc)          {recursively traverse the right subtree of p}
```

Algorithm above inorder for performing an inorder traversal of a subtree rooted at position p of a binary tree

Traversals of a Binary Tree

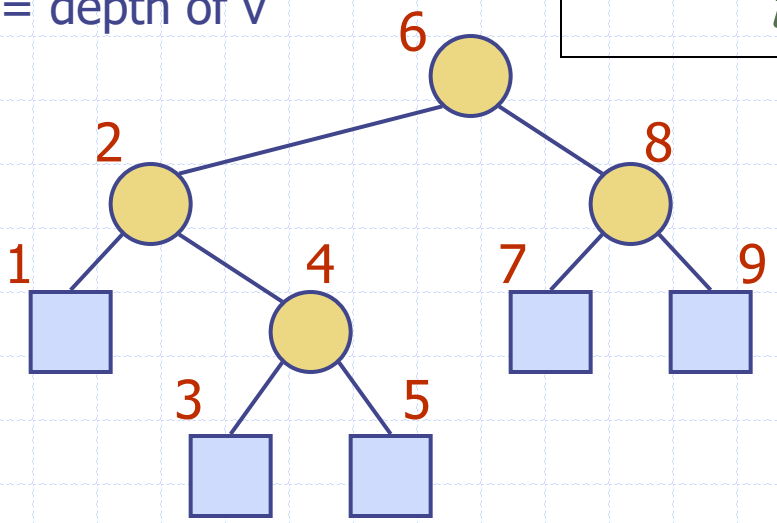


The visitation order of an **inorder** traversal

Inorder Traversal

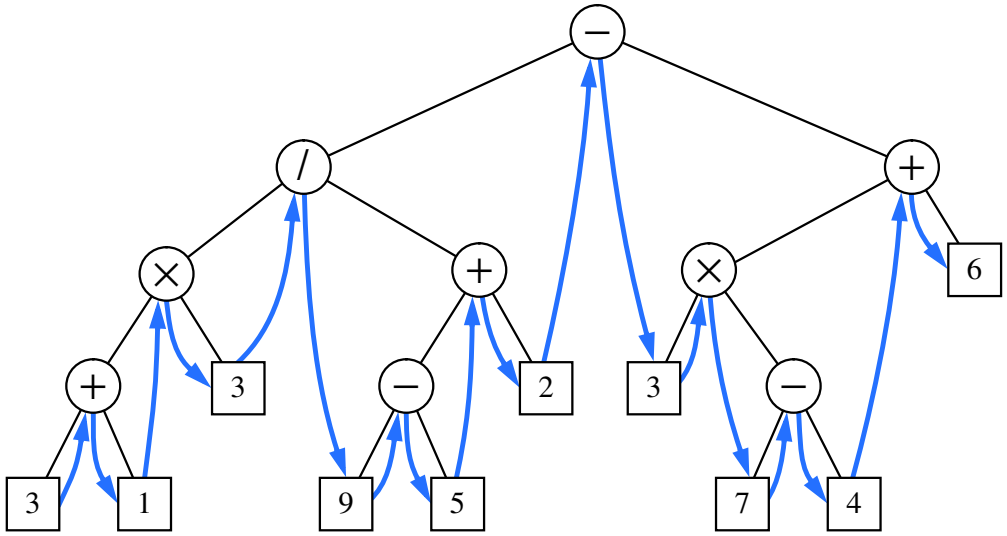
- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm *inOrder(v)*
  if *v* **has a left child**
    inOrder (*left* (*v*))
  visit(*v*)
  if *v* **has a right child**
    inOrder (*right* (*v*))



Inorder Traversal of a Binary Tree

Algorithm inorder(p):
if p has a left child lc then
inorder(lc) {recursively traverse the left subtree of p}
perform the "visit" action for position p
if p has a right child rc then
inorder(rc) {recursively traverse the right subtree of p}



Inorder traversal for this tree is =???

Inorder Traversal of a Binary Tree Implementation in Python

Encapsulation
Using wrappers

```
def inorder(self):
  """Generate an inorder iteration of positions in the tree."""
  if not self.is_empty():
    for p in self._subtree_inorder(self.root()):
      yield p

def _subtree_inorder(self, p):
  """Generate an inorder iteration of positions in subtree rooted at p."""
  if self.left(p) is not None:          # LEFT: if left child exists, traverse its subtree
    for other in self._subtree_inorder(self.left(p)):
      yield other
  yield p                               # ROOT :visit p between its subtrees
  if self.right(p) is not None:         # RIGHT: if right child exists, traverse its subtree
    for other in self._subtree_inorder(self.right(p)):
      yield other
```

Removing a node from a BST using Zybooks algorithm

- https://learn.zybooks.com/zybook/MSUCSE331OnsayFall2020/chapter/24/section/6

In order Traversal Example from Zybooks

https://learn.zybooks.com/zybook/MSUCSE331Onsay2020/chapter/24/section/7

Height and insertion

https://learn.zybooks.com/zybook/MSUCSE331OnsaySpring2020/chapter/24/section/8