CSCI 204: Data Structures & Algorithms

Revised by Xiannong Meng based on textbook author's notes

Binary Tree Implementation

Revised based on textbook author's notes.

Binary Tree Implementation

- Many different implementations. We'll discuss two.
  - Linked node based
  - Array based

Linked node based

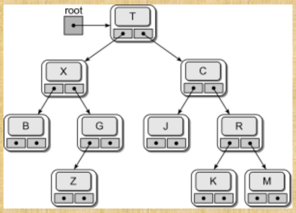
```
# The storage class for creating binary tree nodes.
class BinTreeNode :
  def __init__( self, data ):
    self.data = data
    self.left = None
    self.right = None

  def set_left(self, leftnode):
     """Set the incoming node as the left child"""
     self.left = leftnode

  """similar functions follow"""
  def set_right(self, rightnode):
  def set_data(self, new_data):
  def get_data(self):
  def get_left(self):
  def get_right(self):
```

bintreenode.py

Physical Implementation



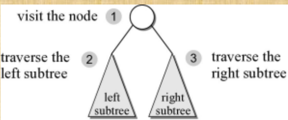
testbintree.py

Tree Traversals

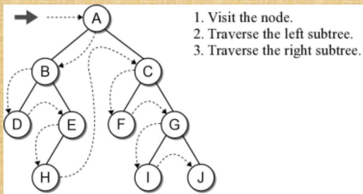
- Iterates through the nodes of a tree, one node at a time in order to visit every node.
  - With a linear structure this was simple.
  - How is this done with a hierarchical structure?
    - Must begin at the root node.
    - Every node must be visited.
    - Typically results in a recursive solution.

Preorder Traversal

- After visiting the root,
  - traverse the nodes in the left subtree
  - then traverse the nodes in the right subtree.



Preorder Traversal



1. Visit the node.
2. Traverse the left subtree.
3. Traverse the right subtree.

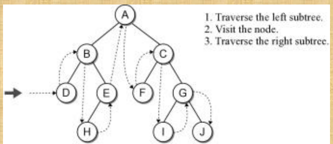
Preorder Traversal

- The implementation is rather simple.
- Given a binary tree of size n, a complete traversal requires O(n) to visit every node.

```
def preorderTrav( subtree ):
  if subtree is not None :
    print( subtree.data )
    preorderTrav( subtree.left )
    preorderTrav( subtree.right )
```

Inorder Traversal

- Similar to the preorder traversal, but we traverse the left subtree before visiting the node.



1. Traverse the left subtree.
2. Visit the node.
3. Traverse the right subtree.

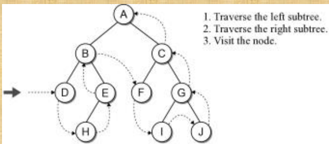
Inorder Traversal

- The implementation swaps the order of the visit operation and the recursive calls.

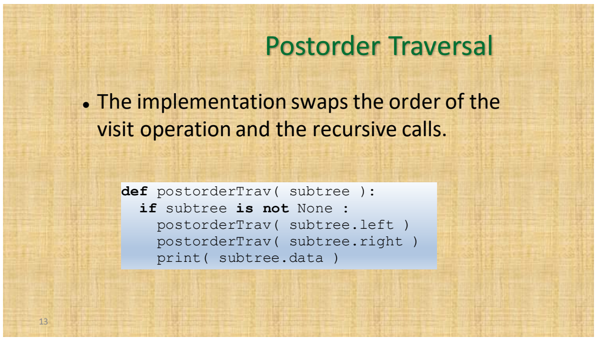
```
def inorderTrav( subtree ):
  if subtree is not None :
    inorderTrav( subtree.left )
    print( subtree.data )
    inorderTrav( subtree.right )
```

Postorder Traversal

- Is the opposite of the preorder traversal.
  - Traverse both the left and right subtrees before visiting the node.



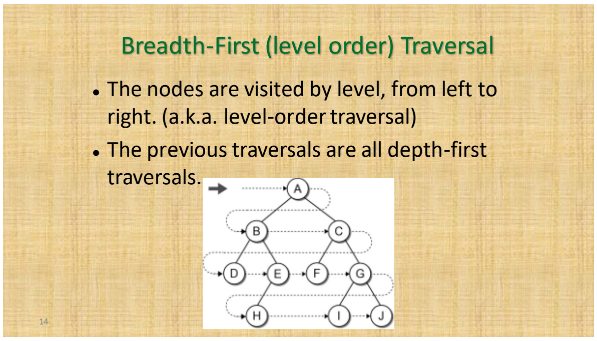
1. Traverse the left subtree.
2. Traverse the right subtree.
3. Visit the node.



Postorder Traversal

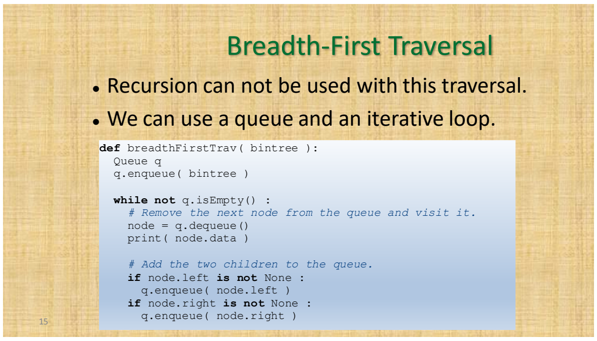
- The implementation swaps the order of the visit operation and the recursive calls.

```
def postorderTrav( subtree ):
  if subtree is not None :
    postorderTrav( subtree.left )
    postorderTrav( subtree.right )
    print( subtree.data )
```



Breadth-First (level order) Traversal

- The nodes are visited by level, from left to right. (a.k.a. level-order traversal)
- The previous traversals are all depth-first traversals.



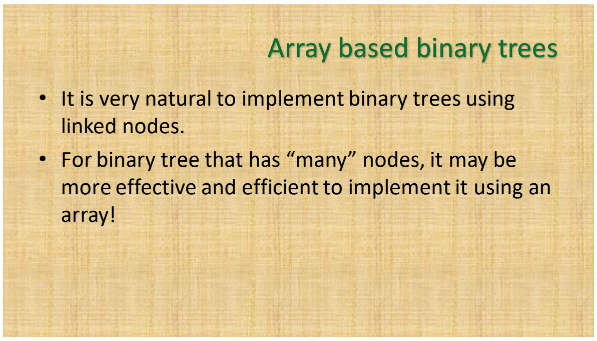
Breadth-First Traversal

- Recursion can not be used with this traversal.
- We can use a queue and an iterative loop.

```
def breadthFirstTrav( bintree ):
  Queue q
  q.enqueue( bintree )

  while not q.isEmpty() :
    # Remove the next node from the queue and visit it.
    node = q.dequeue()
    print( node.data )

    # Add the two children to the queue.
    if node.left is not None :
      q.enqueue( node.left )
    if node.right is not None :
      q.enqueue( node.right )
```



Array based binary trees

- It is very natural to implement binary trees using linked nodes.
- For binary tree that has “many” nodes, it may be more effective and efficient to implement it using an array!