Binary Tree Implementation And Applications

Revised based on textbook author's notes.

Re-build the tree from traversals

```
>>>
 RESTART: C:\Users\home\Desktop\bu-wo
intree.py
inorder traversal ...
B, X, Z, G, T, J, C, K, R, M,
preorder traversal ...
T, X, B, G, Z, C, J, R, K, M,
postorder traversal ...
B, Z, G, X, J, K, M, R, C, T,
level order traversal ...
T, X, C, B, G, J, R, Z, K, M,
>>>
```

From these traversals, can we rebuild the tree?

The answer is ‘Yes.’

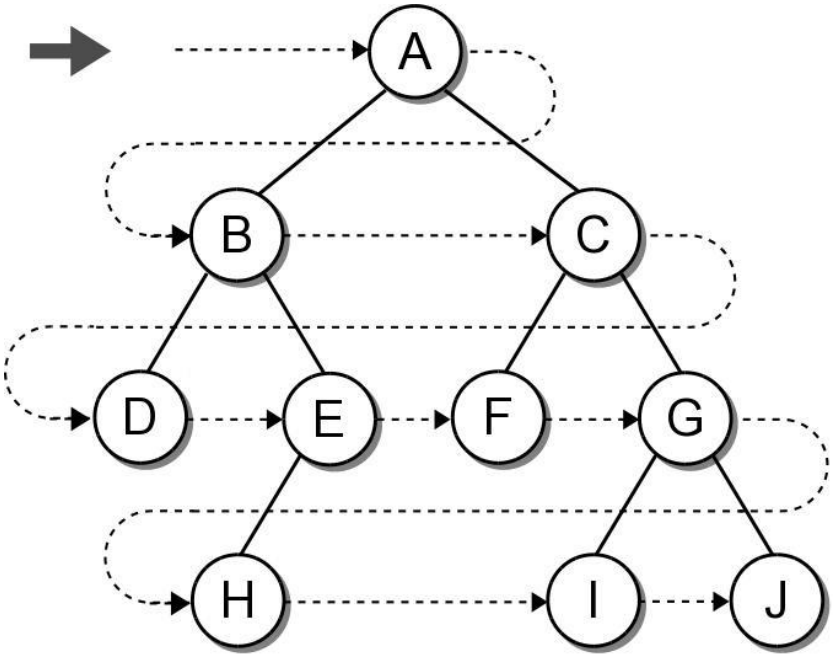
Re-build the tree from traversals

inorder traversal ...
B, X, Z, G, T, J, C, K, R, M,
preorder traversal ...
T, X, B, G, Z, C, J, R, K, M,

- From pre-order, we know T is the root.
- From in-order, we know T has left child(ren)
- From pre-order, we know X is the root of left branch
- From in-order, we know X has left child(ren)
- From pre-oder, we know B is the root of left branch of X
- Because B doesn't have child, from pre-order, we know G is the root of the right subtree of X
- …

Breadth-First (level order) Traversal

- The nodes are visited by level, from left to right.
  - 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 breadth_first_trav( bintree ):
  q = Queue()
  q.enqueue( bintree )

  while not q.is_empty() :
    # 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 )
```

Try testbintree.py

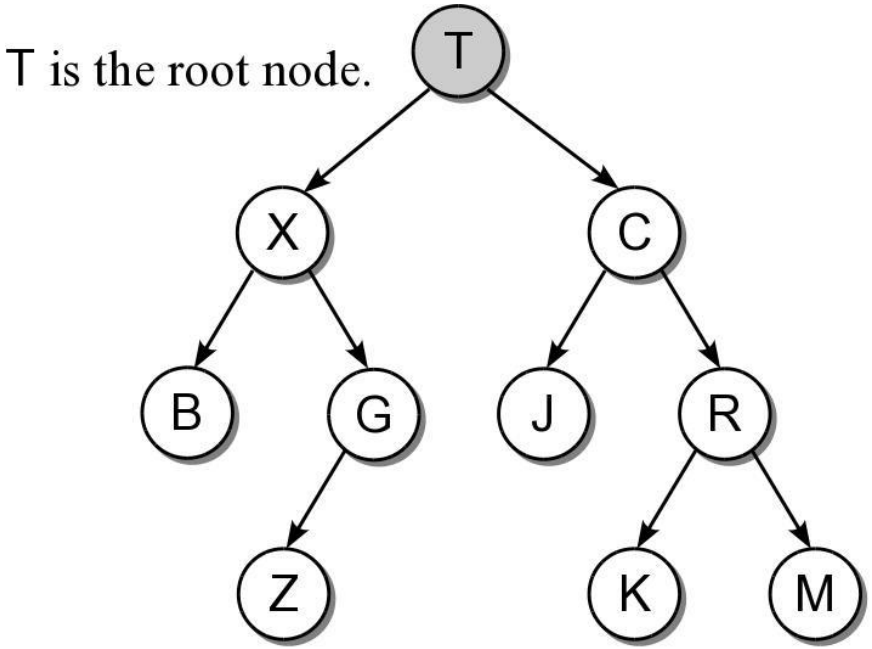
Array based binary trees

- 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!

Relation among nodes

- If the root is at index n, its left child will be at index 2*n+1, its right child will be at index 2*n+2
- If a node is at index k, its parent is at index (k-1) // 2

An array-based tree example



T	X	C	B	G	J	R			Z				K	M	
0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5

Implementation: constructor

```
from array204 import Array
from pylistqueue import Queue

class BinTree:
  """An array-based implementation of binary tree."""

  def __init__( self, maxSize ):
    """Create a binary tree with maximum capacity of maxSize."""
    self._elements = Array( maxSize )
    for i in range( maxSize ):
      self._elements[ i ] = None
    self._max_index = 0
    self.root = 0

  def __len__( self ):
    """Return the number of items in the tree"""
    return self._max_index
```

Implementation: adding node

```
def add_left( self, pindex, value ):
  """Add a new value as the left child of the pindex"""
  left = pindex * 2 + 1
  assert left < self.capacity(), "Cannot add to a full tree."
   # Add the new value
  self._elements[ left ] = value
  if left > self._max_index:      # update the max_index
    self._max_index = left

def add_right( self, pindex, value ):
  """ Add a new value as the right child of the pindex"""
  right = pindex * 2 + 2
  assert right < self.capacity(), "Cannot add to a full tree."
   # Add the new value
  self._elements[ right ] = value
  if right > self._max_index:      # update the max_index
    self._max_index = right
```

Implementation: accessors

```
def get_left( self, index ):
  """Return the index of left child"""
  left = index * 2 + 1
  if left <= self._max_index:
    return left
  else:
    return None

def get_right( self, index ):
  """Return the index of right child"""
  right = index * 2 + 2
  if right <= self._max_index:
    return right
  else:
    return None

def get_data( self, index ):
  """Return the content of the node"""
  if index <= self._max_index:
    return self._elements[ index ]
  else:
    return None

def set_data( self, index, value ):
  """Set the content of a node"""
  assert index < self.capacity(), "Index out of bound."
  self._elements[ index ] = value
```

Traversals: in-order

```
def inorder( self ):
  """Wrapper function for 'inorder'"""
  self.inorder_trav( self.root )
  print()

def inorder_trav( self, subtree ):
  """Traverse the tree inorder"""
  if subtree is not None :
    self.inorder_trav( self.get_left( subtree ) )
    self.visit( subtree )
    self.inorder_trav( self.get_right( subtree ) )
```

Traversals: pre-order

```
def preorder( self ):
  """Wrapper function for 'preorder'"""
  self.preorder_trav( self.root )
  print()

def preorder_trav( self, subtree ):
  """Traverse the tree postorder"""
  if subtree is not None :
    self.visit( subtree )
    self.preorder_trav( self.get_left( subtree ) )
    self.preorder_trav( self.get_right( subtree ) )
```

Traversals: post-order

```
def postorder( self ):
  """Wrapper function for 'postorder'"""
  self.postorder_trav( self.root )
  print()

def postorder_trav( self, subtree ):
  """Traverse the tree postorder"""
  if subtree is not None :
    self.postorder_trav( self.get_left( subtree ) )
    self.postorder_trav( self.get_right( subtree ) )
    self.visit( subtree )
```

Traversals: level-order

```
def levelorder( self ):
  """Wrapper function for 'levelorder'"""
  self.breadth_first_trav( self.root )
  print()

def breadth_first_trav( self, bintree ):
  """Traverse the binary tree in breadth-first (level) order"""
  # Create a queue and add the root node to it.
  q = Queue()
  q.enqueue( bintree )

  # Visit each node in the tree.
  while not q.is_empty() :
    # Remove the next node from the queue and visit it.
    node = q.dequeue()
    self.visit( node )

    # Add the two children to the queue.
    if self.get_left( node ) is not None :
      q.enqueue( self.get_left( node ) )
    if self.get_right( node ) is not None :
      q.enqueue( self.get_right( node ) )
```