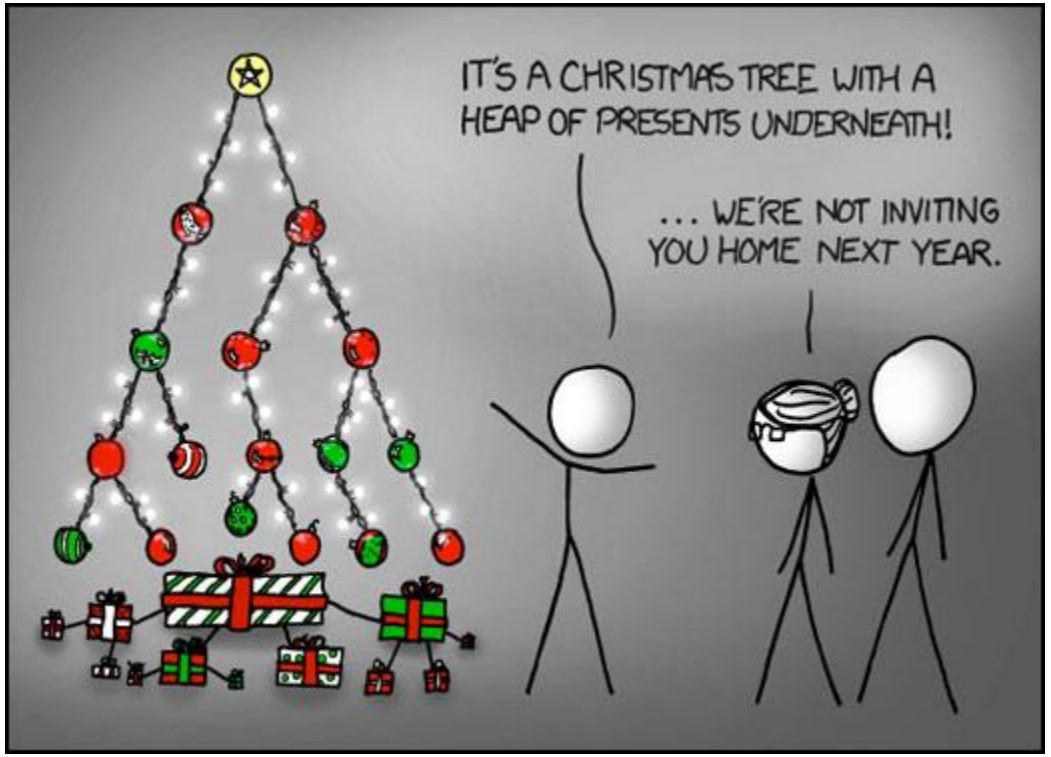
TREES

School of Artificial Intelligence

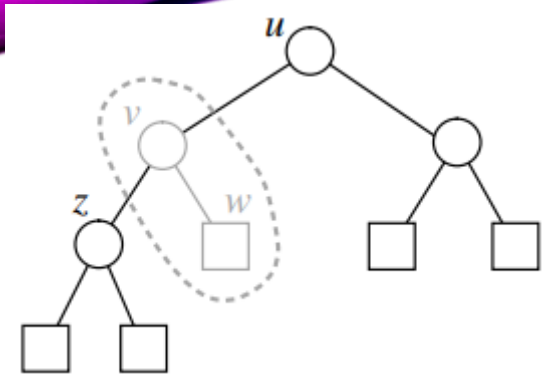
PREVIOUSLY ON DS&A

- Trees
- Terminologies
- Binary Trees
- Implementation of Trees
  - Linked Structure
  - Array-based Structure
- Tree traversal algorithms
  - Pre-order
  - Post-order
  - In-order
- Binary Search Trees



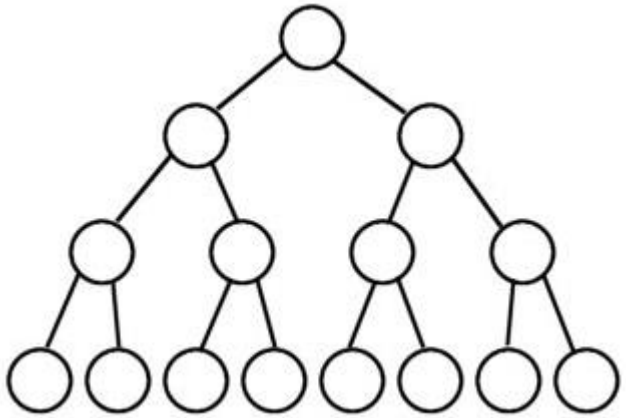
PREVIOUS LECTURE

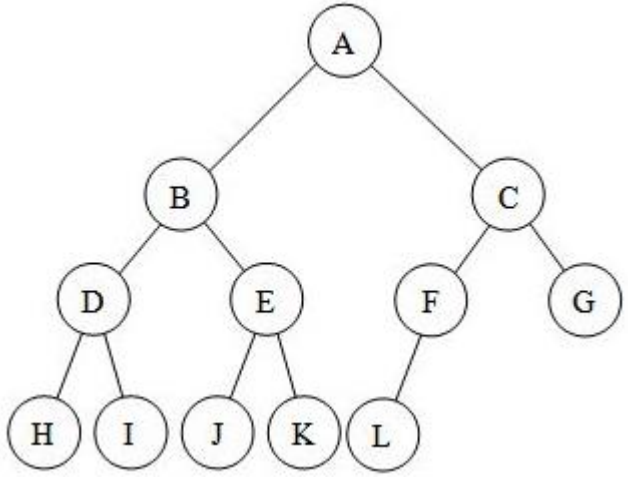
- Definition of a Tree
  - If T is non-empty, it has a special node, called the **root** of T, that has no parent
  - Each node v of T different from the root has a *unique* **parent** node w, every node with parent w is a **child** of w
- Siblings(兄弟结点): two nodes that are children of the same parent
- External node（外部结点）/leaves（叶结点）: if node v has no children
- Internal node（内部结点）: if node v has one or more children
- Edge: pair of nodes (u,v) such that u is the parent of v, or vice versa.
- Path: sequence of nodes such that any two consecutive nodes in the sequence form an edge
- Depth of a tree?
  - If p is the root then depth of p is 0
  - Otherwise, the depth of p is one plus the depth of the parent of p
- Height of a tree?
  - If p is a leaf, its height is 0
  - Otherwise, the height of p is one more than the maximum of the heights of p's children
- Binary Tree?
  - A binary tree is either empty or consists of:
    - A node r, called the root of T, that stores an element
    - A binary tree (may be empty), called the left subtree of T
    - A binary tree (may be empty), called the right subtree of T



PREVIOUS LECTURE

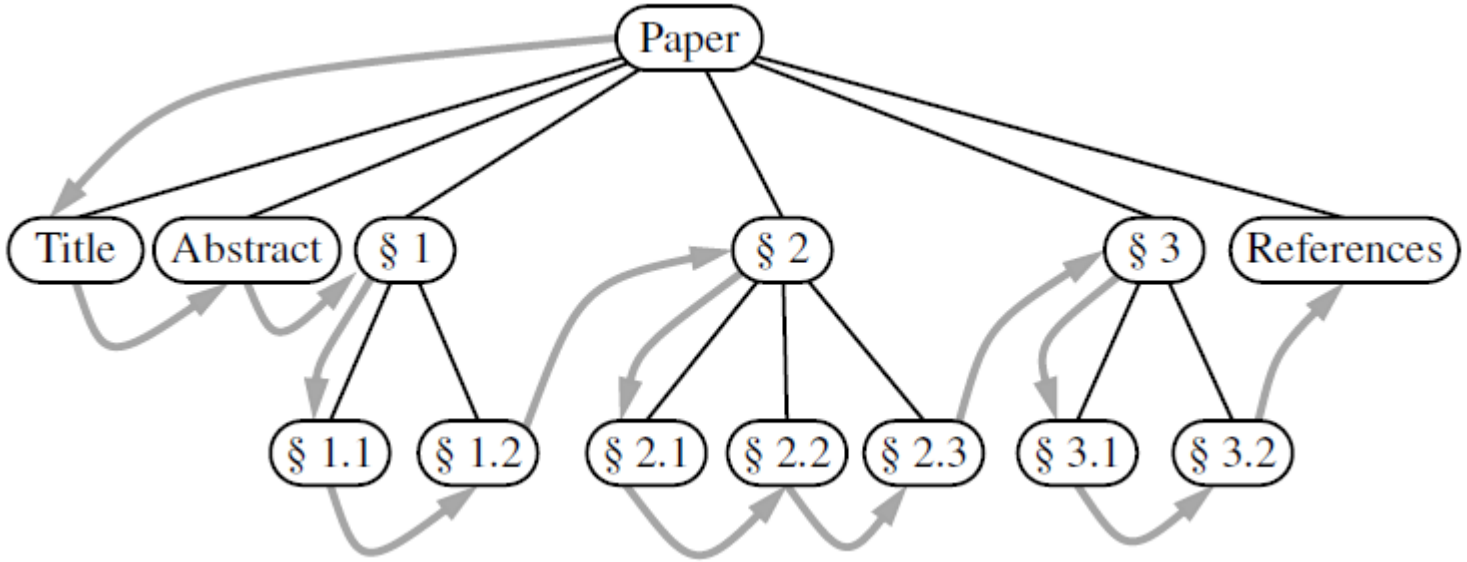
- Proper binary tree
  - A binary tree if each node has either zero or two children
- Full binary tree
  - A binary tree in which every node other than the leaves has two children
- Complete binary tree
  - A binary tree in which every level except (maybe) the last, is completely filled, and all nodes are as far left as possible





TREE TRAVERSAL （遍历） ALGORITHMS

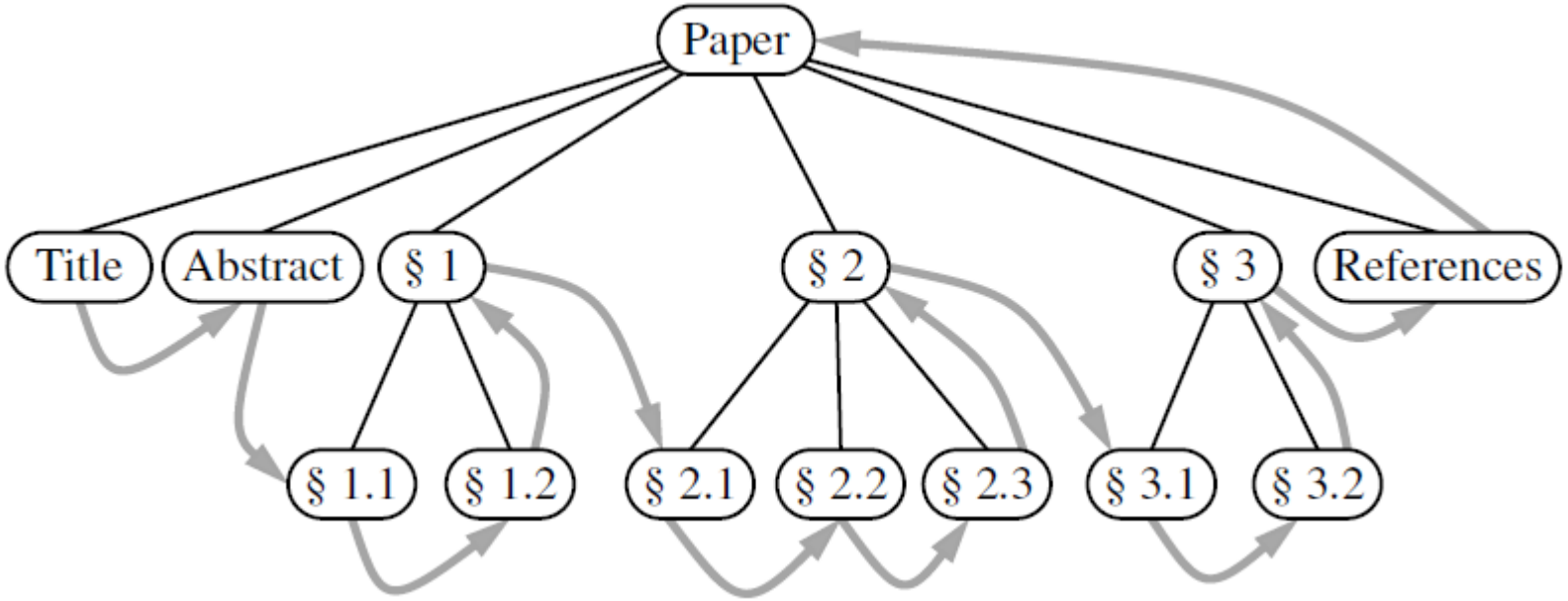
- Traversal: a systematic way of accessing or ‘visiting’ all the positions of T
- Preorder traversal （正序遍历）:
  - visit root of T first and then visit the sub-trees recursively
  - If tree is ordered, then the subtrees are traversed according to the order of the children



Algorithm preorder(T, p):
  perform the “visit” action for position p
  for each child c in T.children(p) **do**
    preorder(T, c) {recu

TREE TRAVERSAL （遍历） ALGORITHMS

- Postorder traversal （逆序遍历）：
  - The opposite of preorder traversal
  - Recursively traverses the subtrees at the children of the root first and then visits the root



Algorithm postorder(T, p):
  for each child c in T.children(p) **do**
    postorder(T, c) {recu
  perform the "visit" action for position p

TREE TRAVERSAL （遍历） ALGORITHMS

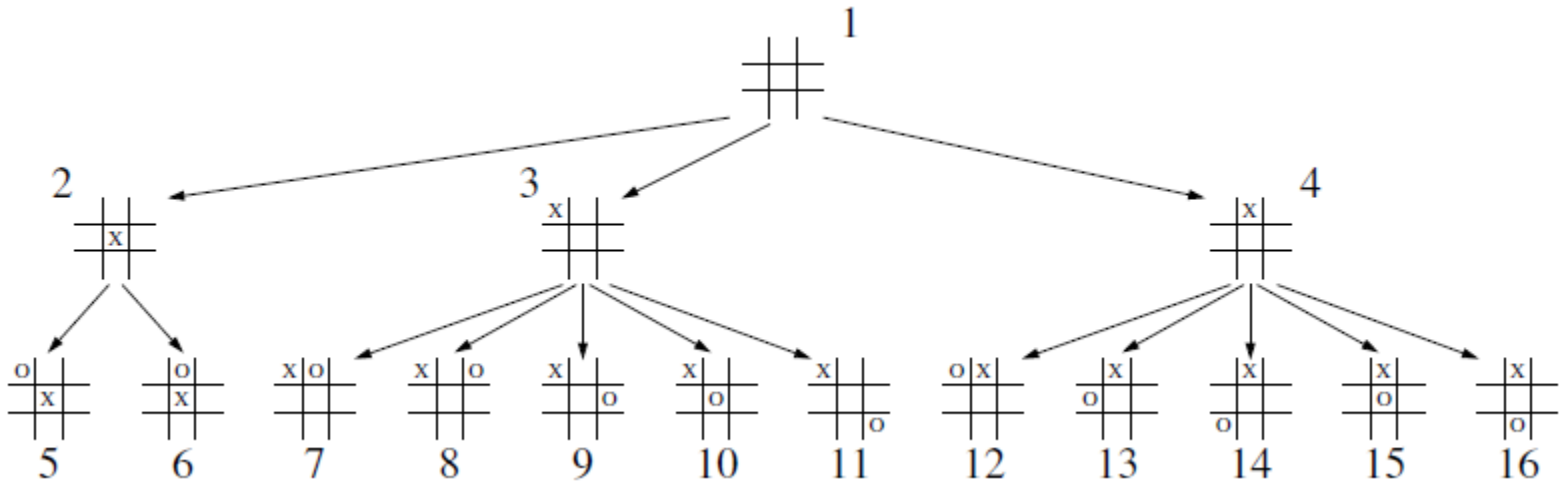
- Running time
- At each position p, non-recursive part: $O(c_p + 1)$
  - c_p number of children of p
- Overall: O(n)

Algorithm preorder(T, p):
  perform the "visit" action for position p
  for each child c in T.children(p) **do**
    preorder(T, c) {recu

Algorithm postorder(T, p):
  for each child c in T.children(p) **do**
    postorder(T, c) {recu
  perform the "visit" action for position p

TREE TRAVERSAL （遍历） ALGORITHMS

- Breadth-First （广度优先） Tree Traversal
  - Visit all the position at depth d before visit the position at depth d+1
- Commonly used in software for playing games



TREE TRAVERSAL ALGORITHMS

- Breadth-First （广度优先） Tree Traversal
  - Visit all the position at depth d before visit the position at depth d+1
- O(n) running time: n calls to enqueue() and n calls to dequeue()

```
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}
```

TREE TRAVERSAL ALGORITHMS

- In-order （中序） Traversal of a Binary Tree
  - Visit left
  - Visit node
  - Visit right

Algorithm inorder(p):

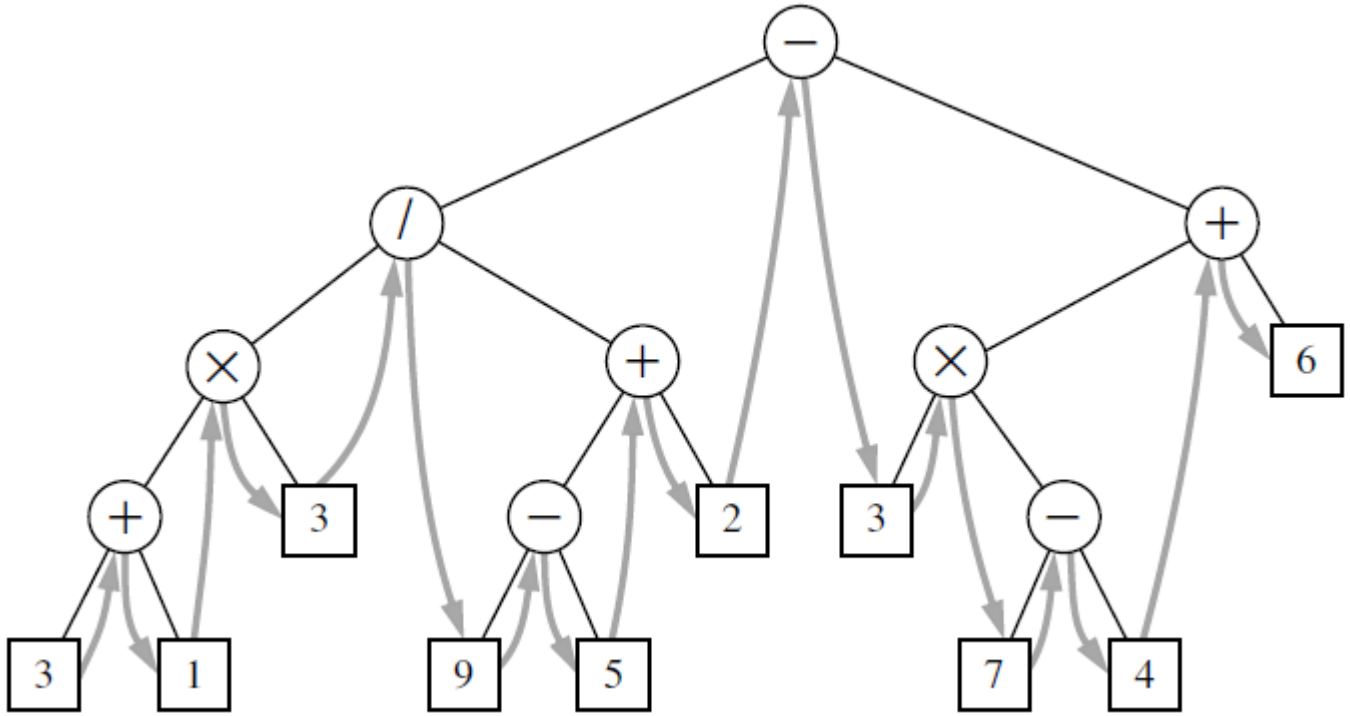
if p has a left child lc **then**

inorder(lc) {r

perform the "visit" action for position p

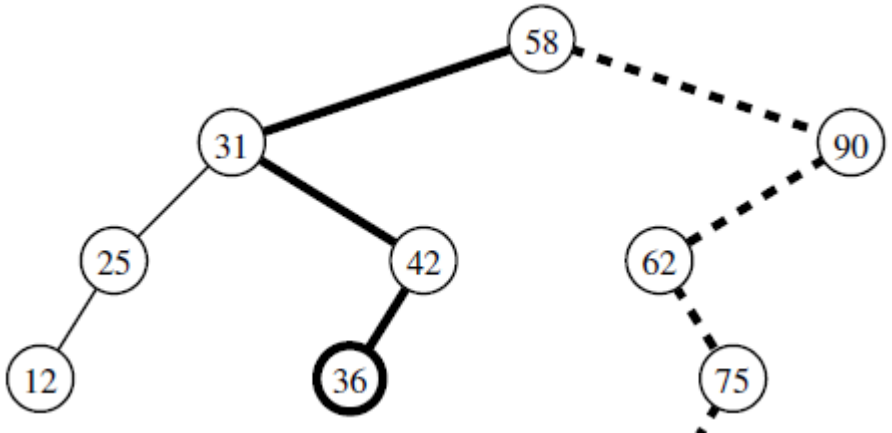
if p has a right child rc **then**

inorder(rc) {rec



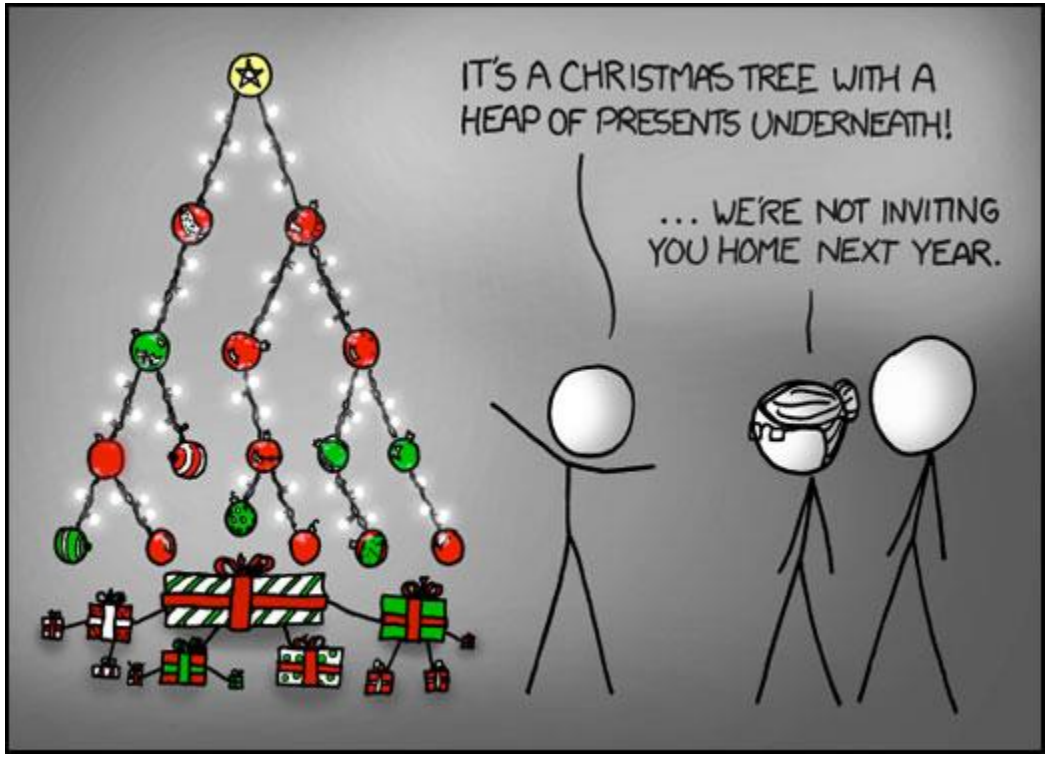
BINARY SEARCH TREES

- Binary search tree: using tree to store an ordered sequence of elements in a binary tree
- Let S be a set whose unique elements have an order relation
- A binary search tree for S is a binary tree T such that
  - Position p stores an element of S, denoted as e(p)
  - Elements stored in the left subtree of p are less than e(p)
  - Elements stored in the right subtree of p are greater than e(p)
- Running time for finding an occurrence?
  - Proportional to the height of T
  - How you organize the tree really matters
  - Best case?
  - Worst case?



THIS LECTURE

- Implementing tree traversals
- Euler's Tour
- Generic way of writing tree traversal algorithms



OBJECT ORIENTED PROGRAMMING

- An abstract base class for Tree
- Define default behaviours for operations
- Greater re-use
- Position class:
  - Element() and __eq__() not implemented
  - NotImplementedError when called
  - Typical for Object Oriented Programming
  - You need to define your own subclass for Position that overrides these two

```
class Tree:
  """Abstract base class representing a tree structure."""

  #------------------------------- nested Position class -------------------------------
  class Position:
    """An abstraction representing the location of a single element."""

    def element(self):
      """Return the element stored at this Position."""
      raise NotImplementedError('must be implemented by subclass')

    def __eq__(self, other):
      """Return True if other Position represents the same location."""
      raise NotImplementedError('must be implemented by subclass')

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)              # opposite of __eq__
```

OBJECT ORIENTED PROGRAMMING

- Tree abstract base class
- root(), parent(), num_children(), children(), __len__() not implemented

```
# --------- abstract methods that concrete subclass must support ----------
def root(self):
  """Return Position representing the tree's root (or None if empty)."""
  raise NotImplementedError('must be implemented by subclass')

def parent(self, p):
  """Return Position representing p's parent (or None if p is root)."""
  raise NotImplementedError('must be implemented by subclass')

def num_children(self, p):
  """Return the number of children that Position p has."""
  raise NotImplementedError('must be implemented by subclass')

def children(self, p):
  """Generate an iteration of Positions representing p's children."""
  raise NotImplementedError('must be implemented by subclass')

def __len__(self):
  """Return the total number of elements in the tree."""
  raise NotImplementedError('must be implemented by subclass')
```

OBJECT ORIENTED PROGRAMMING

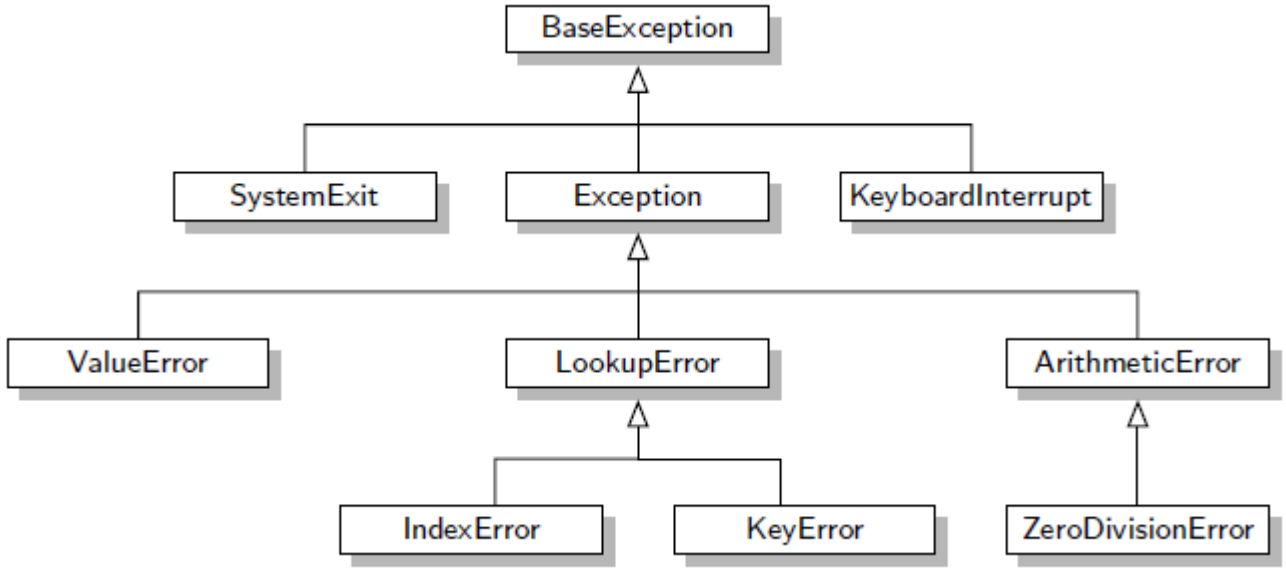
- Tree abstract base class
- Some functions are implemented as concrete methods
  - Why?
- Is it sensible to create a direct instance （实例）of the abstract Tree class?

```
# ---------- concrete methods implemented in this class ----------
def is_root(self, p):
  """Return True if Position p represents the root of the tree."""
  return self.root( ) == p

def is_leaf(self, p):
  """Return True if Position p does not have any children."""
  return self.num_children(p) == 0

def is_empty(self):
  """Return True if the tree is empty."""
  return len(self) == 0
```

Code Fragment 8.2: Some concrete methods of our Tree abstract base class.

COMPUTING DEPTH

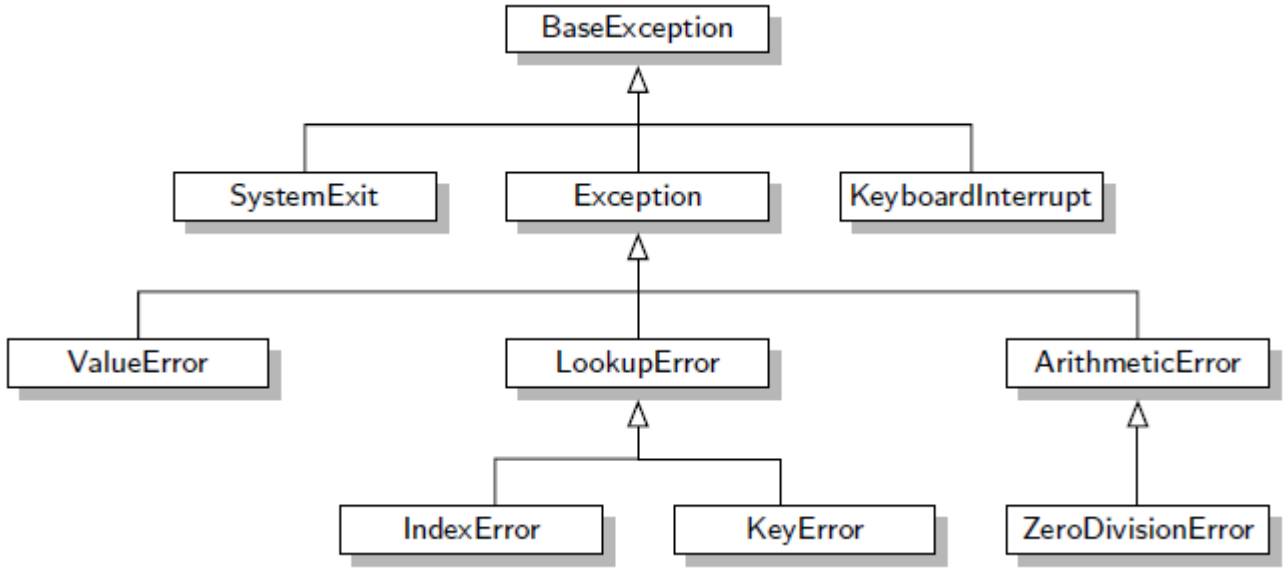
- Tree T, position of a node p
- The depth of p: number of ancestors of p, excluding p itself.
- Depth of LookUpeError: 2
- Formal definition:
  - If p is the root then depth of p is 0
  - Otherwise, the depth of p is one plus the depth of the parent of p




```
def depth(self, p):
  """Return the number of levels separating Position p from the root."""
  if self.is_root(p):
    return 0
  else:
    return 1 + self.depth(self.parent(p))
```

COMPUTING HEIGHT

- An improved version with recursion
- Complexity?




```
def _height2(self, p):                    # time is linear in size of subtree
  """Return the height of the subtree rooted at Position p."""
  if self.is_leaf(p):
    return 0
  else:
    return 1 + max(self._height2(c) for c in self.children(p))
```

IMPLEMENTING TREE TRAVERSALS

- The Tree ADT
- T.positions(): generate an iteration of all positions of tree T
- iter(T): generate an iteration of all elements stored within tree T

```
def __iter__(self):
  """Generate an iteration of the tree's elements."""
  for p in self.positions( ):        # use same order as positions()
    yield p.element( )               # but yield each element
```

- Positions() relies on the tree traversal algorithm
- We will look at how each of the traversal algorithm works
  - Independently

IMPLEMENTING TREE TRAVERSALS

- Pre-order traversal
- Implemented with generator

Algorithm preorder(T, p):
  perform the "visit" action for position p
  for each child c in T.children(p) **do**
    preorder(T, c) {recu

```
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                                          # visit p before its subtrees
  for c in self.children(p):                       # for each child c
    for other in self._subtree_preorder(c):        # do preorder of c's subtree
      yield other                                  # yielding each to our caller
```

IMPLEMENTING TREE TRAVERSALS

- Post-order traversal
- Implemented with generator

Algorithm postorder(T, p):
  for each child c in T.children(p) **do**
    postorder(T, c) {recu
  perform the "visit" action for position p

```
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):                          # 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                                             # visit p after its subtrees
```

IMPLEMENTING TREE TRAVERSALS

- Breadth-First Traversal
- Store elements to be processed in a queue
- Fringe (边缘)
  - Just a name

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}

```
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
```

IMPLEMENTING TREE TRAVERSALS

- In-order Traversal

Algorithm inorder(p):
  if p has a left child lc **then**
    inorder(lc) {r
  perform the "visit" action for position p
  if p has a right child rc **then**
    inorder(rc) {rec

```
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:        # if left child exists, traverse its subtree
    for other in self._subtree_inorder(self.left(p)):
      yield other
  yield p                             # visit p between its subtrees
  if self.right(p) is not None:       # if right child exists, traverse its subtree
    for other in self._subtree_inorder(self.right(p)):
      yield other
```

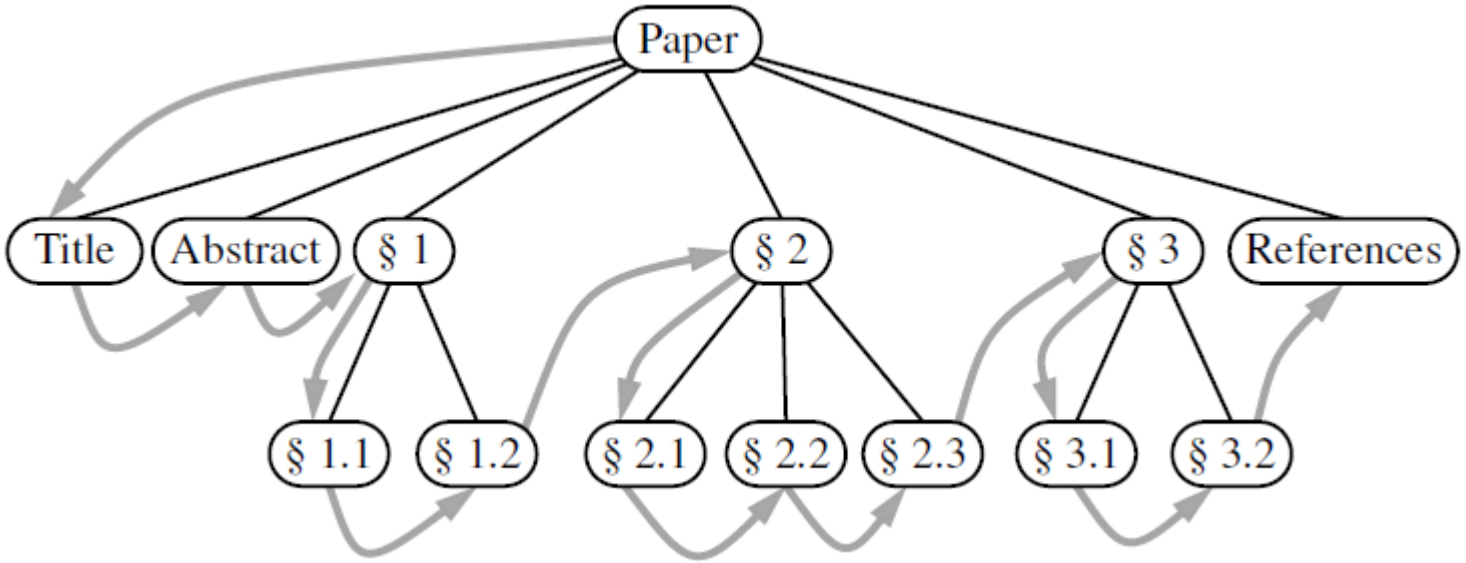
APPLICATION OF TRAVERSALS

- Table of Contents

```
for p in T.preorder():
    print(p.element())
```

```
Paper
Title
Abstract
§1
§1.1
§1.2
§2
§2.1
...
```

```
Paper
  Title
  Abstract
  §1
    §1.1
    §1.2
  §2
    §2.1
    ...
```



APPLICATION OF TRAVERSALS

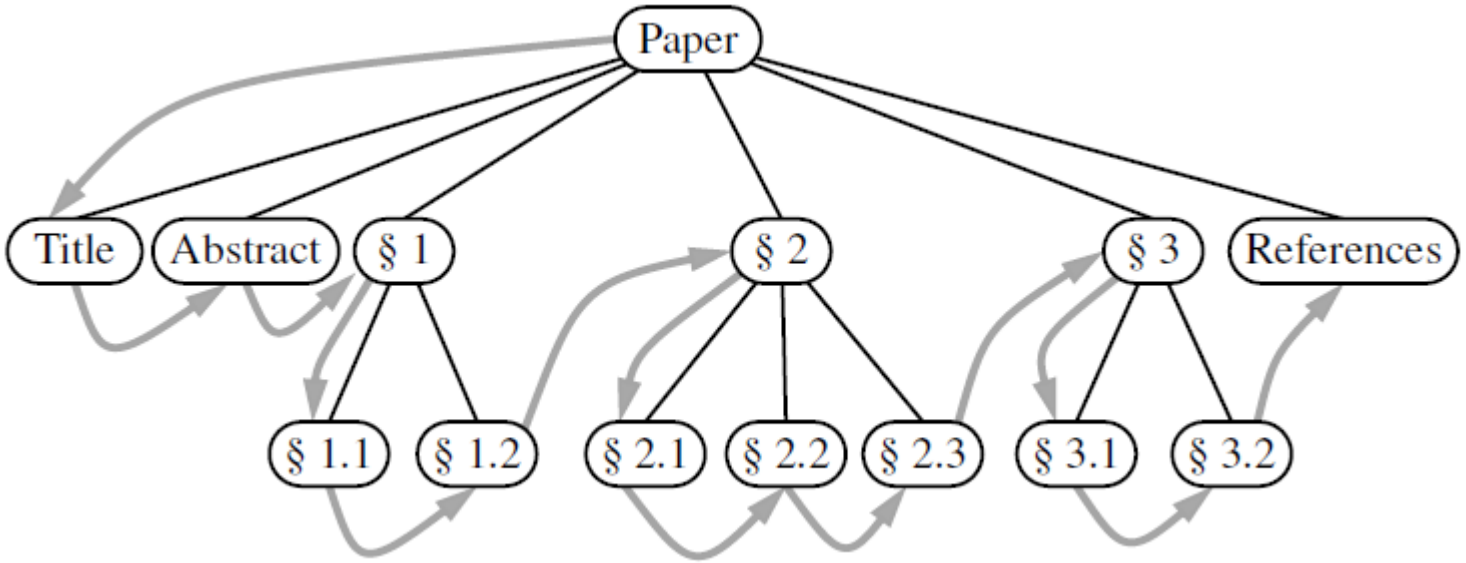
- Table of Contents

```
for p in T.preorder():
    print(2*T.depth(p)*' ' + str(p.element()))
```

Problem? $O(n^2)$

```
Paper
Title
Abstract
§1
§1.1
§1.2
§2
§2.1
...
```

```
Paper
  Title
  Abstract
  §1
    §1.1
    §1.2
  §2
    §2.1
    ...
```



APPLICATION OF TRAVERSALS

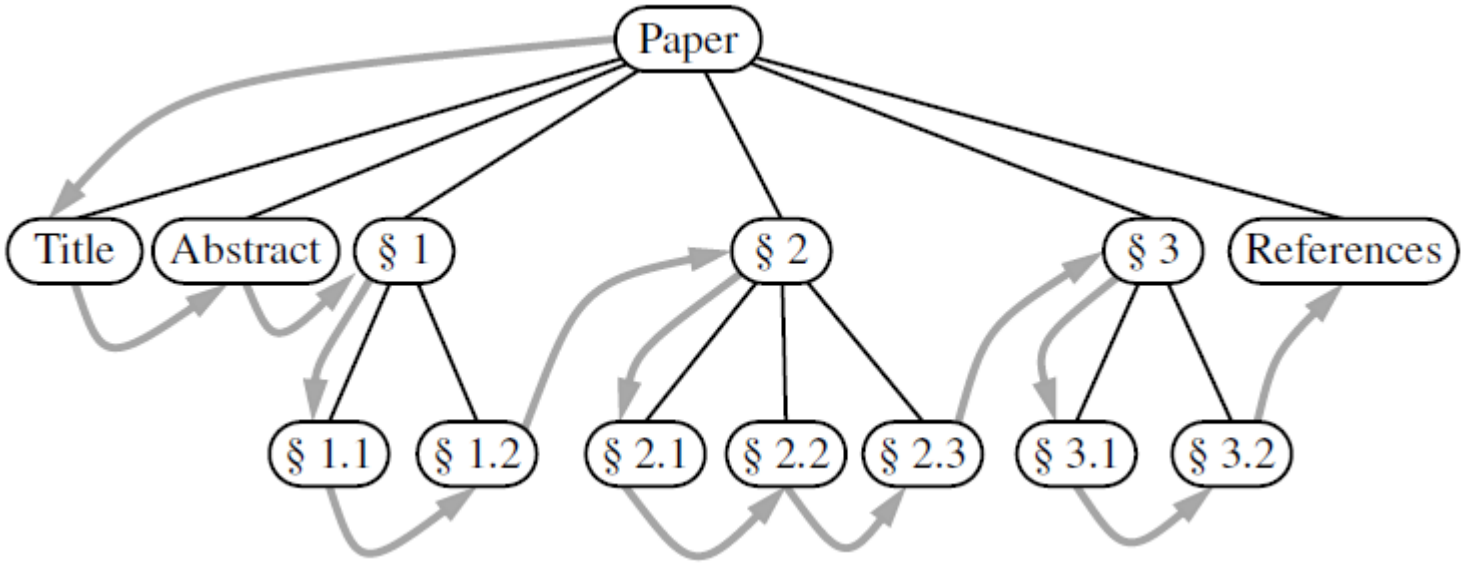
- Table of Contents

```
for p in T.preorder():
    print(p)
```

```
def preorder_indent(T, p, d):
  """Print preorder representation of subtree of T rooted at p at depth d."""
  print(2*d*' ' + str(p.element()))          # use depth for indentation
  for c in T.children(p):
    preorder_indent(T, c, d+1)               # child depth is d+1
```

```
Paper
Title
Abstract
§1
§1.1
§1.2
§2
§2.1
...
```

```
Paper
  Title
  Abstract
  §1
    §1.1
    §1.2
  §2
    §2.1
    ...
```

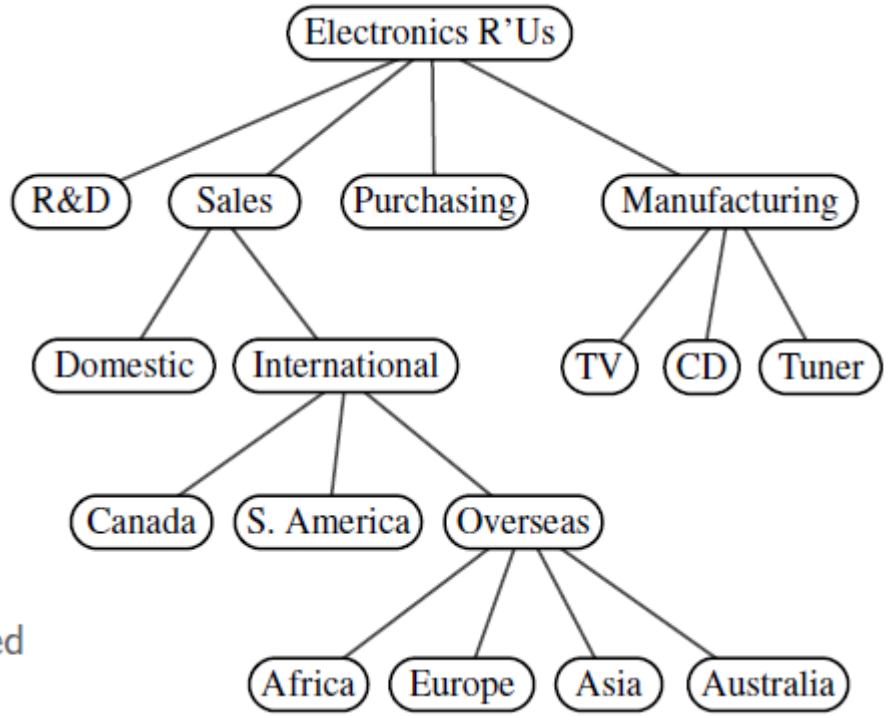


APPLICATION OF TRAVERSALS

- What to do if we want number + label?
- Number: related to index, depth, and siblings

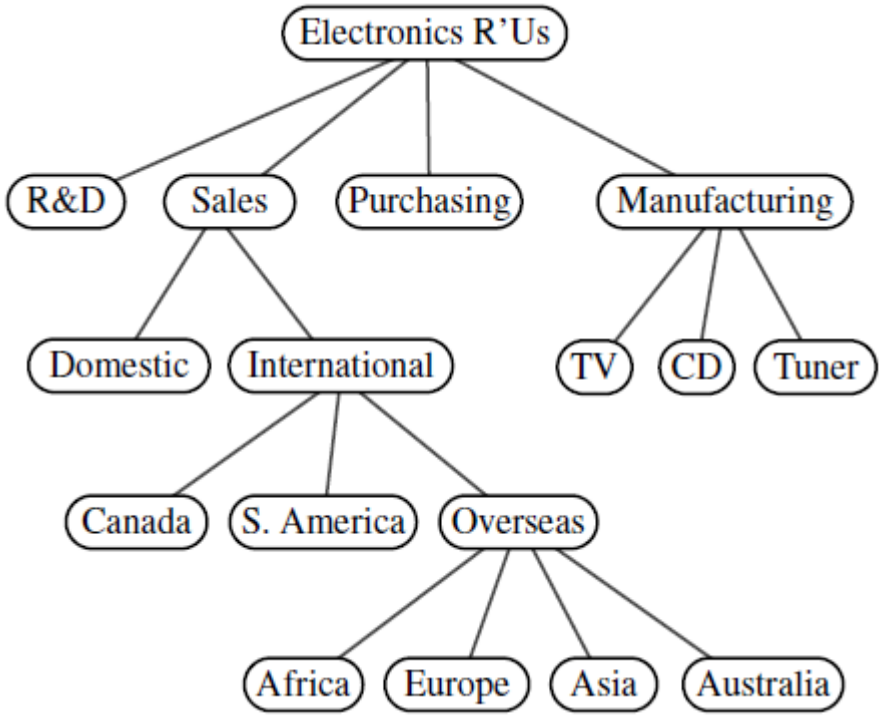
```
Electronics R'Us
  1 R&D
  2 Sales
    2.1 Domestic
    2.2 International
      2.2.1 Canada
      2.2.2 S. America
```

```
def preorder_label(T, p, d, path):
  """Print labeled representation of subtree of T rooted at p at depth d."""
  label = '.'.join(str(j+1) for j in path)    # displayed labels are one-indexed
  print(2*d*'  ' + label, p.element())
  path.append(0)                              # path entries are zero-indexed
  for c in T.children(p):
    preorder_label(T, c, d+1, path)           # child depth is d+1
    path[-1] += 1
  path.pop()
```



APPLICATION OF TRAVERSALS

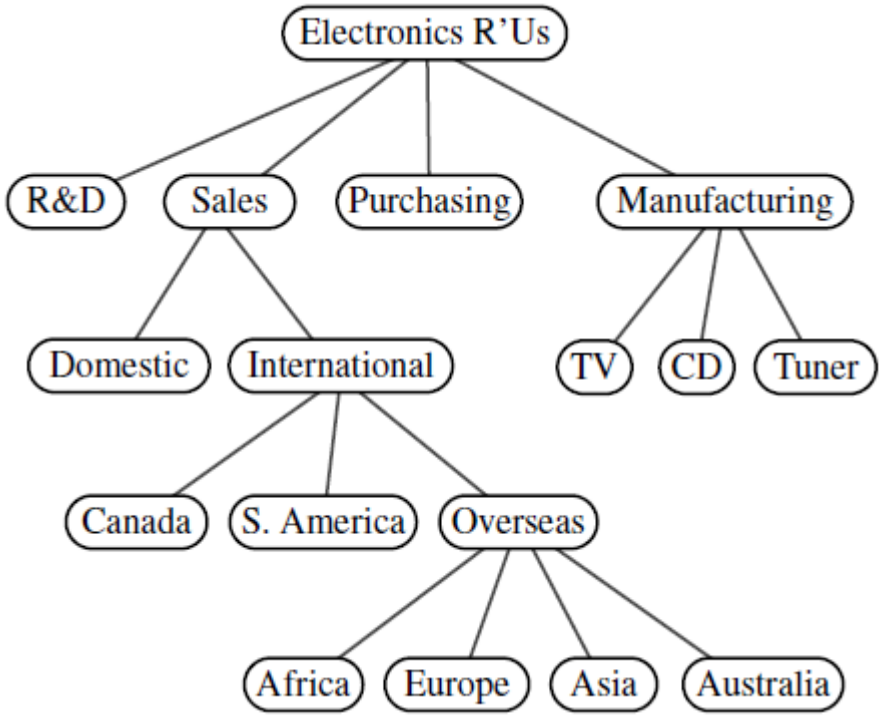
- Parenthetic representation of a Tree
- We can represent a Tree with a String
  - Each level in depth is surrounded with parentheses (and)
- If T consists of a single position p, then
  - P(t) = str(p.element)
- Else
  - P(t) = str(p.element) + '(' + P(T1) + ', ' + … P(Tk) + ')'




```
Electronics R'Us (R&D, Sales (Domestic, International (Canada,
S. America, Overseas (Africa, Europe, Asia, Australia))),
Purchasing, Manufacturing (TV, CD, Tuner))
```

APPLICATION OF TRAVERSALS

```
Electronics R'Us (R&D, Sales (Domestic, International (Canada,
S. America, Overseas (Africa, Europe, Asia, Australia))),
Purchasing, Manufacturing (TV, CD, Tuner))
```




```
def parenthesize(T, p):
  """Print parenthesized representation of subtree of T rooted at p."""
  print(p.element(), end='')          # use of end avoids trailing newline
  if not T.is_leaf(p):
    first_time = True
    for c in T.children(p):
      sep = ' (' if first_time else ', '          # determine proper separator
      print(sep, end='')
      first_time = False              # any future passes will not be the first
      parenthesize(T, c)              # recur on child
    print(')', end='')                # include closing parenthesis
```

FILE SYSTEMS

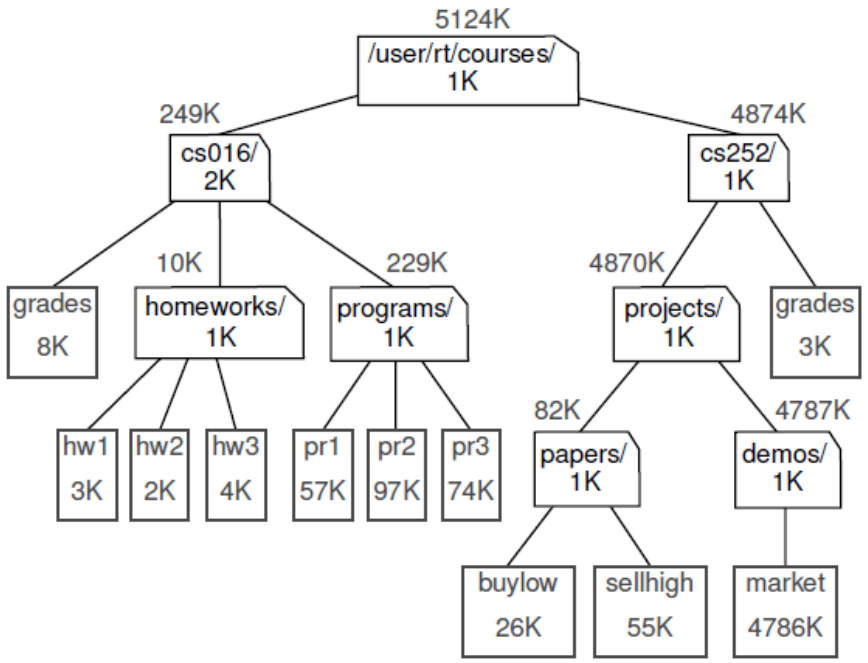
Algorithm DiskUsage(path):

Input: A string designating a path to a file-system entry

Output: The cumulative disk space used by that entry and any nested entries

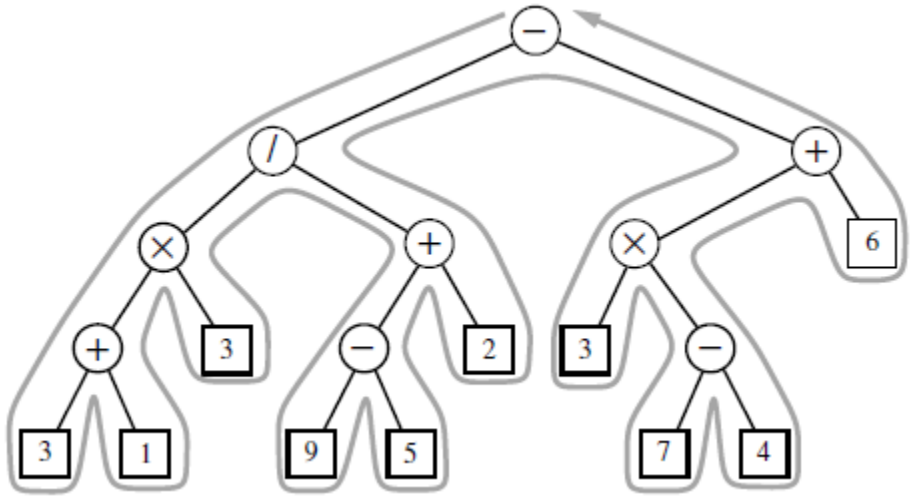
```
total = size(path)                    {immediate disk space used by the entry}
if path represents a directory then
    for each child entry stored within directory path do
        total = total + DiskUsage(child)                {recursive call}
return total
```

```
def disk_space(T, p):
  """Return total disk space for subtree of T rooted at p."""
  subtotal = p.element( ).space( )          # space used at position p
  for c in T.children(p):
    subtotal += disk_space(T, c)            # add child's space to subtotal
  return subtotal
```



EULER TOURS AND THE TEMPLATE METHOD PATTERN

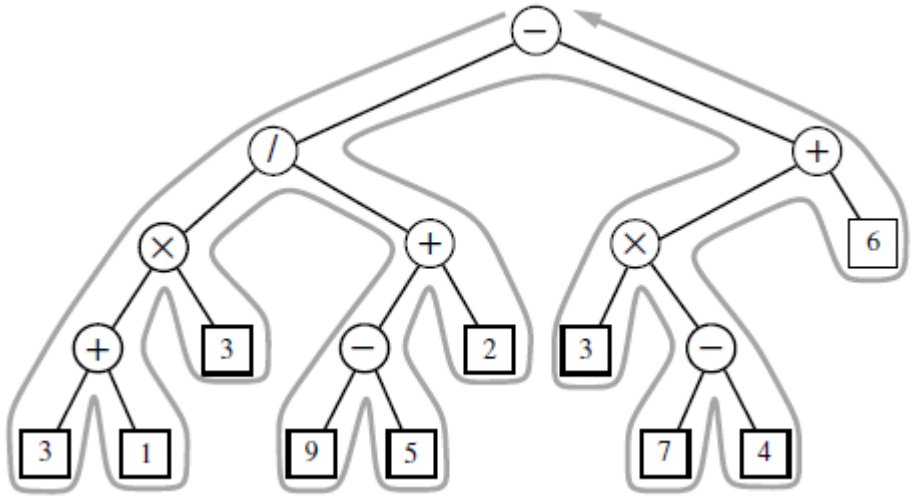
- Euler tour traversal: a "walk" around T
- Each edge is visited twice
- Each node visited once
- O(n) complexity
- For each position p:
- A "pre visit" – before visiting p's subtree
  - The walk passes left of the node
- A "post visit" – after visiting p's subtree
  - The walk passes to the right of the node



EULER TOURS AND THE TEMPLATE METHOD PATTERN

- Euler tour traversal: a "walk" around T
- A "pre visit" – before visiting p's subtree
  - The walk passes left of the node
- A "post visit" – after visiting p's subtree
  - The walk passes to the right of the node

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



EULER TOURS AND THE TEMPLATE METHOD PATTERN

- Implementation: reusable and adaptable code
- Constructor
- Tree() returns _tree instance

```
class EulerTour:
  """Abstract base class for performing Euler tour of a tree.

  _hook_previsit and _hook_postvisit may be overridden by subclasses.
  """
  def __init__(self, tree):
    """Prepare an Euler tour template for given tree."""
    self._tree = tree

  def tree(self):
    """Return reference to the tree being traversed."""
    return self._tree
```

EULER TOURS AND THE TEMPLATE METHOD PATTERN

- Implementation: reusable and adaptable code
- Execute() calls the touring algorithm
- _hook.previsit() performs the pre visit algorithm
- _hook.postvisit() performs the post visit algorithm

```
def execute(self):
  """Perform the tour and return any result from post visit of root."""
  if len(self._tree) > 0:
    return self._tour(self._tree.root( ), 0, [ ])          # start the recursion

def _tour(self, p, d, path):
  """Perform tour of subtree rooted at Position p.

  p        Position of current node being visited
  d        depth of p in the tree
  path     list of indices of children on path from root to p
  """
  self._hook_previsit(p, d, path)                          # "pre visit" p
  results = [ ]
  path.append(0)          # add new index to end of path before recursion
  for c in self._tree.children(p):
    results.append(self._tour(c, d+1, path))     # recur on child's subtree
    path[−1] += 1         # increment index
  path.pop( )             # remove extraneous index from end of path
  answer = self._hook_postvisit(p, d, path, results)       # "post visit" p
  return answer
```

EULER TOURS AND THE TEMPLATE METHOD PATTERN

- Implementation: reusable and adaptable code
- _hook.previsit() performs the pre visit algorithm
  - Left empty so that it can be overridden
- _hook.postvisit() performs the post visit algorithm
  - Left empty so that it can be overridden

```
def _hook_previsit(self, p, d, path):                    # can be overridden
  pass

def _hook_postvisit(self, p, d, path, results):          # can be overridden
  pass
```

EULER TOURS AND THE TEMPLATE METHOD PATTERN

- _hook_previsit(p, d, path)
  - Called once for each position
  - Immediately before p's subtrees are traversed
  - p is the position of the tree
  - d is the depth of p
  - path is a list of indices
- _hook_postvisit(p, d, path, results)
  - Called once for each position
  - Immediately after p's subtrees are traversed
  - Result is a list of objects that are provided as return values from the post visits of the subtrees of p

```
def _hook_previsit(self, p, d, path):                    # can be overridden
  pass

def _hook_postvisit(self, p, d, path, results):          # can be overridden
  pass
```

EULER TOURS AND THE TEMPLATE METHOD PATTERN

- Print label with indentation

```
class PreorderPrintIndentedTour(EulerTour):
  def _hook_previsit(self, p, d, path):
    print(2*d*' ' + str(p.element()))
```

```
def execute(self):
  """Perform the tour and return any result from post visit of root."""
  if len(self._tree) > 0:
    return self._tour(self._tree.root(), 0, [])          # start the recursion

def _tour(self, p, d, path):
  """Perform tour of subtree rooted at Position p.

  p        Position of current node being visited
  d        depth of p in the tree
  path     list of indices of children on path from root to p
  """
  self._hook_previsit(p, d, path)                          # "pre visit" p
  results = []
  path.append(0)          # add new index to end of path before recursion
  for c in self._tree.children(p):
    results.append(self._tour(c, d+1, path))      # recur on child's subtree
    path[-1] += 1         # increment index
  path.pop()              # remove extraneous index from end of path
  answer = self._hook_postvisit(p, d, path, results)       # "post visit" p
  return answer
```

EULER TOURS AND THE TEMPLATE METHOD PATTERN

- Print label with indentation

```
class PreorderPrintIndentedTour(EulerTour):
  def _hook_previsit(self, p, d, path):
    print(2*d*' ' + str(p.element()))
```

```
tour = PreorderPrintIndentedTour(T)
tour.execute()
```

```
def execute(self):
  """Perform the tour and return any result from post visit of root."""
  if len(self._tree) > 0:
    return self._tour(self._tree.root(), 0, [])          # start the recursion

def _tour(self, p, d, path):
  """Perform tour of subtree rooted at Position p.

  p        Position of current node being visited
  d        depth of p in the tree
  path     list of indices of children on path from root to p
  """
  self._hook_previsit(p, d, path)                          # "pre visit" p
  results = []
  path.append(0)          # add new index to end of path before recursion
  for c in self._tree.children(p):
    results.append(self._tour(c, d+1, path))      # recur on child's subtree
    path[-1] += 1         # increment index
  path.pop()              # remove extraneous index from end of path
  answer = self._hook_postvisit(p, d, path, results)       # "post visit" p
  return answer
```

EULER TOURS AND THE TEMPLATE METHOD PATTERN

- Print tree with numbers + labels

```
class PreorderPrintIndentedLabeledTour(EulerTour):
  def _hook_previsit(self, p, d, path):
    label = '.'.join(str(j+1) for j in path)     # labels
    print(2*d*'  ' + label, p.element())
```

```
def execute(self):
  """Perform the tour and return any result from post visit of root."""
  if len(self._tree) > 0:
    return self._tour(self._tree.root(), 0, [])          # start the recursion

def _tour(self, p, d, path):
  """Perform tour of subtree rooted at Position p.

  p        Position of current node being visited
  d        depth of p in the tree
  path     list of indices of children on path from root to p
  """
  self._hook_previsit(p, d, path)                          # "pre visit" p
  results = []
  path.append(0)          # add new index to end of path before recursion
  for c in self._tree.children(p):
    results.append(self._tour(c, d+1, path))      # recur on child's subtree
    path[-1] += 1         # increment index
  path.pop()              # remove extraneous index from end of path
  answer = self._hook_postvisit(p, d, path, results)          # "post visit" p
  return answer
```

EULER TOURS AND THE TEMPLATE METHOD PATTERN

- Print tree as a parenthesized string

```
class ParenthesizeTour(EulerTour):
  def _hook_previsit(self, p, d, path):
    if path and path[−1] > 0:
      print(', ', end='')
    print(p.element( ), end='')
    if not self.tree( ).is_leaf(p):
      print('  (', end='')

  def _hook_postvisit(self, p, d, path, results):
    if not self.tree( ).is_leaf(p):
      print(')', end='')
```

```
def execute(self):
  """Perform the tour and return any result from post visit of root."""
  if len(self._tree) > 0:
    return self._tour(self._tree.root( ), 0, [ ])          # start the recursion

def _tour(self, p, d, path):
  """Perform tour of subtree rooted at Position p.

  p        Position of current node being visited
  d        depth of p in the tree
  path     list of indices of children on path from root to p
  """
  self._hook_previsit(p, d, path)                          # "pre visit" p
  results = [ ]
  path.append(0)          # add new index to end of path before recursion
  for c in self._tree.children(p):
    results.append(self._tour(c, d+1, path))      # recur on child's subtree
    path[−1] += 1         # increment index
  path.pop( )             # remove extraneous index from end of path
  answer = self._hook_postvisit(p, d, path, results)          # "post visit" p
  return answer
```

EULER TOURS AND THE TEMPLATE METHOD PATTERN

- Computing disk space

```
class DiskSpaceTour(EulerTour):
  def _hook_postvisit(self, p, d, path, results):
    # we simply add space associated with p to th
    return p.element( ).space( ) + sum(results)
```

```
def execute(self):
  """Perform the tour and return any result from post visit of root."""
  if len(self._tree) > 0:
    return self._tour(self._tree.root( ), 0, [ ])          # start the recursion

def _tour(self, p, d, path):
  """Perform tour of subtree rooted at Position p.

  p        Position of current node being visited
  d        depth of p in the tree
  path     list of indices of children on path from root to p
  """
  self._hook_previsit(p, d, path)                          # "pre visit" p
  results = [ ]
  path.append(0)          # add new index to end of path before recursion
  for c in self._tree.children(p):
    results.append(self._tour(c, d+1, path))      # recur on child's subtree
    path[−1] += 1         # increment index
  path.pop( )             # remove extraneous index from end of path
  answer = self._hook_postvisit(p, d, path, results)       # "post visit" p
  return answer
```

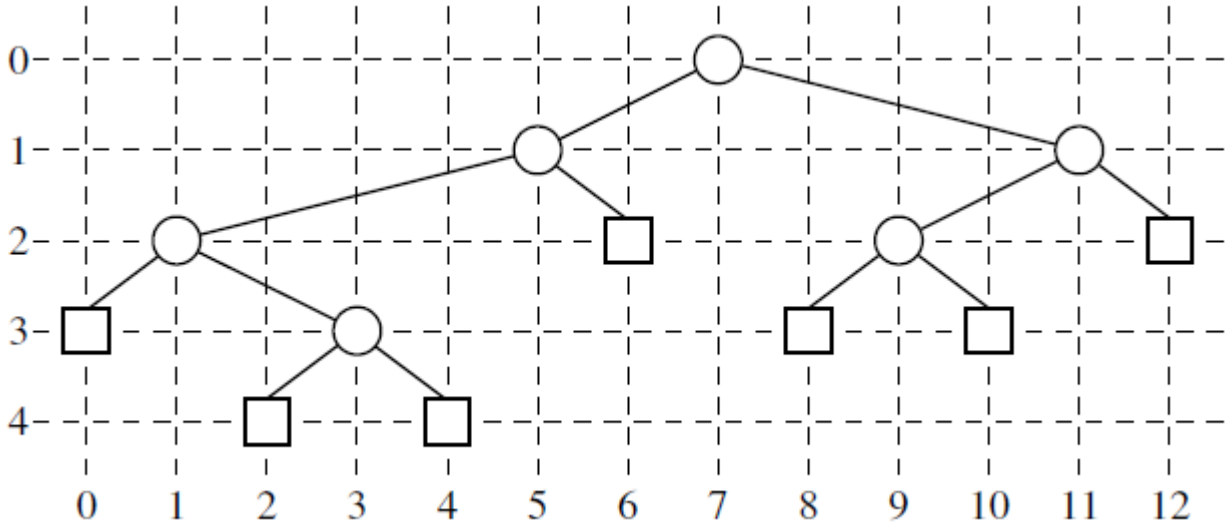
EULER TOUR TRAVERSAL OF A BINARY TREE

- Binary tree traversal
- Additional _hook_invisit() function

```
class BinaryEulerTour(EulerTour):

  def _tour(self, p, d, path):
    results = [None, None]          # will update with results of recursions
    self._hook_previsit(p, d, path)                    # "pre visit" for p
    if self._tree.left(p) is not None:                 # consider left child
      path.append(0)
      results[0] = self._tour(self._tree.left(p), d+1, path)
      path.pop()
    self._hook_invisit(p, d, path)                     # "in visit" for p
    if self._tree.right(p) is not None:                # consider right child
      path.append(1)
      results[1] = self._tour(self._tree.right(p), d+1, path)
      path.pop()
    answer = self._hook_postvisit(p, d, path, results)       # "post visit" p
    return answer

  def _hook_invisit(self, p, d, path): pass            # can be overridden
```

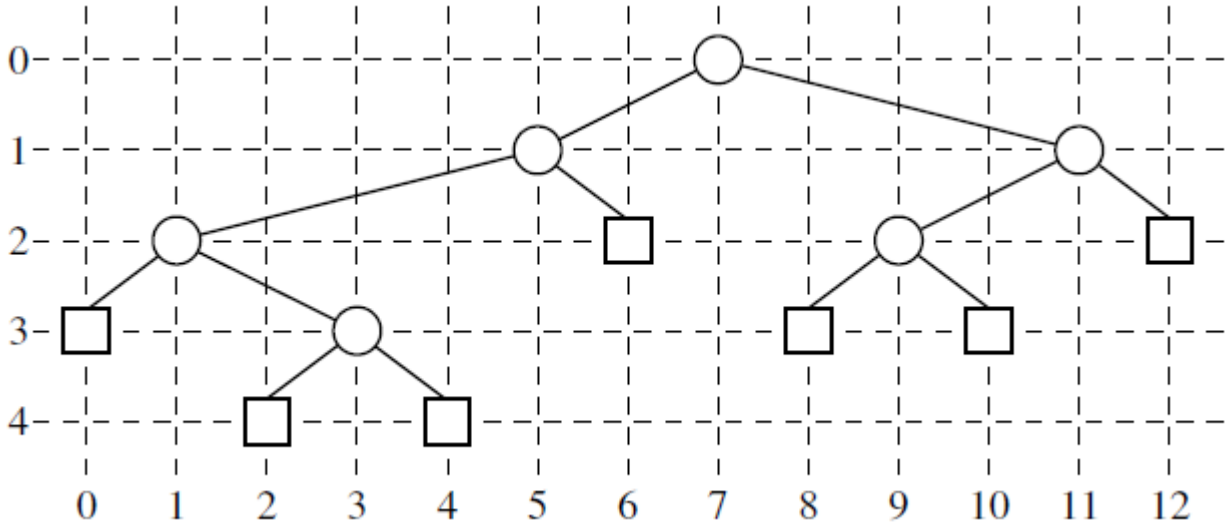
EULER TOUR TRAVERSAL OF A BINARY TREE

- Binary tree traversal
- Compute a graphical layout of a binary tree (with X-Y coordinates)
- Rules:
  - X(p) is the number of positions visited before p in an inorder traversal of T
  - Y(p) is the depth of p in T



EULER TOUR TRAVERSAL OF A BINARY TREE

```
class BinaryLayout(BinaryEulerTour):
  """Class for computing (x,y) coordi
  def __init__(self, tree):
    super().__init__(tree)
    self._count = 0

  def _hook_invisit(self, p, d, path):
    p.element().setX(self._count)
    p.element().setY(d)
    self._count += 1
```



QUIZ OF THE WEEK

- 100 passengers to board a plane
- Assume each passenger should take their own seat assigned to them
- If the first passenger disregard the rule and pick a random seat
- For the rest 99 passengers, if their seats are taken, they will pick another random seat to sit in
- What's the probability that the 100th passenger sits on his/her assigned seat?

THANKS

See you in the next session!