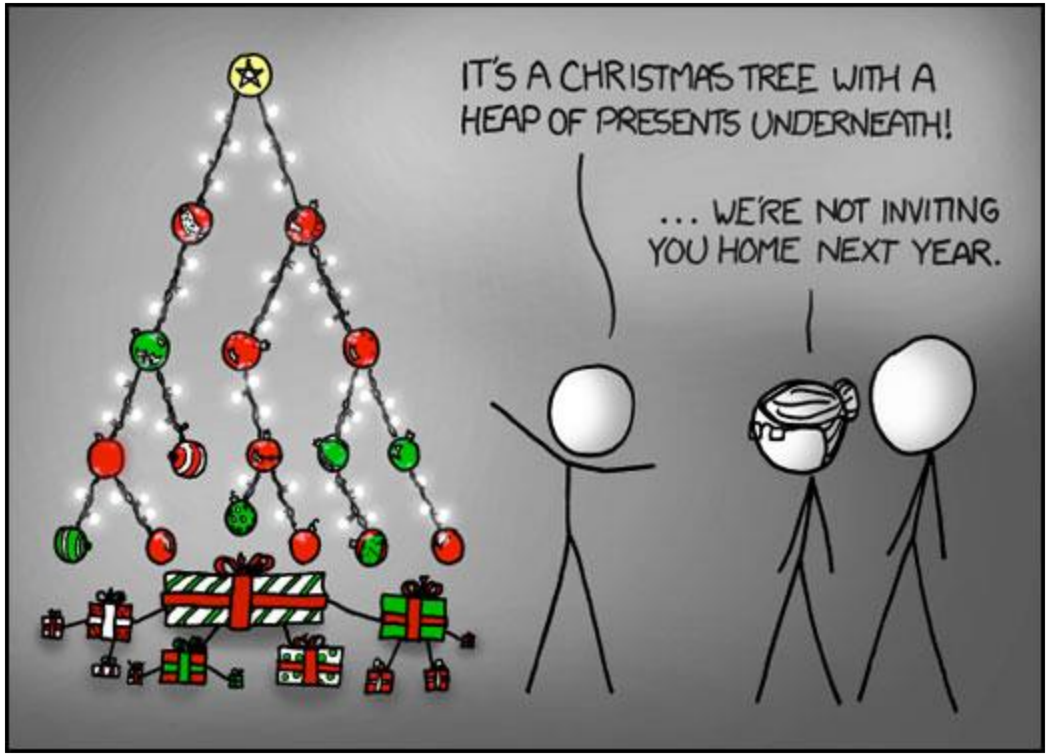
PRIORITY QUEUES

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
- Euler tour tree traversal

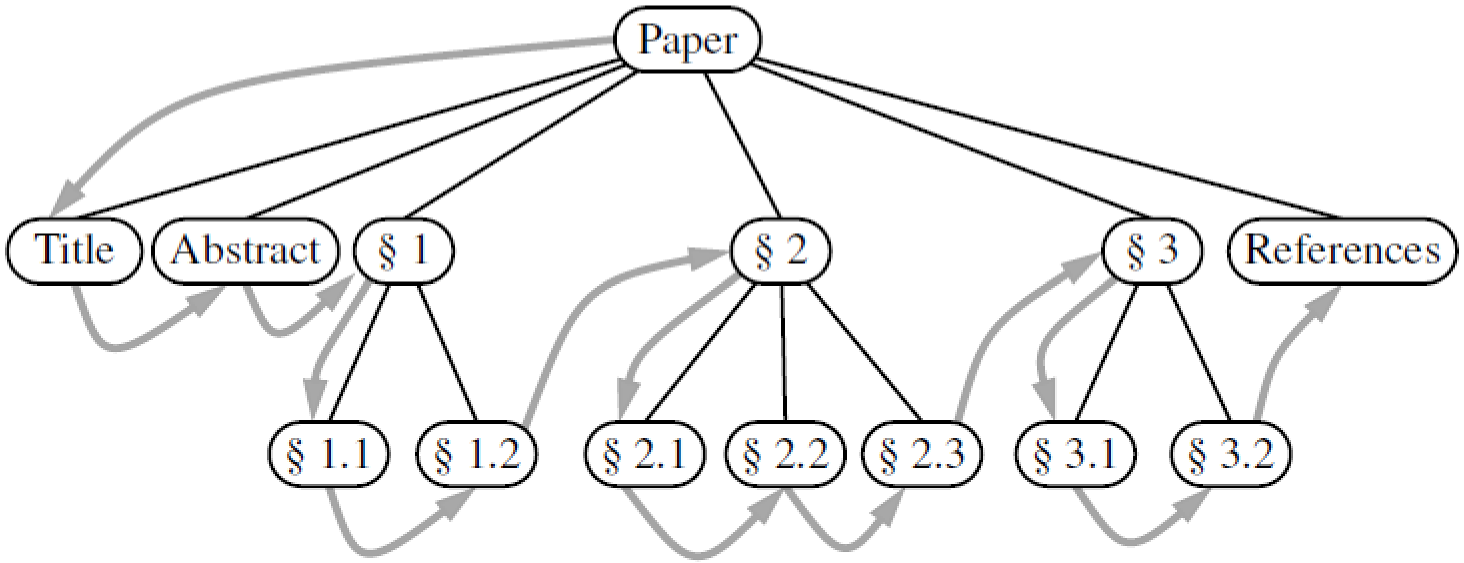


PREVIOUS LECTURE

- 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

TREE TRAVERSAL （遍历） ALGORITHMS

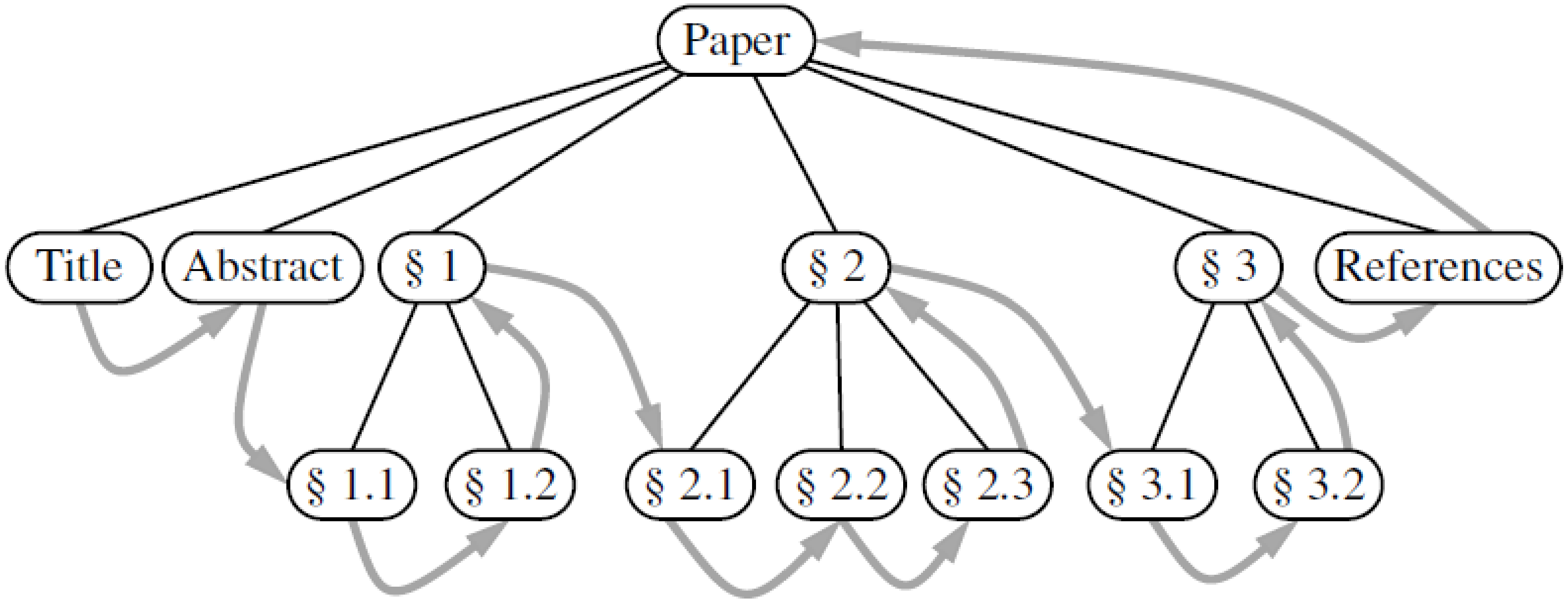
- Traversal: a systematic way of accessing or ‘visiting’ all the positions of T
- Preorder traversal （正序遍历）:
  - visit root of T first and then visit the sub-trees recursively
  - If tree is ordered, then the subtrees are traversed according to the order of the children



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

TREE TRAVERSAL （遍历） ALGORITHMS

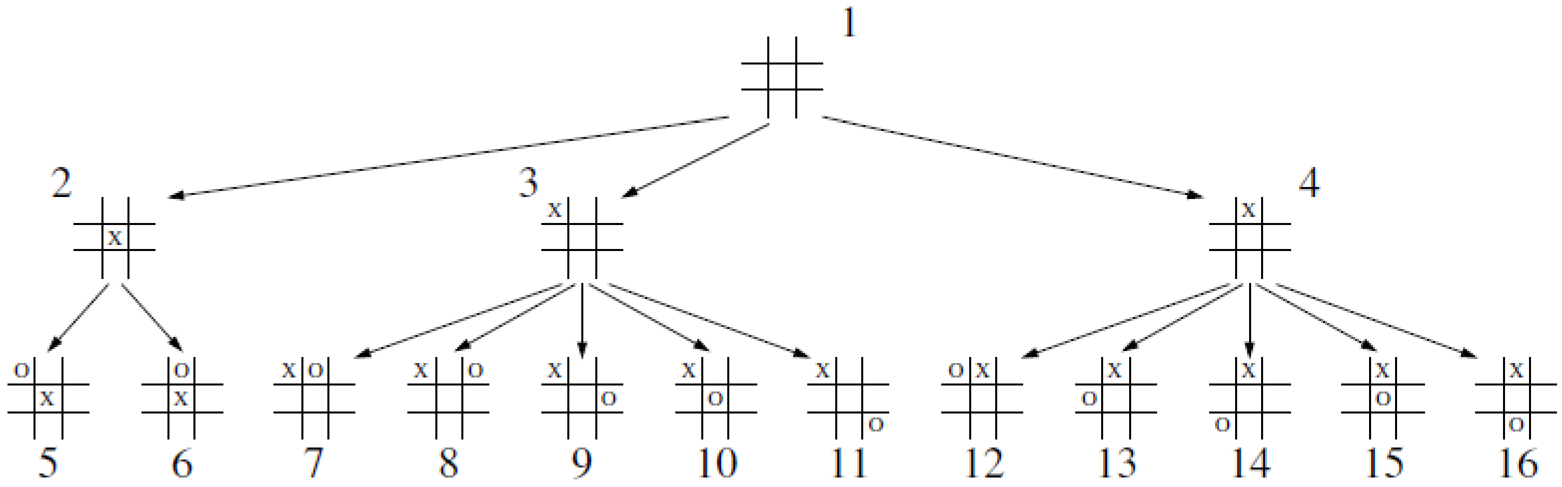
- Postorder traversal （逆序遍历）：
  - The opposite of preorder traversal
  - Recursively traverses the subtrees at the children of the root first and then visits the root



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

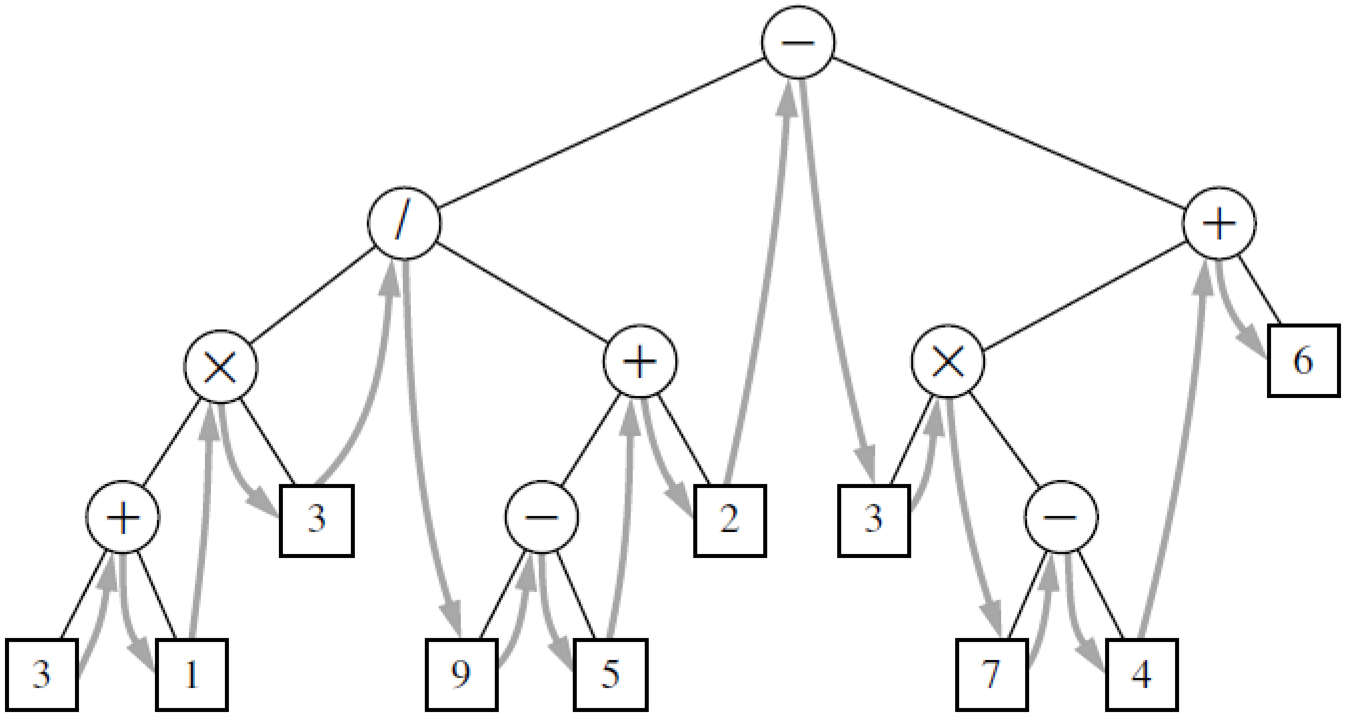
TREE TRAVERSAL （遍历） ALGORITHMS

- 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

- 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

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
- Fringge (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}
```

```python
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
```

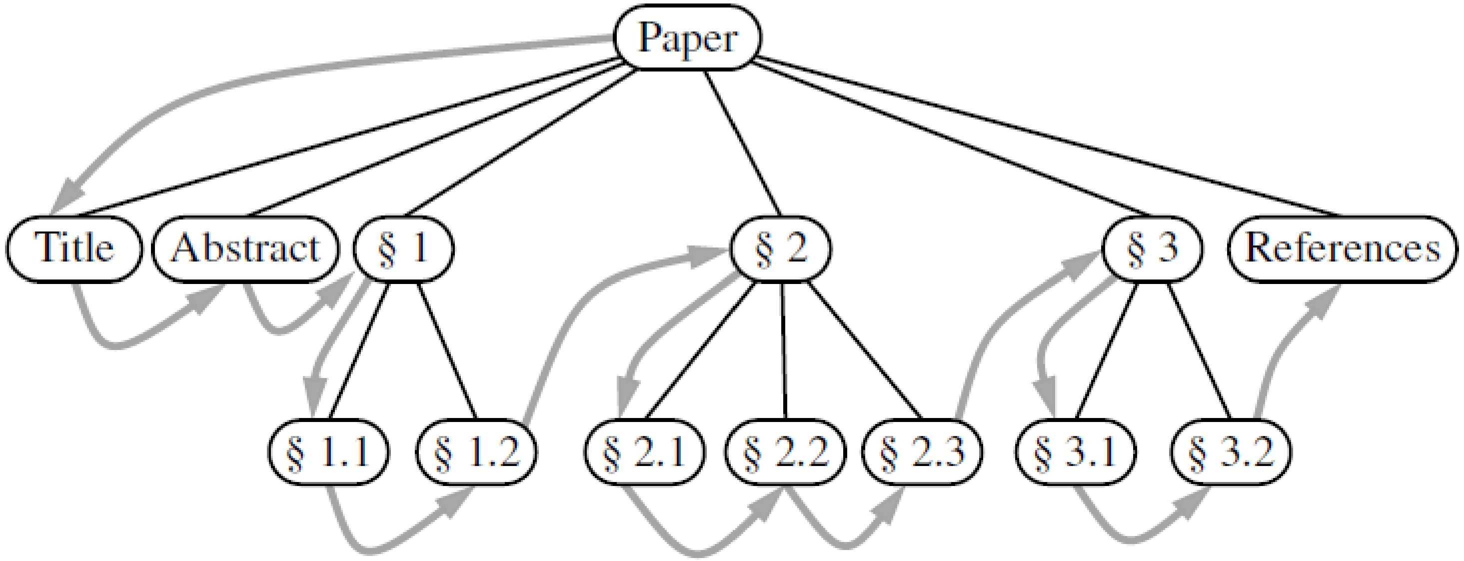
APPLICATION OF TRAVERSALS

- 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

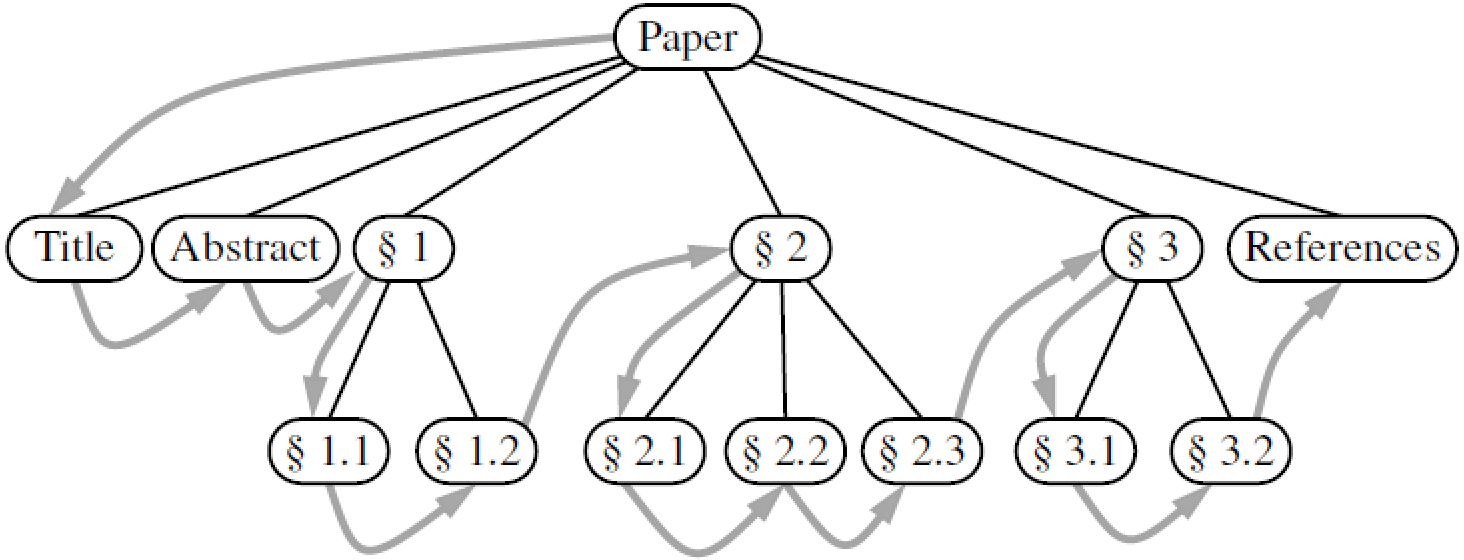
- Table of Contents

```
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

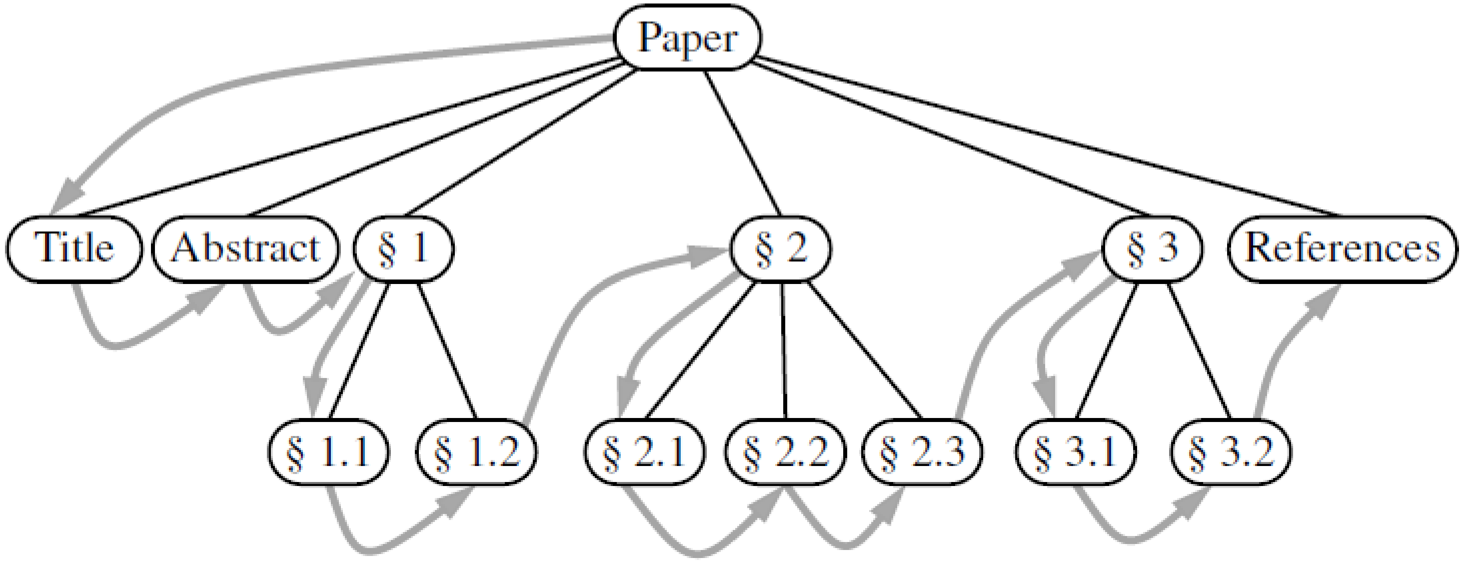
- Table of Contents

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
    ...
```

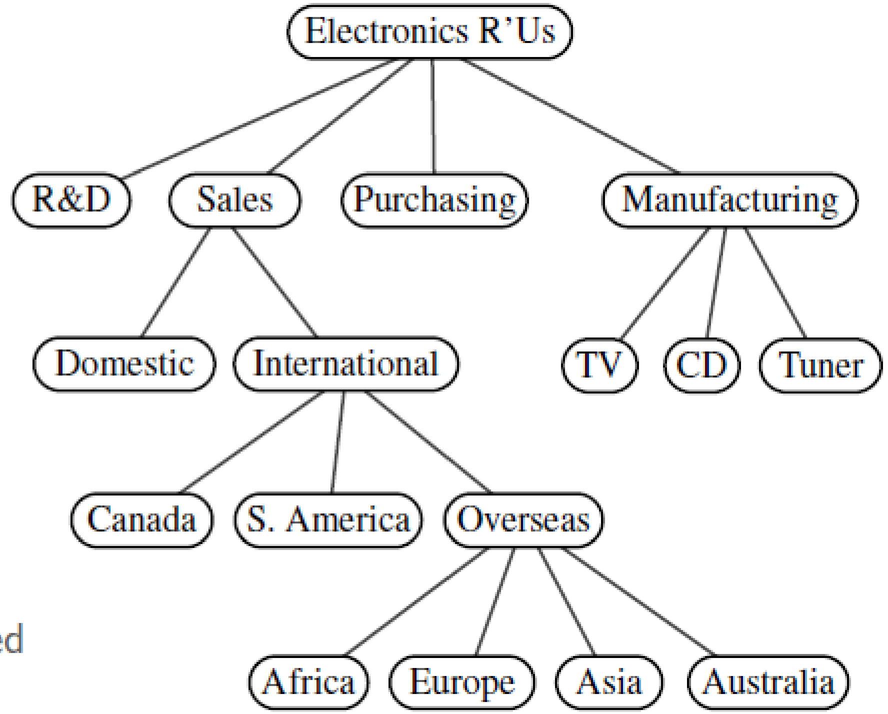


APPLICATION OF TRAVERSALS

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

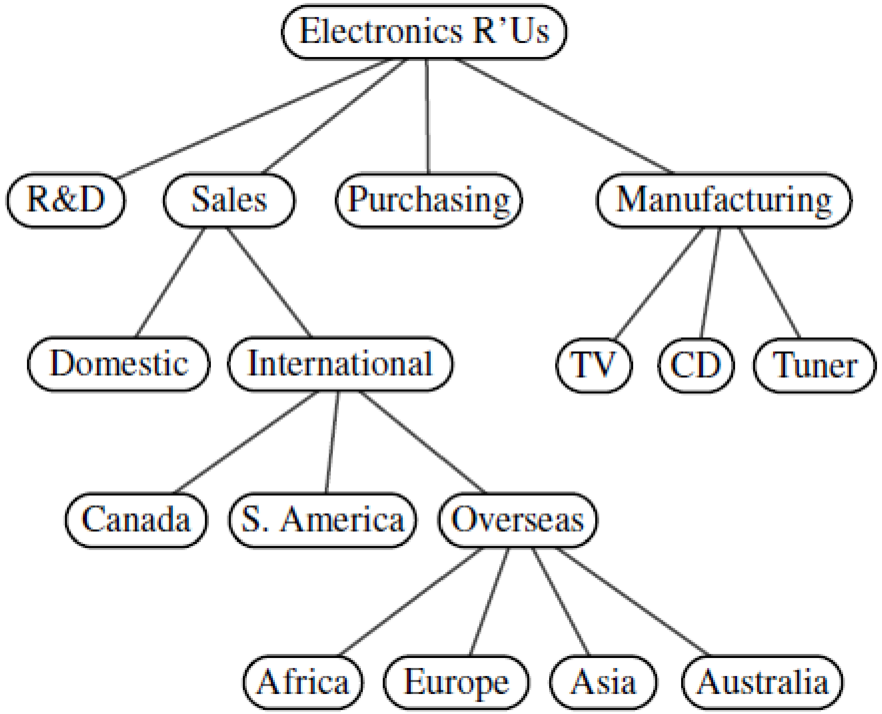
```
Electronics R'Us
  1 R&D
  2 Sales
    2.1 Domestic
    2.2 International
      2.2.1 Canada
      2.2.2 S. America
```

```
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

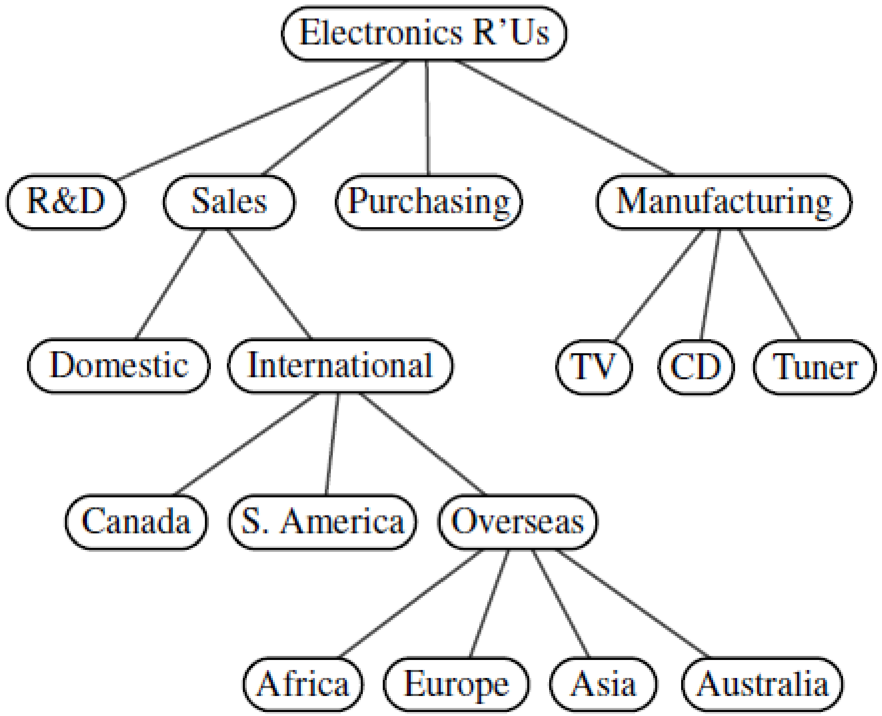
- Parenthetic representation of a Tree
- We can represent a Tree with a String
  - Each level in depth is surrounded with parentheses (and)
- If T consists of a single position p, then
  - P(t) = str(p.element)
- Else
  - P(t) = str(p.element) + '(' + P(T1) + ', ' + … P(Tk) + ')'




```
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))
```

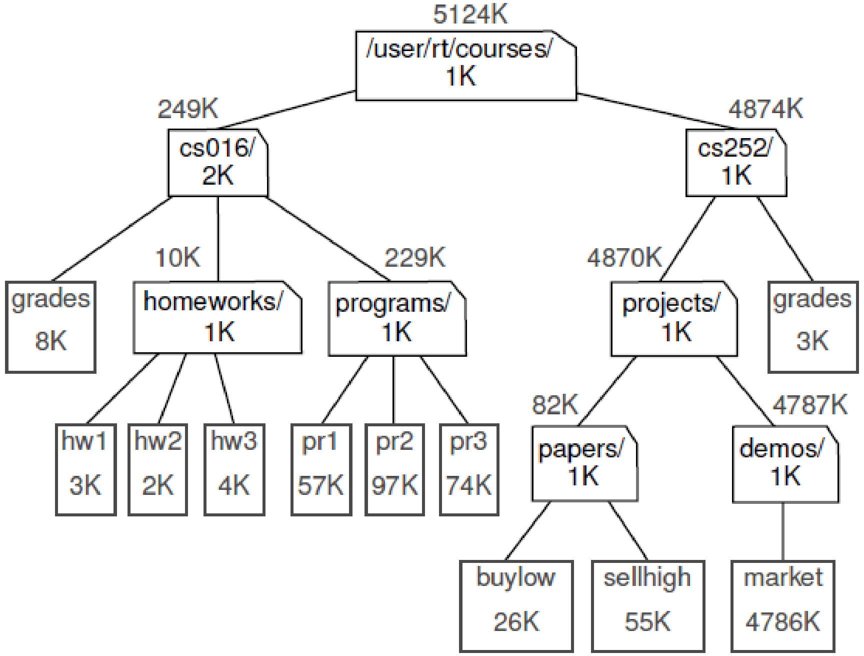



```python
def parenthesize(T, p):
  """Print parenthesized representation of subtree of T rooted at p."""
  print(p.element(), end='')               # use of end avoids trailing newline
  if not T.is_leaf(p):
    first_time = True
    for c in T.children(p):
      sep = ' (' if first_time else ', '        # 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

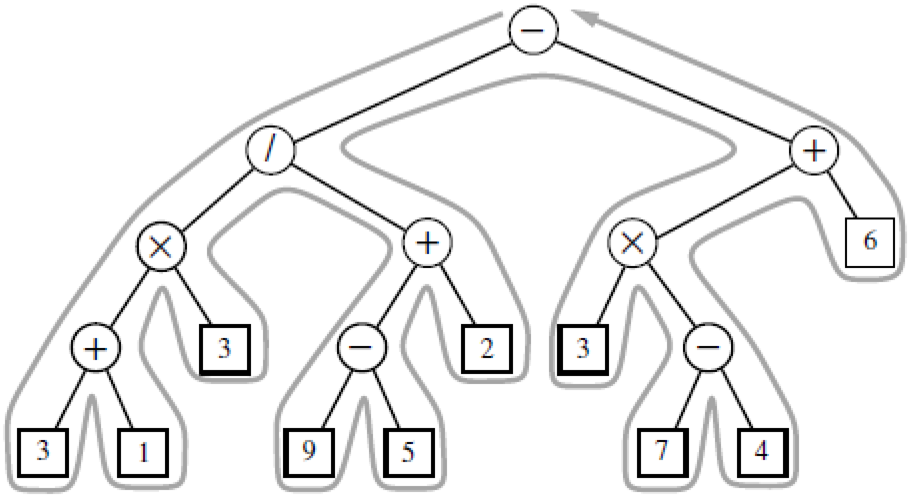
```
total = size(path)                          {immediate disk space used by the entry}
if path represents a directory then
    for each child entry stored within directory path do
        total = total + DiskUsage(child)                    {recursive call}
return total
```




```python
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

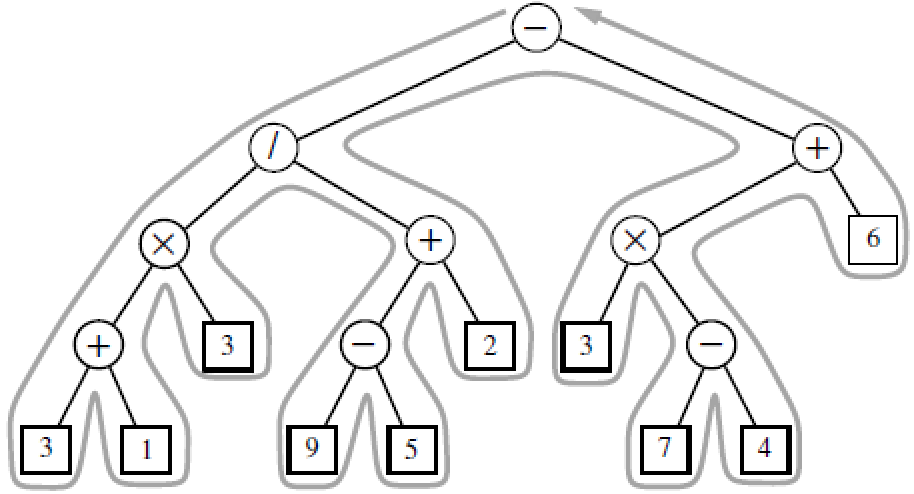
- Euler tour traversal: a “walk” around T
- Each edge is visited twice
- Each node visited once
- O(n) complexity
- For each position p:
- A “pre visit” – as pre-order
  - The walk passes left of the node
- A “post visit” – as post-order
  - The walk passes to the right of the node



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)     # label
    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

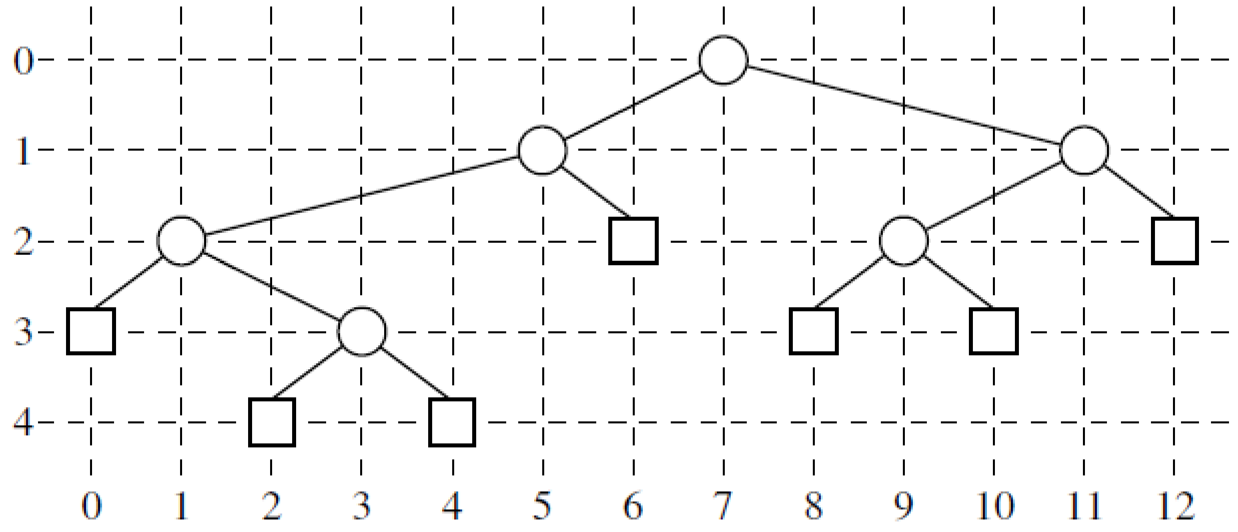
```
class BinaryEulerTour(EulerTour):

  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

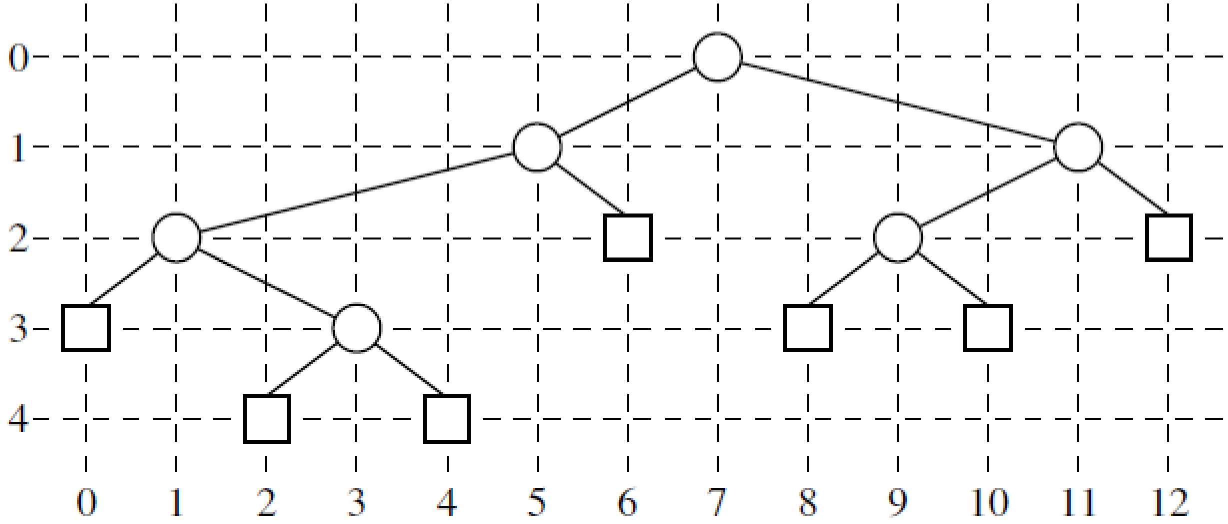
- Binary tree traversal
- Compute a graphical layout of a binary tree (with X-Y coordinates)
- Rules:
  - X(p) is the number of positions visited before p in an inorder traversal of T
  - Y(p) is the depth of p in T



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
```

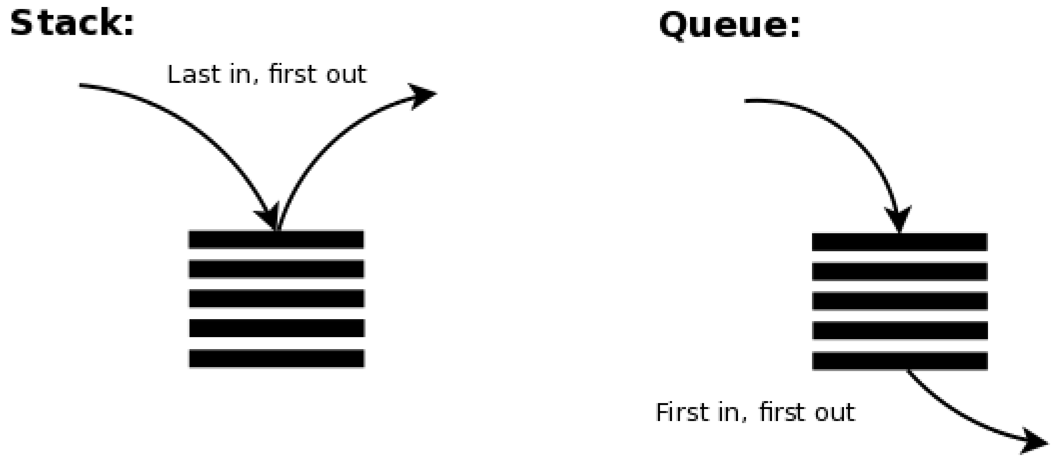


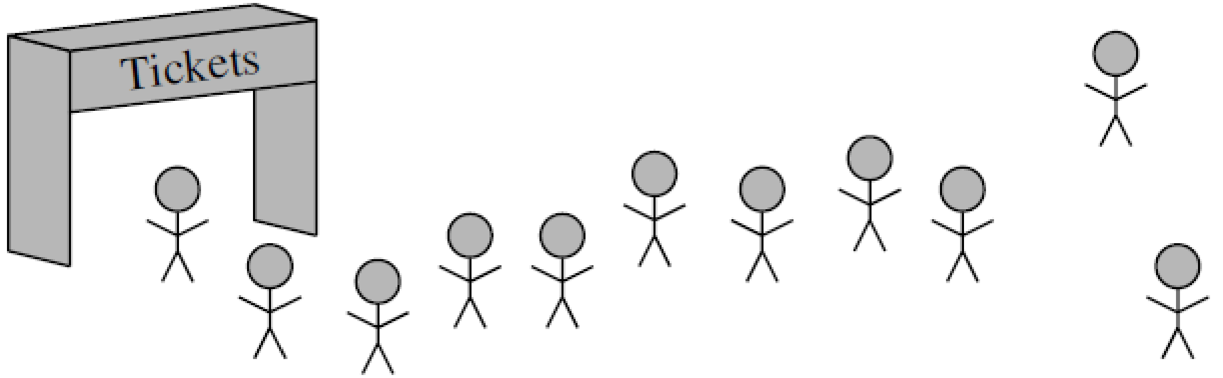
THIS LECTURE

- Priority Queues
- Implementation of Priority Queues
- Heaps
- Implementation of Heaps

QUEUES （队列）

- FIFO principle
  - First-In, First-Out (先进先出)
- Elements can be inserted at any time
- Only the elements that has been in the queue the longest can be removed next
- Applications
  - Reservation centres
  - Process management
  - Web server





QUEUES （队列）

- FIFO principle
  - May not always be useful
- Air traffic control
  - Planes arrives in different times
  - Some planes need priority
    - Time spent on waiting
    - Amount of fuel remaining
- Process scheduling
  - Schedules processes based on their priority
  - A process (e.g. emergency stop) may have the highest priority
- Hence, we need a data structure to cater with such situations
  - Priority queue

PRIORITY QUEUES （优先队列）

- Collection of prioritized elements
- Arbitrary element insertion
- Removal of the element that has first priority
- When an element is added, a priority can be assigned to it with a **key**
- Element with the minimum key will be removed next from the queue
- **Key**s can be other data types, as long as there is a way to compare them
  - E.g. a < b for instances a and b

PRIORITY QUEUES （优先队列）

- Abstract Data Type (ADT)
- P.add(k, v): insert an item with key k and value v into priority queue P
- P.min(): returns a tuple (k, v), representing the key and value of an item in the priority queue P with the minimal key (but do not remove the item)
  - Error when the queue is empty
- P.remove_min(): remove an item with the minimal key, return a tuple (k, v), representing the key-value pair to be removed
  - Error when the queue is empty
- P.is_empty(): returns true when P has no items
- len(p): returns the number of items in the priority queue P

PRIORITY QUEUES （优先队列）

- Multiple entries with equivalent keys
  - remove_min() may pick an arbitrary choice of item
  - We will look at how this is done in other chapters
- For now, an element’s key is fixed
  - We will look at how one may change the priority of an element later

Operation	Return Value	Priority Queue
P.add(5,A)		{(5,A)}
P.add(9,C)		{(5,A), (9,C)}
P.add(3,B)		{(3,B), (5,A), (9,C)}
P.add(7,D)		{(3,B), (5,A), (7,D), (9,C)}
P.min()	(3,B)	{(3,B), (5,A), (7,D), (9,C)}
P.remove_min()	(3,B)	{(5,A), (7,D), (9,C)}
P.remove_min()	(5,A)	{(7,D), (9,C)}
len(P)	2	{(7,D), (9,C)}
P.remove_min()	(7,D)	{(9,C)}
P.remove_min()	(9,C)	{ }
P.is_empty()	True	{ }
P.remove_min()	“error”	{ }

IMPLEMENTING A PRIORITY QUEUE

- Keep track of an element and its key
- Again, we will use the **composition design pattern**
- PriorityQueueBase class
- _Item class
  - Compose **key** and **value**
  - __lt__ to override the “<“ operator

```
class PriorityQueueBase:
  """Abstract base class for a priority queue."""

  class _Item:
    """Lightweight composite to store priority queue items."""
    __slots__ = '_key', '_value'

    def __init__(self, k, v):
      self._key = k
      self._value = v

    def __lt__(self, other):
      return self._key < other._key          # compare items based

  def is_empty(self):                        # concrete method assumin
    """Return True if the priority queue is empty."""
    return len(self) == 0
```

THE POSITIONAL LIST ADT

- Position ADT
  - P.element(): return the element stored at position p
- Positional List ADT
  - L.first(): first element of L, or None if L is empty
  - L.last(): last element of L, or None if L is empty
  - L.before(p): the position in L immediately before p, or None if p is the first position
  - L.after(p): the position in L immediately after p, or None if p is the last position
  - L.is_empty(): true if L is empty
  - len(L): number of elements in the list
  - iter(L): returns a forward iterator for the elements of the list

THE POSITIONAL LIST ADT

- Positional List ADT
  - L.add_first(e): insert a new element e at the front of L, return the position of the new element
  - L.add_last(e): insert a new element e at the back of L, return the position of the new element
  - L.add_before(p, e): insert a new element e before position p in L, return the position of the new element
  - L.add_after(p, e): insert a new element e after position p in L, return the position of the new element
  - L.replace(p, e): replace the element at position p with element e, returning the element formerly at position p
  - L.delete(p): remove and return the element at position p in L

IMPLEMENTING A PRIORITY QUEUE

- Implementation with an Unsorted List
- Instances of _Item are stored within a PositionalList
  - Identified as _data
  - Implemented with a doubly-linked list
  - All operations run in O(1) time
- UnsortedPriorityQueue class
  - Sub-class of PriorityQueueBase
- _find_min(self)

```
class UnsortedPriorityQueue(PriorityQueueBase): # base class
  """A min-oriented priority queue implemented with an unsort

  def _find_min(self):                         # nonpublic utility
    """Return Position of item with minimum key."""
    if self.is_empty():                        # is_empty inherited fr
      raise Empty('Priority queue is empty')
    small = self._data.first()
    walk = self._data.after(small)
    while walk is not None:
      if walk.element() < small.element():
        small = walk
      walk = self._data.after(walk)
    return small
```

IMPLEMENTING A PRIORITY QUEUE

- Implementation with an Unsorted List
- Instances of _Item are stored within a PositionalList
  - Identified as _data
  - Implemented with a doubly-linked list
  - All operations run in O(1) time

```
  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = PositionalList()

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair."""
    self._data.add_last(self._Item(key, value))

  def min(self):
    """Return but do not remove (k,v) tuple with minimum key.
    p = self._find_min()
    item = p.element()
    return (item._key, item._value)
```

IMPLEMENTING A PRIORITY QUEUE

- Implementation with an Unsorted List
- Instances of _Item are stored within a PositionalList
  - Identified as _data
  - Implemented with a doubly-linked list
  - All operations run in O(1) time

```
def remove_min(self):
  """Remove and return (k,v) tuple with minimum key."""
  p = self._find_min()
  item = self._data.delete(p)
  return (item._key, item._value)
```

Operation	Running Time
len	$O(1)$
is_empty	$O(1)$
add	$O(1)$
min	$O(n)$
remove_min	$O(n)$

IMPLEMENTING A PRIORITY QUEUE

- Implementation with a Sorted List
- SortedPriorityQueue class
  - Sub-class of PriorityQueueBase
- Instances of _Item are stored within a PositionalList
  - Identified as _data
  - Implemented with a doubly-linked list
  - All operations run in O(1) time

```
class SortedPriorityQueue(PriorityQueueBase): # base class defin
  """A min-oriented priority queue implemented with a sorted list

  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = PositionalList()

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair."""
    newest = self._Item(key, value)                 # make new i
    walk = self._data.last()            # walk backward looking fo
    while walk is not None and newest < walk.element():
      walk = self._data.before(walk)
    if walk is None:
      self._data.add_first(newest)                  # new key is
    else:
      self._data.add_after(walk, newest)            # newest goe
```

IMPLEMENTING A PRIORITY QUEUE

- Implementation with a Sorted List
- SortedPriorityQueue class
  - Sub-class of PriorityQueueBase
- Instances of _Item are stored within a PositionalList
  - Identified as _data
  - Implemented with a doubly-linked list
  - All operations run in O(1) time

```
  def min(self):
    """Return but do not remove (k,v) tuple with minimum key.
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    p = self._data.first()
    item = p.element()
    return (item._key, item._value)

  def remove_min(self):
    """Remove and return (k,v) tuple with minimum key."""
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    item = self._data.delete(self._data.first())
    return (item._key, item._value)
```

IMPLEMENTING A PRIORITY QUEUE

- Implementation with a Sorted List
- SortedPriorityQueue class
  - Sub-class of PriorityQueueBase
- Instances of _Item are stored within a PositionalList
  - Identified as _data
  - Implemented with a doubly-linked list
  - All operations run in O(1) time

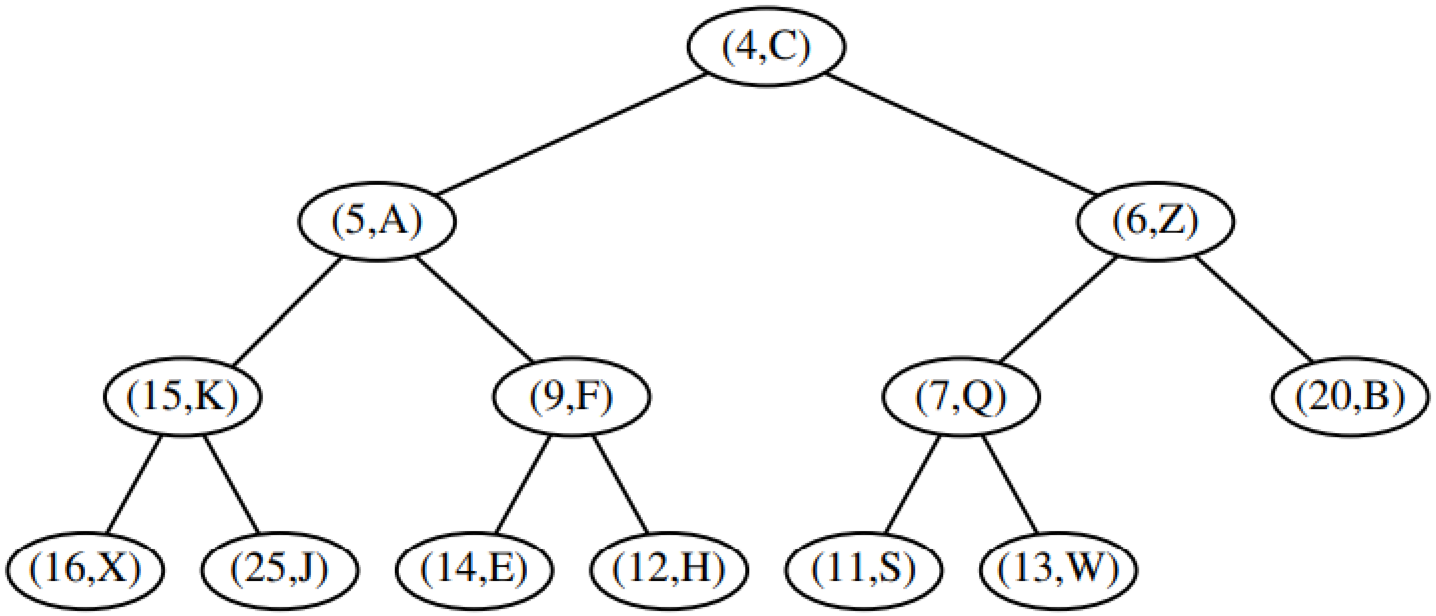
Operation	Unsorted List	Sorted List
len	$O(1)$	$O(1)$
is_empty	$O(1)$	$O(1)$
add	$O(1)$	$O(n)$
min	$O(n)$	$O(1)$
remove_min	$O(n)$	$O(1)$

HEAP（堆）

- Heap: a binary tree that stores a collection of items at its positions
  - A relational property defined in terms of the way keys are stored in T
  - A structural property defined in terms of the shape of T itself
- Relational property (**heap order property**): In a heap T, for every position p other than the root, the key stored at p is greater than or equal to the key stored at p’s parent
- Structural property (**complete binary tree property**): A heap T with height h is a complete binary tree if levels 0, 1, 2, …, h-1 of T have the maximum number of nodes possible (level i has 2^i nodes, for 0<= i <= h-1) and the remaining nodes at level h reside in the lfeftmost possible positions at that level

HEAP（堆）

- Complete
  - Levels 0, 1, and 2 are full
  - 6 nodes in level 3 are in the six leftmost possible positions at that level
- An alternative definition
  - If we are to store a complete binary tree T with n elements in an array A , then its 13 entries would be stored from A[0] to A[n-1]



HEAP（堆）

- Height of a heap
- A heap T storing n entries has height h = floor(log n)
- From the fact that T is complete, we know that the number of nodes in levels 0 through h-1 of T is: 1 + 2 + 4 + … + 2^{h-1} = 2^h -1, and that the number of nodes in level h is at least 1 and at most 2^h. Therefore

$$n \geq 2^h - 1 + 1 = 2^h \qquad \text{and} \qquad n \leq 2^h - 1 + 2^h = 2^{h+1} - 1.$$

- Therefore h <= log n (take log on both sides of 2^h <= n), and log(n+1) -1 <= h (same principle for n <= 2^{h+1} -1)

IMPLEMENTING A PRIORITY QUEUE WITH A HEAP

- Height of a heap h = floor(log n)
- What does it mean in terms of complexity?
- We can perform update operations on a heap in time proportional to its height: O(log n)
- Insertion: add(k, v)
  - To maintain the complete binary tree property,
  - the new node should be placed at a position p just beyond the rightmost node at the bottom level of the tree
  - or as the leftmost position of a new level (if the bottom level is full, or if the heap is empty)
- All sounds good but

IMPLEMENTING A PRIORITY QUEUE WITH A HEAP

- The heap order property need to be maintained as well
- Unless position p is the root of T, we need to compare the key at position p to that of p's parent q.
- If $k_p >= k_q$, the heap order property is satisfied
- If $k_p < k_q$, then need to restore the heap-order property
  - Swap the entries p and q
  - Perform algorithm recursively until the heap order property is restored
- Up-heap bubbling: keep going upwards until the heap order is restored
- Complexity?

IMPLEMENTING A PRIORITY QUEUE WITH A HEAP

- Removing an item with minimal key: remove_min()
- An item with the smallest key is at the root r of T
- Deleting the root directly?
- Need to make sure that the heap respects the complete binary tree property
  - The last position p (rightmost element at the deepest level) of T needs to be empty
- Given the above, what should we do?

IMPLEMENTING A PRIORITY QUEUE WITH A HEAP

- Removing an item with minimal key: remove_min()
- Remove root, put the element in the last position p to the root
- Next: maintain the heap order property
  - If p has no right child, let c be the left child of p
  - Otherwise, let c be a child of p with minimal key
- If $k_p <= k_c$, the heap order is satisfied
- If $k_p > k_c$, then need to restore the heap order
  - swap p and c
  - Perform algorithm recursively until the heap order property is restored
- Down heap bubbling
- Complexity?

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!