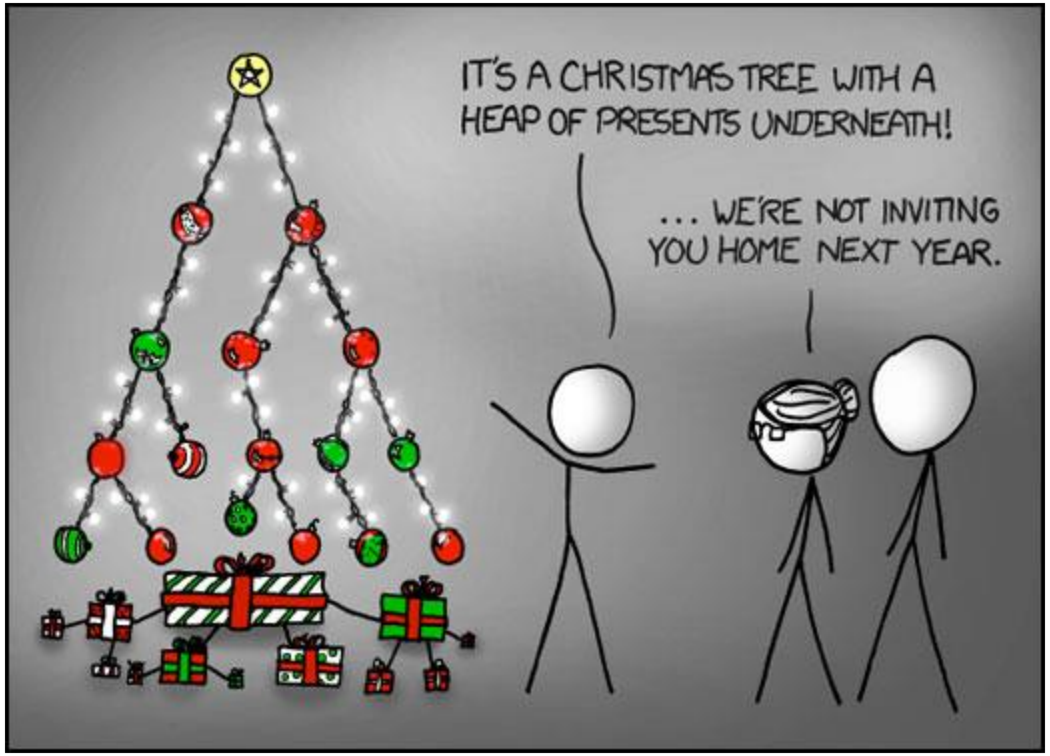
TREES

School of Artificial Intelligence

PREVIOUSLY ON DS&A

- Trees
- Terminologies
- Binary Trees

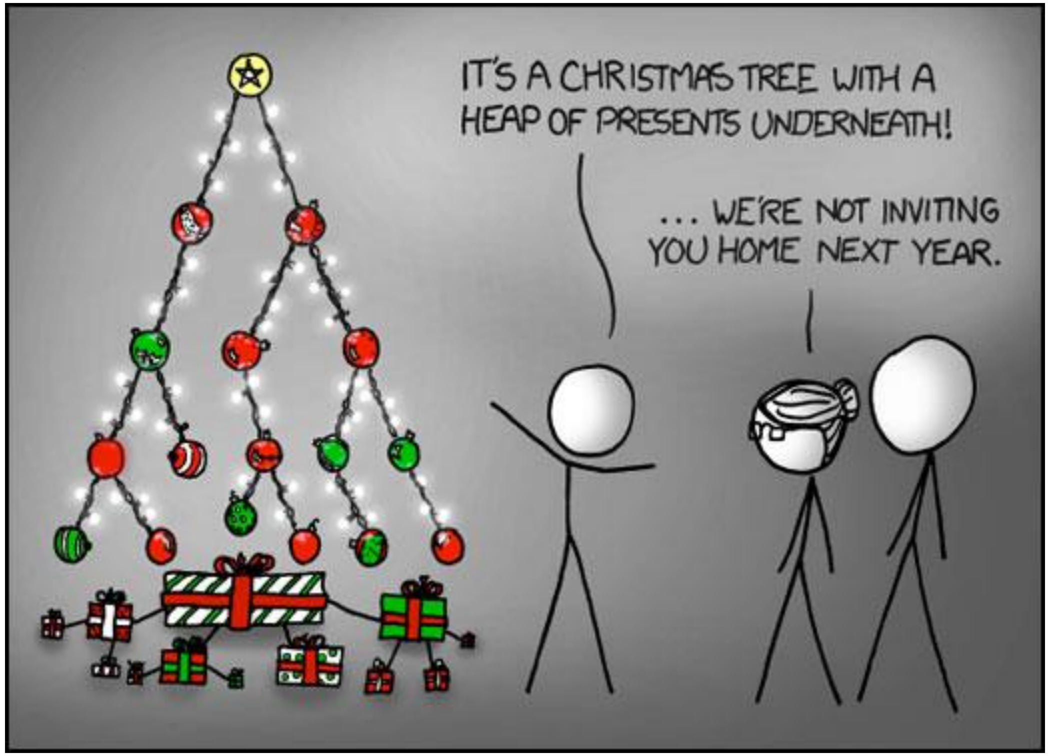


PREVIOUS LECTURE

- 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

THIS LECTURE

- Implementation of Trees
- Tree traversal algorithms

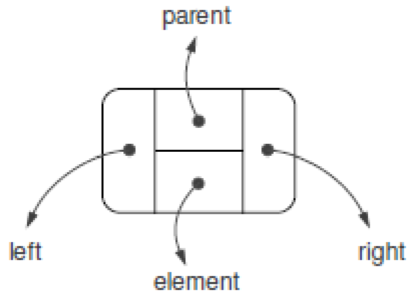


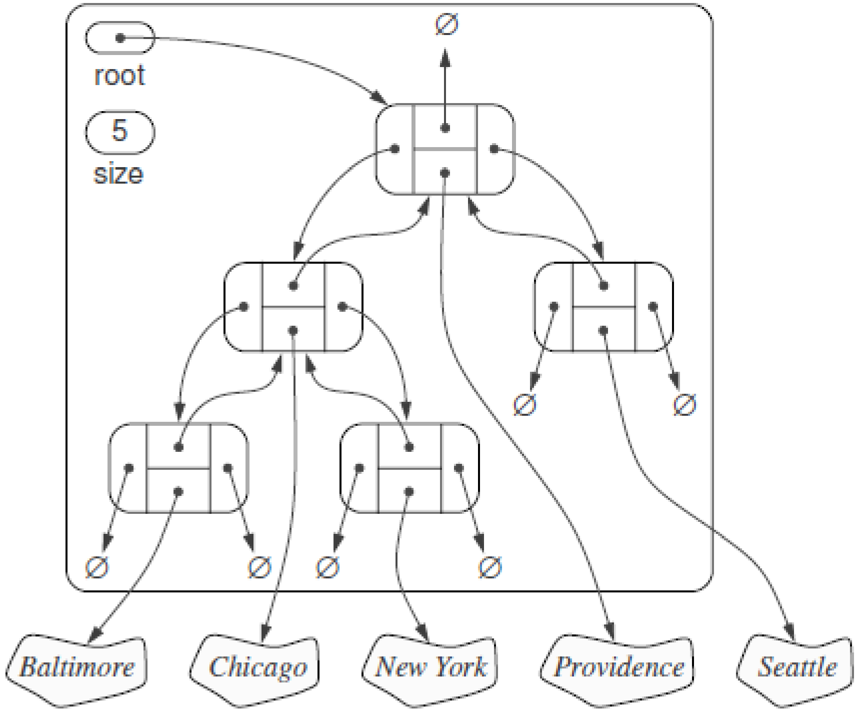
IMPLEMENTING TREES

- Trees we have looked so far: abstract classes
- Abstract class:
  - Key functions are empty: an error is raised when the functions are called
  - Cannot be directly instantiated: an instance of it would be illegal
- Choices for internal representation of trees
  - Intuitive: linked-based implementation
  - Less intuitive: array-based implementation
- We will consider: binary trees for now

LINKED STRUCTURE FOR BINARY TREES

- Intuitive: linked structure allows the use of references
- For each node at position p:
  - Element
  - Parent: None if p is Root
  - Left, right: None if p does not have a left/right





LINKED STRUCTURE FOR BINARY TREES

- LinkedBinaryTree that extends the binary tree ADT we previously defined
- Fundamental element
  - Node: defined as an internal, non-public class
  - Position: public class, wraps a Node inside
    - _validate function to check the if the position is legal
    - _make_position to wrap a node as a position

```
class LinkedBinaryTree(BinaryTree):
  """Linked representation of a binary tree structure."""

  class _Node:          # Lightweight, nonpublic class for storing a node.
    __slots__ = '_element', '_parent', '_left', '_right'
    def __init__(self, element, parent=None, left=None, right=None):
      self._element = element
      self._parent = parent
      self._left = left
      self._right = right
```

LINKED STRUCTURE FOR BINARY TREES

- Position
- Constructor: __init__()
- element()
- __eq__()

```
class Position(BinaryTree.Position):
  """An abstraction representing the location of a single element."""

  def __init__(self, container, node):
    """Constructor should not be invoked by user."""
    self._container = container
    self._node = node

  def element(self):
    """Return the element stored at this Position."""
    return self._node._element

  def __eq__(self, other):
    """Return True if other is a Position representing the same location."""
    return type(other) is type(self) and other._node is self._node
```

LINKED STRUCTURE FOR BINARY TREES

- Position
- _validate()
- _make_position()

```
def _validate(self, p):
  """Return associated node, if position is valid."""
  if not isinstance(p, self.Position):
    raise TypeError('p must be proper Position type')
  if p._container is not self:
    raise ValueError('p does not belong to this container')
  if p._node._parent is p._node:          # convention for deprecated nodes
    raise ValueError('p is no longer valid')
  return p._node

def _make_position(self, node):
  """Return Position instance for given node (or None if no node)."""
  return self.Position(self, node) if node is not None else None
```

LINKED STRUCTURE FOR BINARY TREES

- Operations for updating a Linked Binary Tree
- T.add_root(e): create a root for an empty tree, store e as the element, and return the position of the root
- T.add_left(p, e): create a node storing element e, link the node as the left chilf or position p, and return the position
- T.replace(p, e): replace the element at position p with element e, and return the previous value stored at p
- T.delete(p): remove the node at position p, replace it with its child, if any, and return the element stored previously at p
  - Error when p has two children
- T. attach(p, T1, T2): attach the internal structure of trees T1 and T2, as left and right subtrees of leaf position p; reset T1 and T2 to empty trees
  - Error if p is not a leaf

LINKED STRUCTURE FOR BINARY TREES

- Constructor
- Utility methods

```
def __init__(self):
  """Create an initially empty binary tree."""
  self._root = None
  self._size = 0

#------------------------- public accessors -------------------------
def __len__(self):
  """Return the total number of elements in the tree."""
  return self._size
```

LINKED STRUCTURE FOR BINARY TREES

- Accessors

```
def root(self):
  """Return the root Position of the tree (or None if tree is empty)."""
  return self._make_position(self._root)

def parent(self, p):
  """Return the Position of p's parent (or None if p is root)."""
  node = self._validate(p)
  return self._make_position(node._parent)

def left(self, p):
  """Return the Position of p's left child (or None if no left child)."""
  node = self._validate(p)
  return self._make_position(node._left)

def right(self, p):
  """Return the Position of p's right child (or None if no right child)."""
  node = self._validate(p)
  return self._make_position(node._right)
```

LINKED STRUCTURE FOR BINARY TREES

- Accessors

```
def num_children(self, p):
  """Return the number of children of Position p."""
  node = self._validate(p)
  count = 0
  if node._left is not None:          # left child exists
    count += 1
  if node._right is not None:         # right child exists
    count += 1
  return count
```

LINKED STRUCTURE FOR BINARY TREES

- Updators
  - _add_root()

```
def _add_root(self, e):
  """Place element e at the root of an empty tree and return new Position.

  Raise ValueError if tree nonempty.
  """
  if self._root is not None: raise ValueError('Root exists')
  self._size = 1
  self._root = self._Node(e)
  return self._make_position(self._root)
```

LINKED STRUCTURE FOR BINARY TREES

- Updators
  - _add_left()

```
def _add_left(self, p, e):
  """Create a new left child for Position p, storing element e.

  Return the Position of new node.
  Raise ValueError if Position p is invalid or p already has a left child.
  """
  node = self._validate(p)
  if node._left is not None: raise ValueError('Left child exists')
  self._size += 1
  node._left = self._Node(e, node)                    # node is its parent
  return self._make_position(node._left)
```

LINKED STRUCTURE FOR BINARY TREES

- Updators
  - _add_right()

```
def _add_right(self, p, e):
  """Create a new right child for Position p, storing element e.

  Return the Position of new node.
  Raise ValueError if Position p is invalid or p already has a right child.
  """
  node = self._validate(p)
  if node._right is not None: raise ValueError('Right child exists')
  self._size += 1
  node._right = self._Node(e, node)                    # node is its parent
  return self._make_position(node._right)
```

LINKED STRUCTURE FOR BINARY TREES

- Updators
  - _replace()

```
def _replace(self, p, e):
  """Replace the element at position p with e, and return old element."""
  node = self._validate(p)
  old = node._element
  node._element = e
  return old
```

- Updators
  - _delete()

```
def _delete(self, p):
  """Delete the node at Position p, and replace it with its child, if any.

  Return the element that had been stored at Position p.
  Raise ValueError if Position p is invalid or p has two children.
  """
  node = self._validate(p)
  if self.num_children(p) == 2: raise ValueError('p has two children')
  child = node._left if node._left else node._right        # might be None
  if child is not None:
    child._parent = node._parent      # child's grandparent becomes parent
  if node is self._root:
    self._root = child                # child becomes root
  else:
    parent = node._parent
    if node is parent._left:
      parent._left = child
    else:
      parent._right = child
  self._size -= 1
  node._parent = node                 # convention for deprecated node
  return node._element
```

LINKED STRUCTURE FOR BINARY TREES

- Updators
  - _attach()

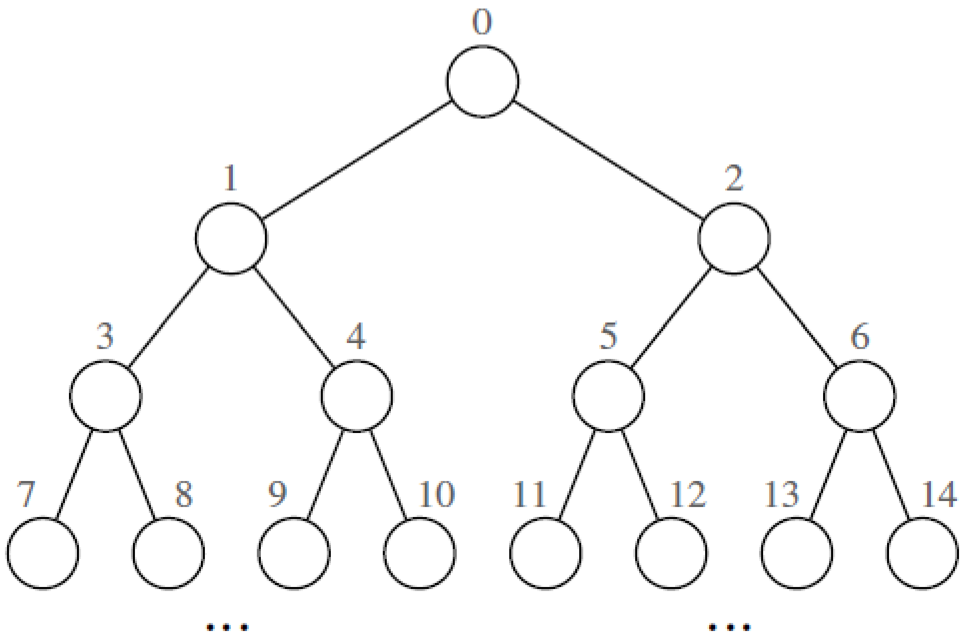
```
def _attach(self, p, t1, t2):
  """Attach trees t1 and t2 as left and right subtrees of external p."""
  node = self._validate(p)
  if not self.is_leaf(p): raise ValueError('position must be leaf')
  if not type(self) is type(t1) is type(t2):   # all 3 trees must be same type
    raise TypeError('Tree types must match')
  self._size += len(t1) + len(t2)
  if not t1.is_empty():                     # attached t1 as left subtree of node
    t1._root._parent = node
    node._left = t1._root
    t1._root = None                         # set t1 instance to empty
    t1._size = 0
  if not t2.is_empty():                     # attached t2 as right subtree of node
    t2._root._parent = node
    node._right = t2._root
    t2._root = None                         # set t2 instance to empty
    t2._size = 0
```

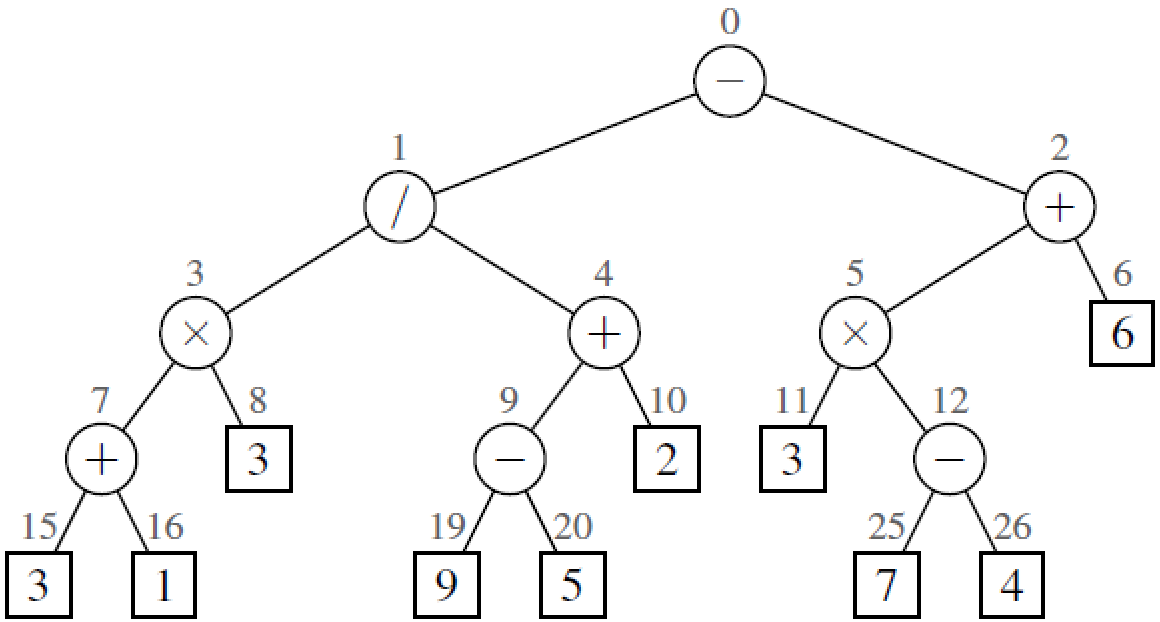
LINKED STRUCTURE FOR BINARY TREES

- Performance of Linked Binary Tree Implementation
- len(): O(1) because we keep track of the size of the tree
- root(), left(), right(), parent(), num_children: O(1)
- is_root(), is_leaf(): O(1)
- depth(): O(d +1), d depth of a node
- height(): O(n) as we previously discussed
- add_root(), add_left(), add_right(), replace(), delete(), attach(): O(1)

ARRAY BASED STRUCTURE FOR BINARY TREES

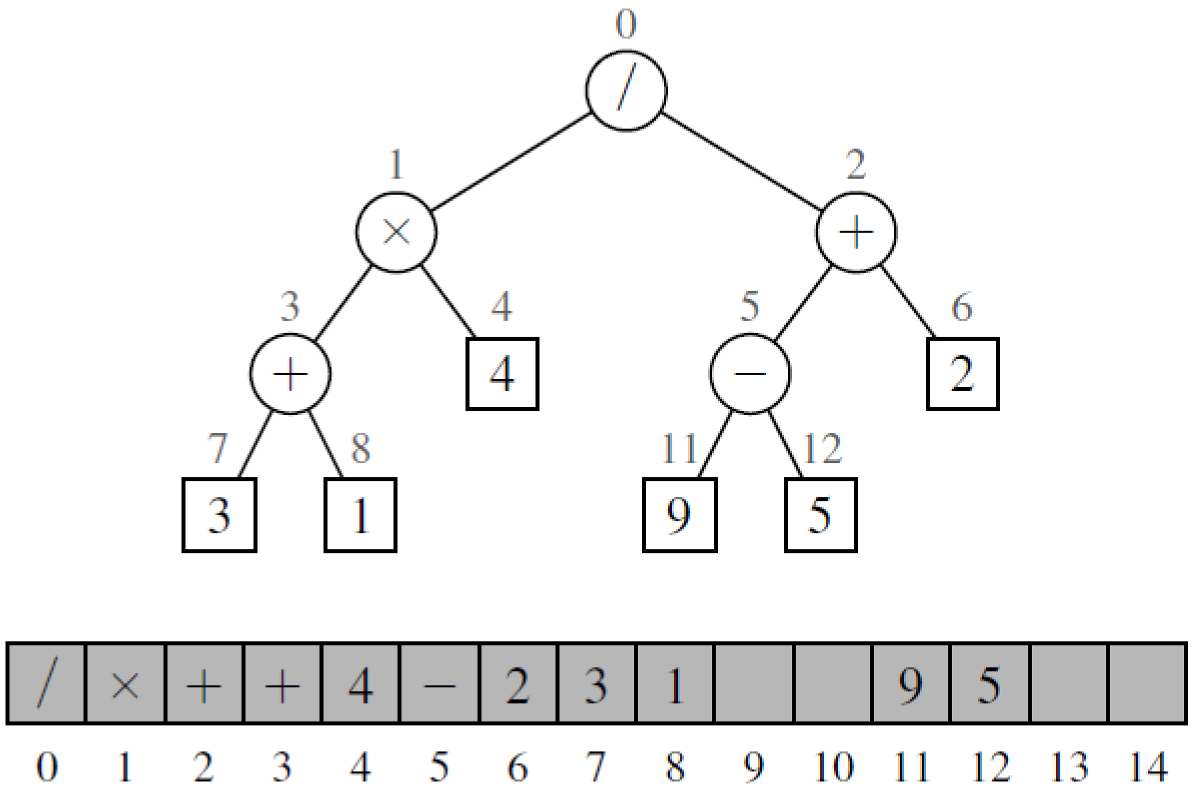
- For every position p of T, let f(p) be the integer defined as follows
  - If p is the root of T, then f(p) = 0
  - If p is the left child of position q, then f(p) = 2f(q) + 1
  - If p is the right child of position q, then f(p) = 2f(q) + 2





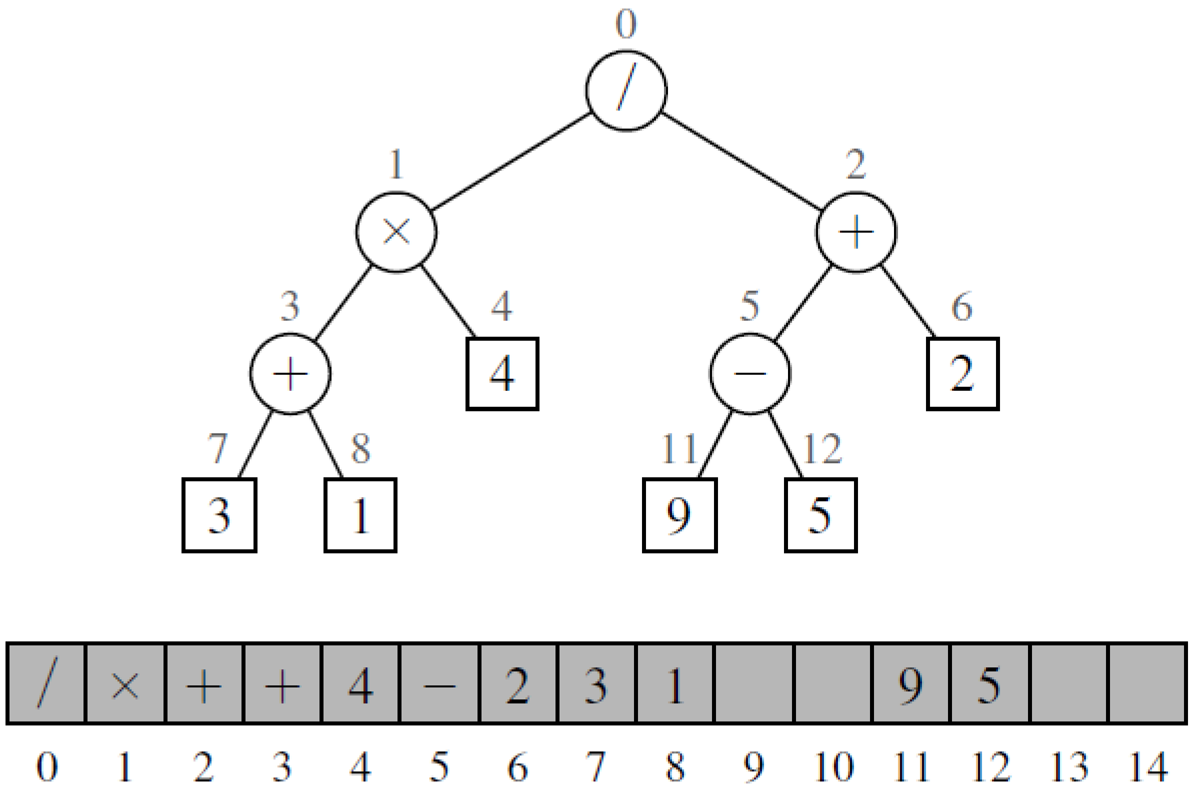
ARRAY BASED STRUCTURE FOR BINARY TREES

- Level numbering: numbers the positions on each level of T in increasing order from left to right (even if there are no children for some nodes)
- Array-based structure for a binary tree based on level numbering



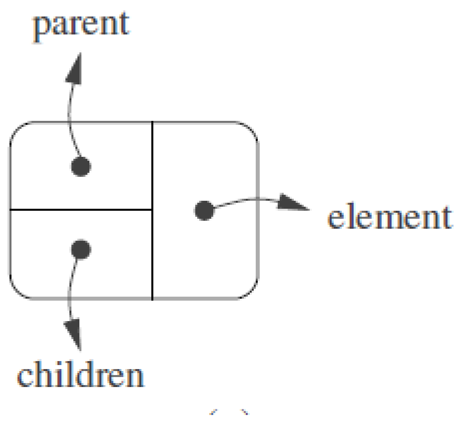
ARRAY BASED STRUCTURE FOR BINARY TREES

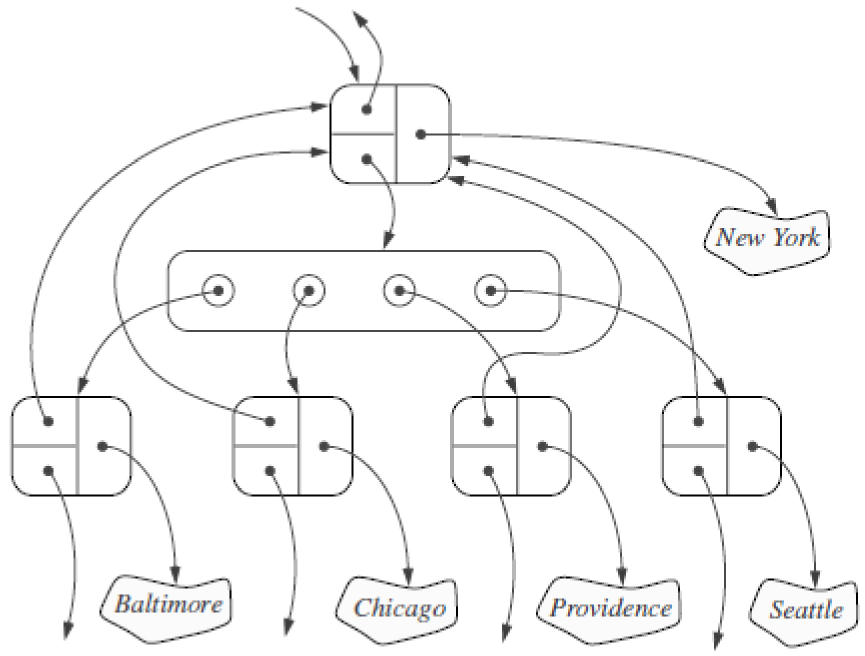
- Positions are represented by single integers
- Methods such as root(), parent(), left() and right() can be performed using arithmetic operations on the number f(p)
  - Left of p: 2f(p) + 1
- Space usage: depends on the shape of the tree
  - n = number of nodes of T
  - f_M = maximum value of f(p) over all nodes of T
  - Array length $N = 1 + f_M$
  - Worst case: $N = 2^n - 1$



LINKED STRUCTURE FOR GENERAL TREES

- Similar to Binary Tree, except there is no notion of left and right
- Single container of reference to its children





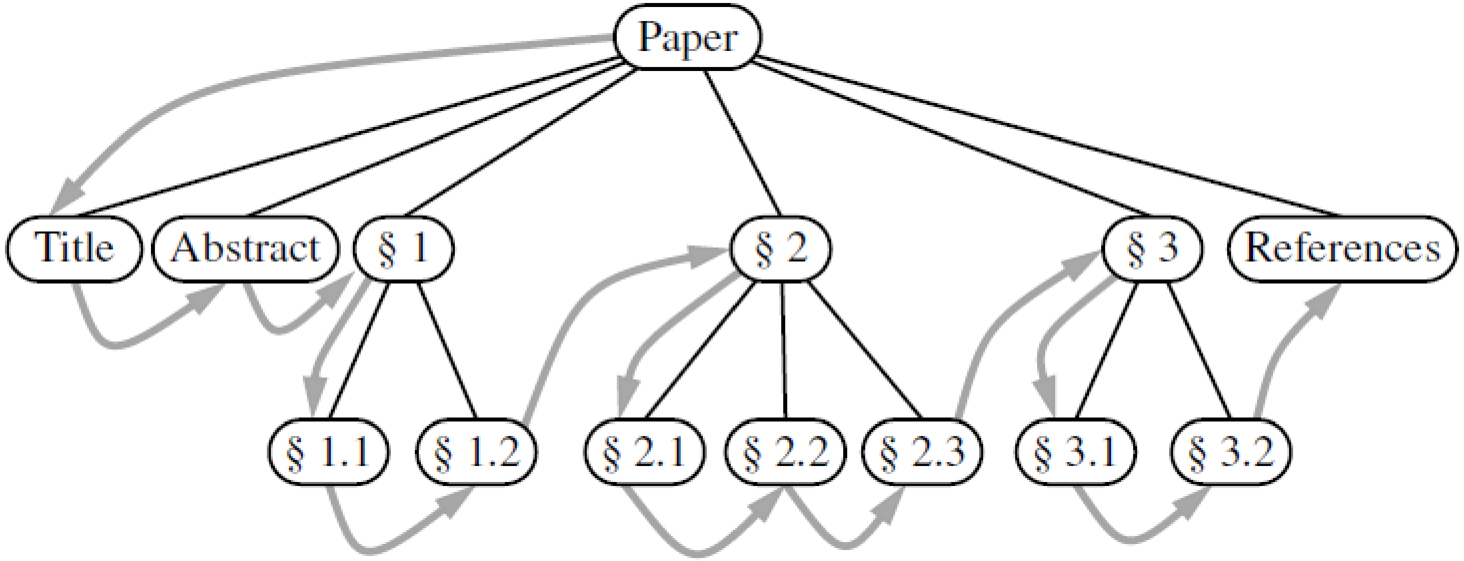
LINKED STRUCTURE FOR GENERAL TREES

- Performance
- len(): O(1)
- root(), parent(), is_root(), is_leaf(): O(1)
- children(p): $O(c_p + 1)$, c_p number of children of a position p
- depth(): $O(d_p + 1)$
- height(): O(n)

Operation	Running Time
len, is_empty	$O(1)$
root, parent, is_root, is_leaf	$O(1)$
children(p)	$O(c_p + 1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$

TREE TRAVERSAL ALGORITHMS

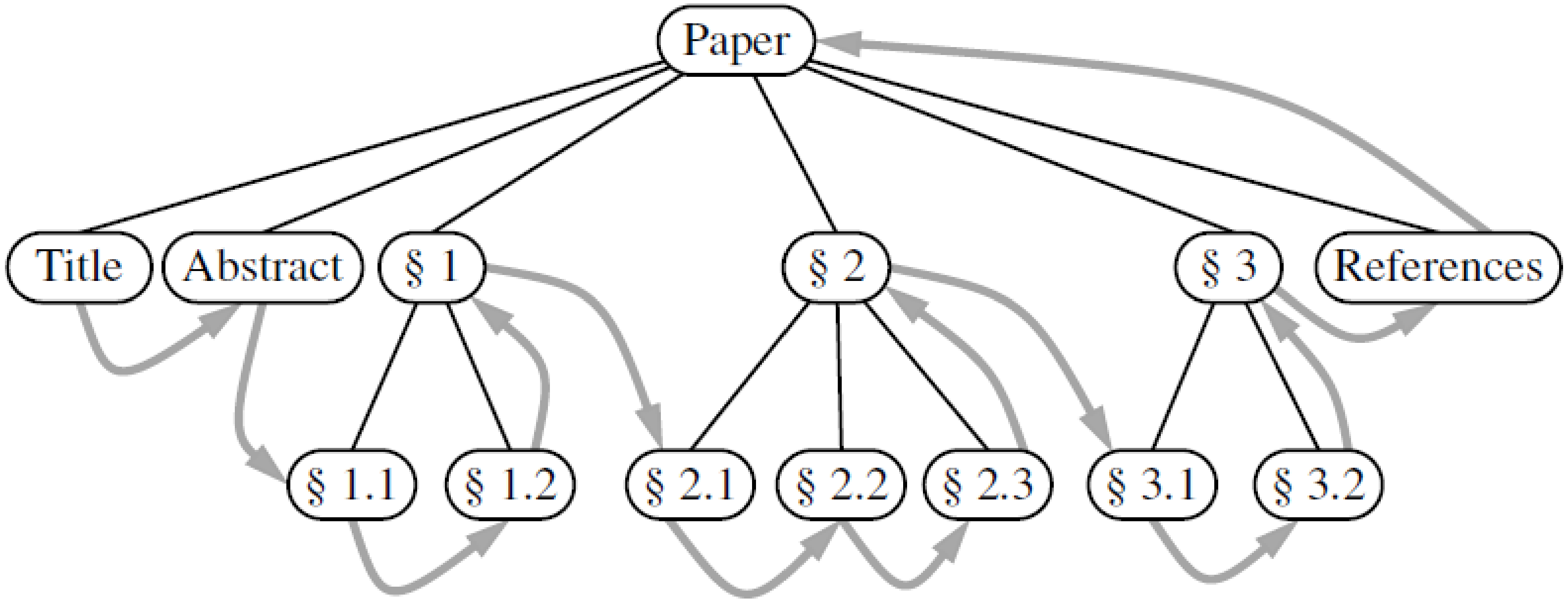
- Traversal: a systematic way of accessing or 'visiting' all the positions of T
- Preorder traversal:
  - visit root of T first and then visit the sub-trees recursively
  - If tree is ordered, then the subtrees are traversed according to the order of the children



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 ALGORITHMS

- Postorder traversal:
  - 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 ALGORITHMS

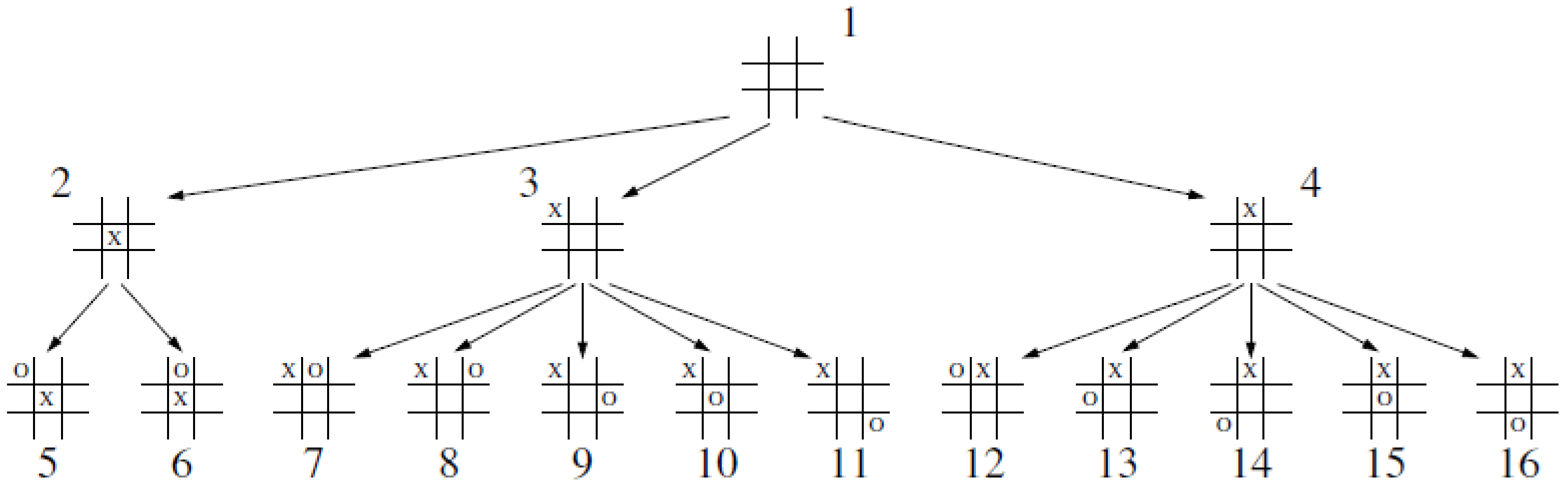
- Running time
- At each position p, non-recursive part: $O(c_p + 1)$
  - c_p number of children of p
- Overall: $O(n)$

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 ALGORITHMS

- Breadth-First 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

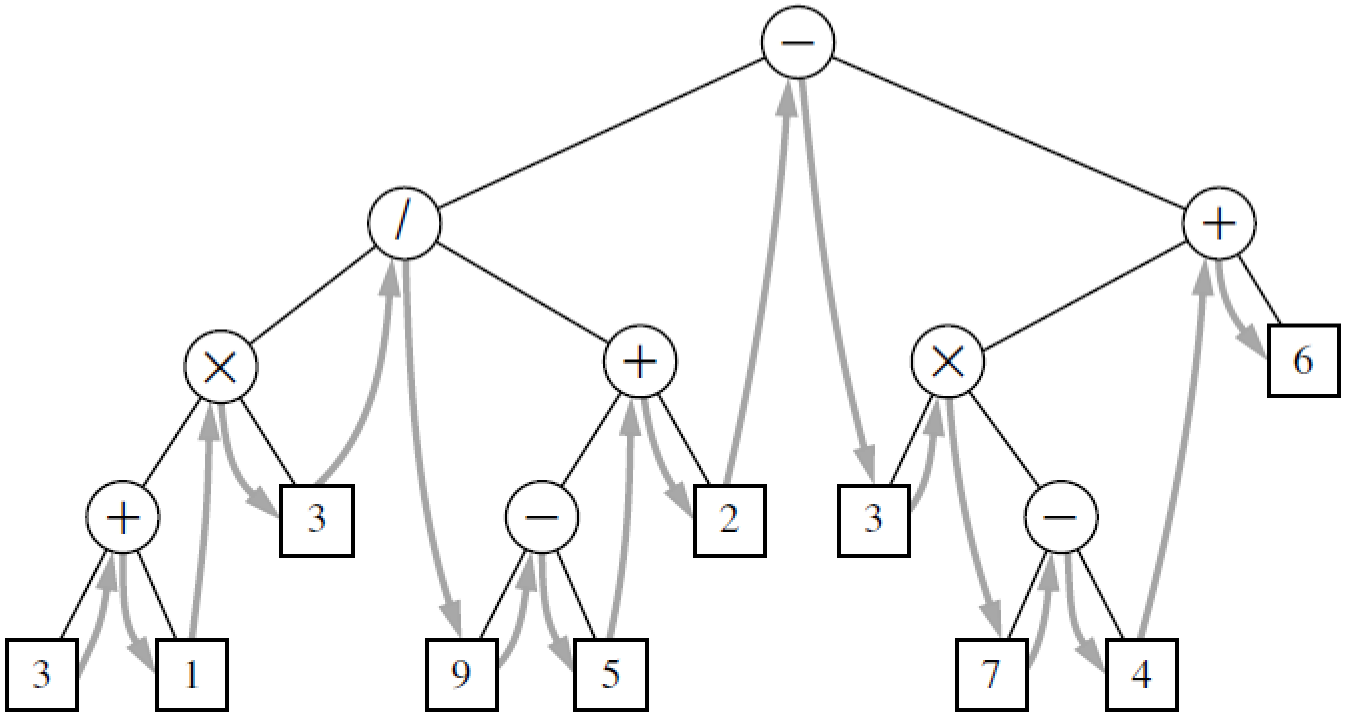
- Breadth-First Tree Traversal
  - Visit all the position at depth d before visit the position at depth d+1
- O(n) running time: n calls to enqueue() and n calls to dequeue()

```
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

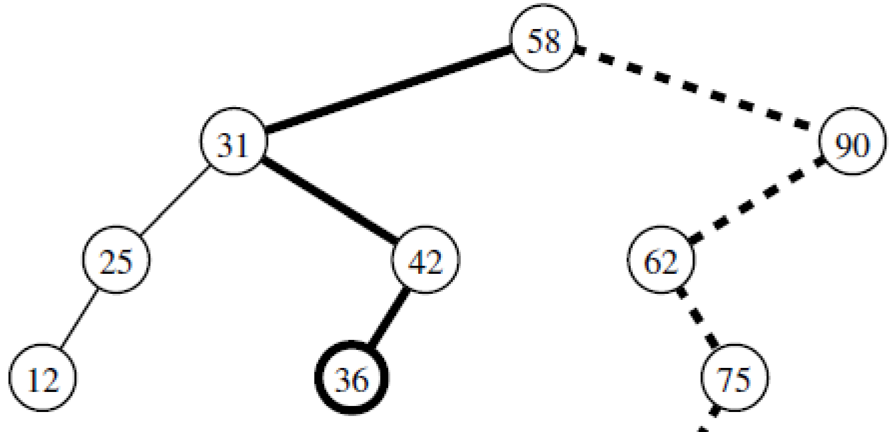
- In-order Traversal of a Binary Tree
  - Visit left
  - Visit node
  - Visit right

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, dentoed 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?



SMALL QUIZ FOR THIS WEEK:

- A stream of 1s and 0s are coming.
- At any time, we have to tell that the binary number from the 1s and 0s is divisible by 3 (or not)
- For example:
  - 1 – not divisible
  - 11 – divisible
  - 110 – divisible
  - 1100 – divisible
- Try to write an algorithm that check any random binary number

THANKS

See you in the next session!