08

Trees

Chapter 8

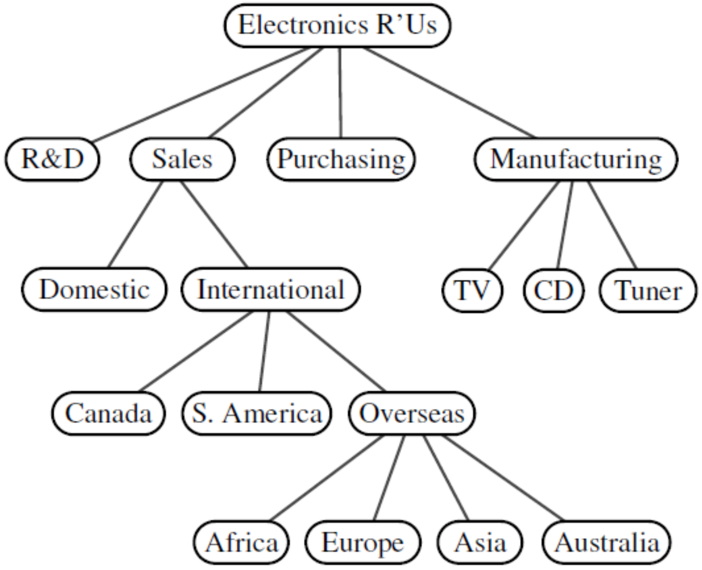
ODTÜ
METU

Trees

Stacks, queues, arrays, sequences, linked lists are **linear data structures**.

Linear data structures have the notions of *next* and *previous*.

Trees are **non-linear data structures** in which data are stored in a *hierarchical* relationship.

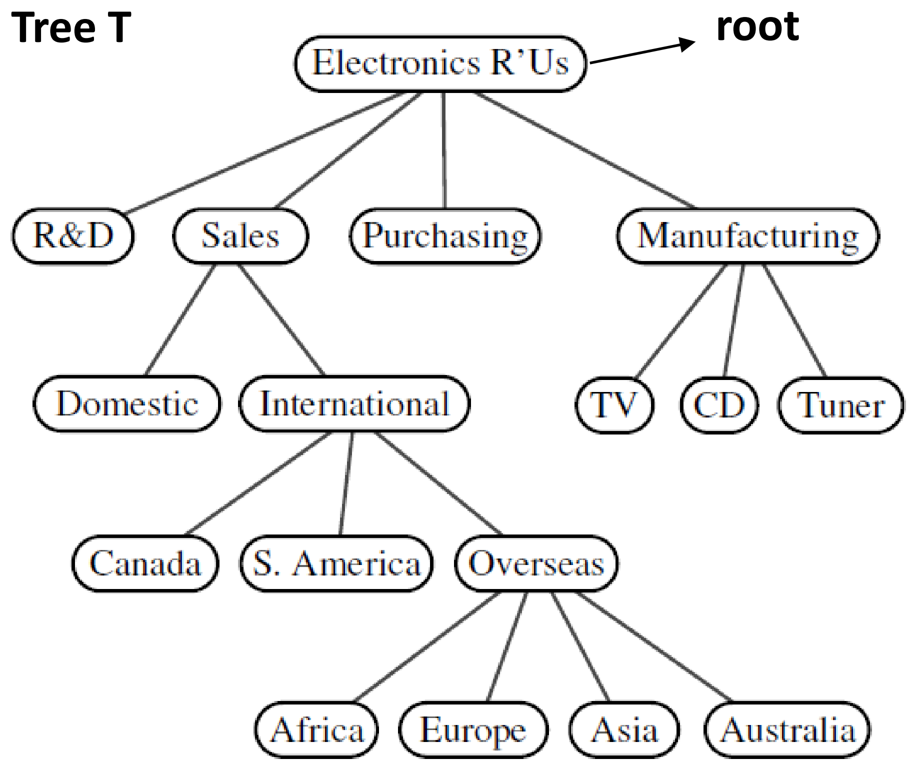


Trees

Tree

Tree T is a set of **nodes** storing elements such that nodes have a **parent-child** relationship.

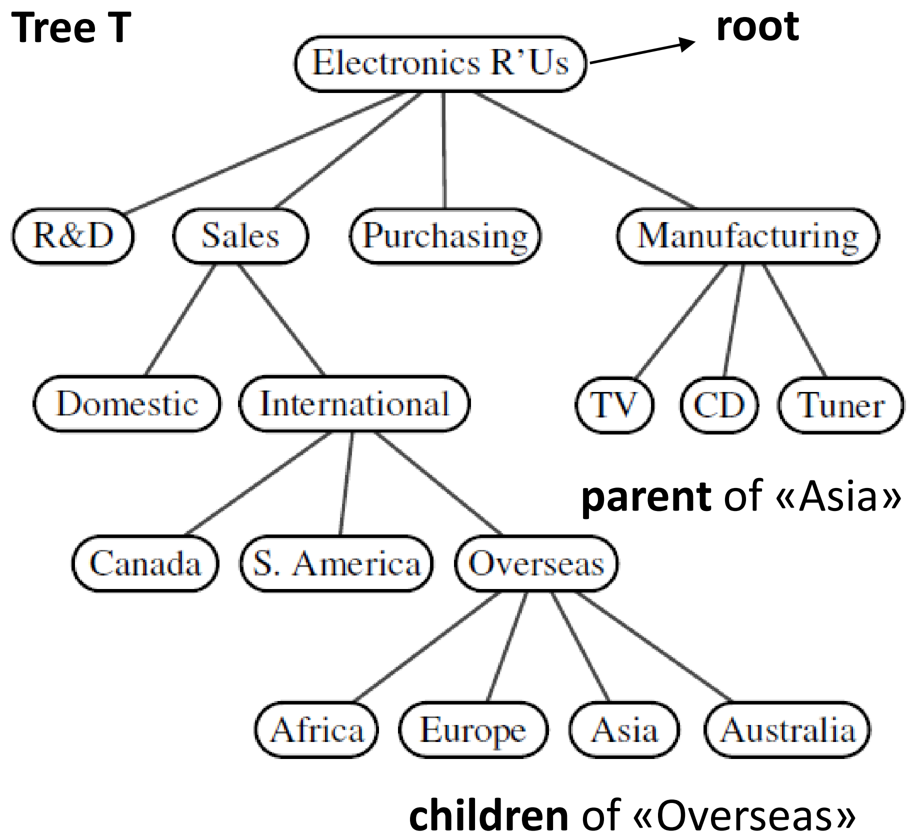
If tree is not empty, it has a node without a parent, which is called **root**.



Trees

Tree

Except for the root, each node v has a single **parent** node w. Every node with parent w is called **child** of w.



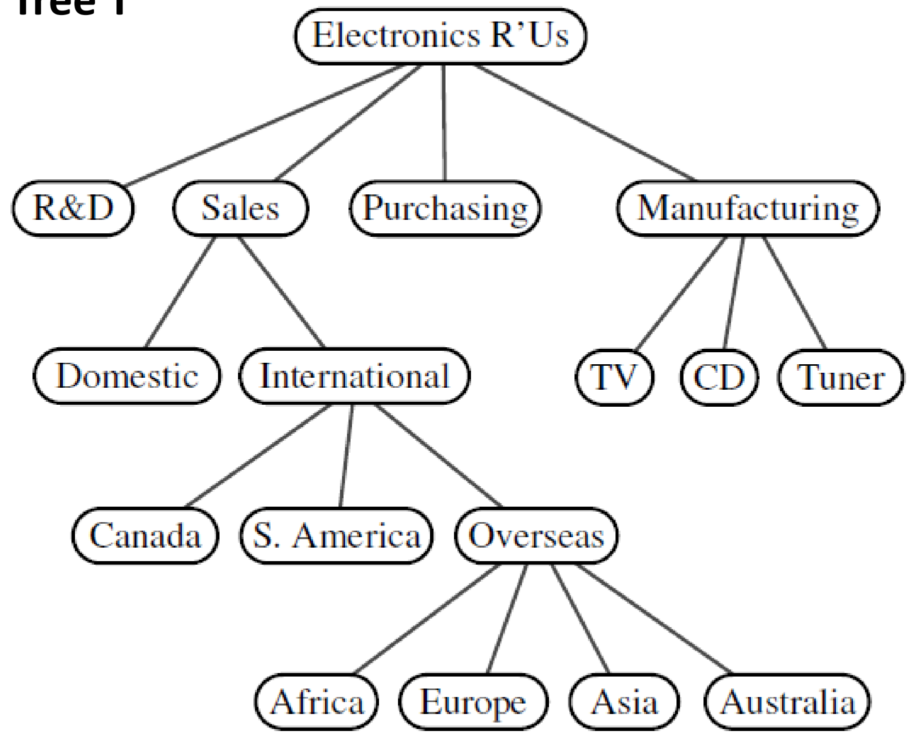
Trees

Tree

Recursive Definition:

- A tree can be empty i.e., no nodes
- A single node by itself is a tree
- Given a node **n** and trees T_1, …, T_m whose roots are n_1, …, n_m, respectively, a new tree can be constructed by making n the parent of n_1, …, n_m.

Tree T

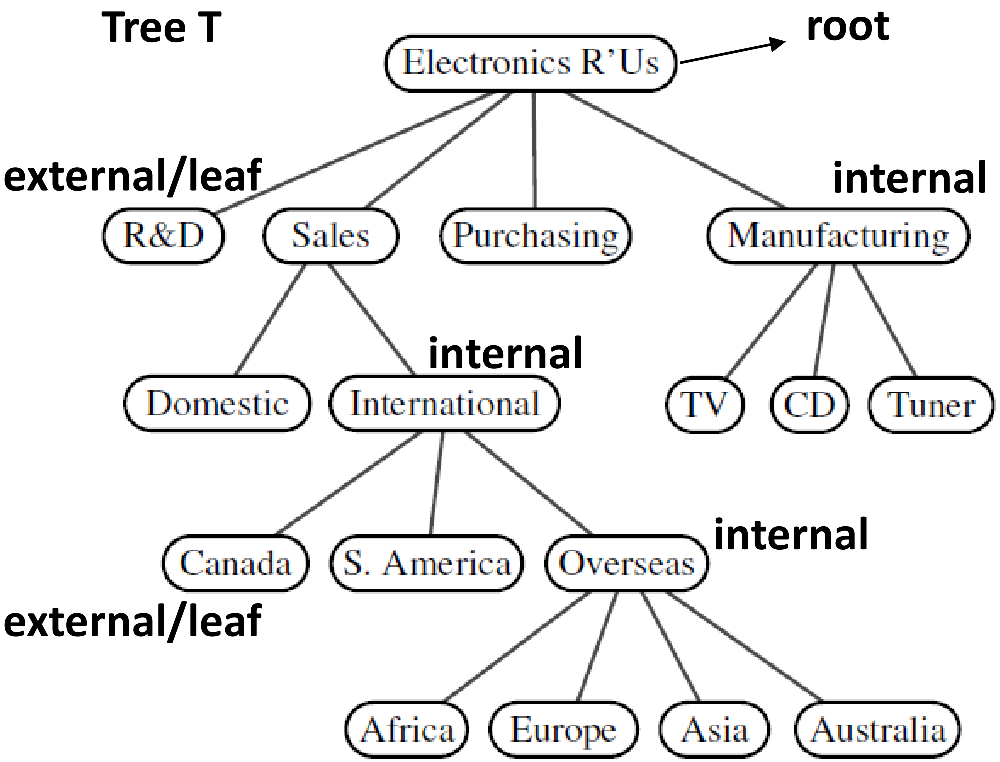


Trees

Tree

Nodes sharing the same parent are called **siblings**.

If a node has one or more children, it is said to be **internal**, otherwise it is an **external/leaf** node.



«Africa» and «Europe» are **siblings**

Trees

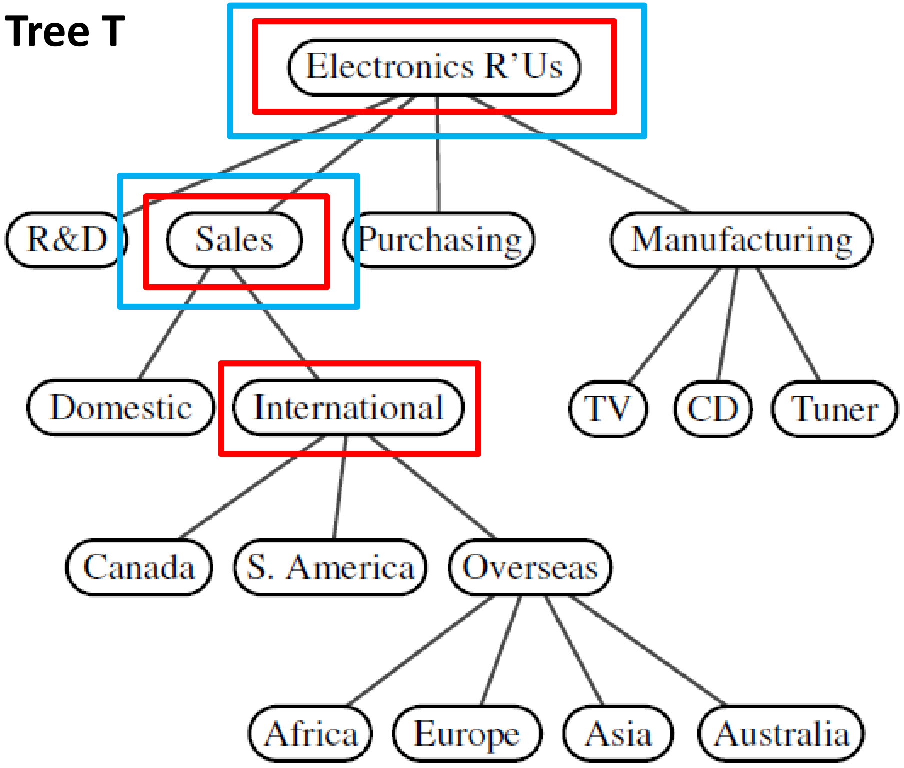
Tree

Node u is **ancestor** of node v if

- u = v, or
- u is an ancestor of parent of v.

Node u is a **proper ancestor** of node v if u is an ancestor of v and u ≠ v.

Tree T



Ancestors of «International»

Proper Ancestors of «International»

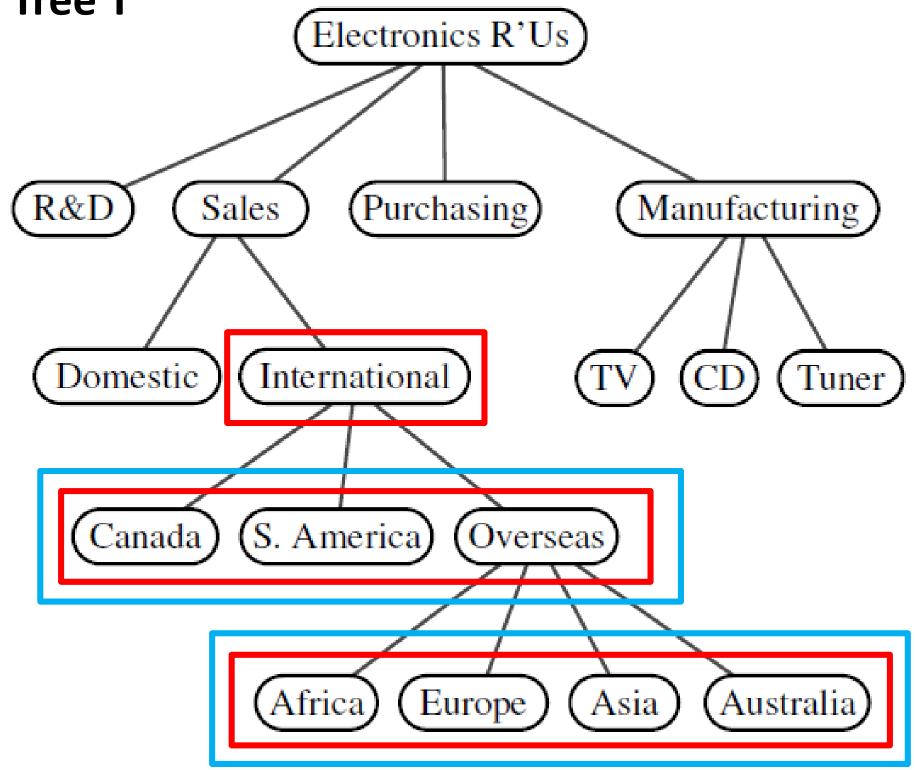
Trees

Tree

Node u is **descendant** of node v if v is an ancestor of v.

Node u is a **proper descendant** of node v if v is an ancestor of u and u ≠ v.

Tree T



Descendants of «International»

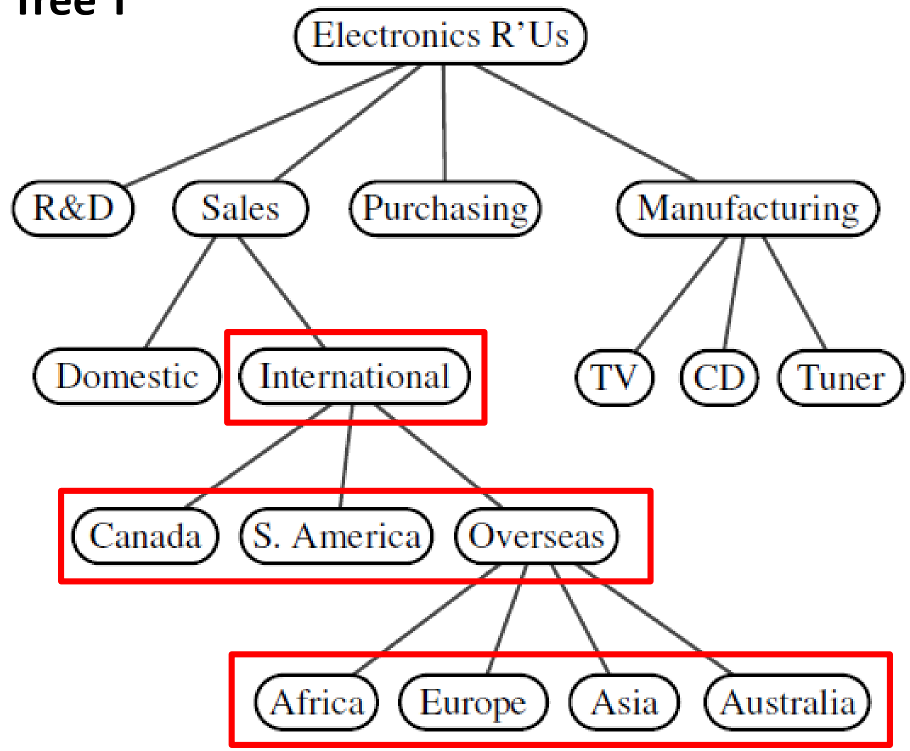
Proper Descendants of «International»

Trees

Tree

A **subtree** T' of T rooted at node v is the tree formed with the descendants of v.

Tree T



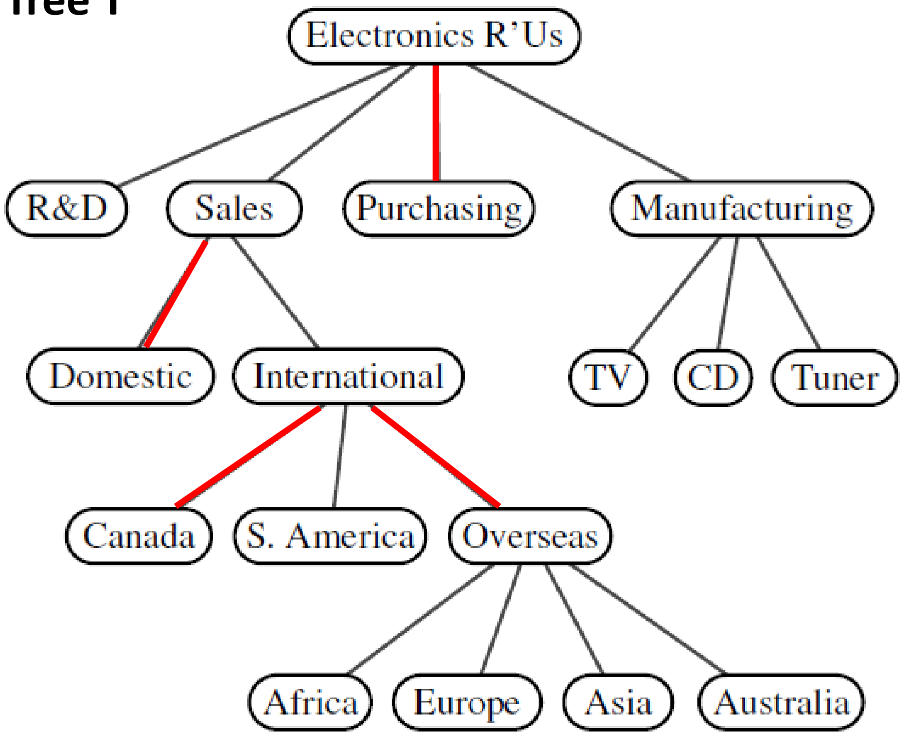
subtree rooted at «International»

Trees

An **edge** of a tree T is a tuple (u,v) such that u and v are nodes and u is parent of v or v is a parent of u.

Hence, edge concept is bidirectional.

Tree T

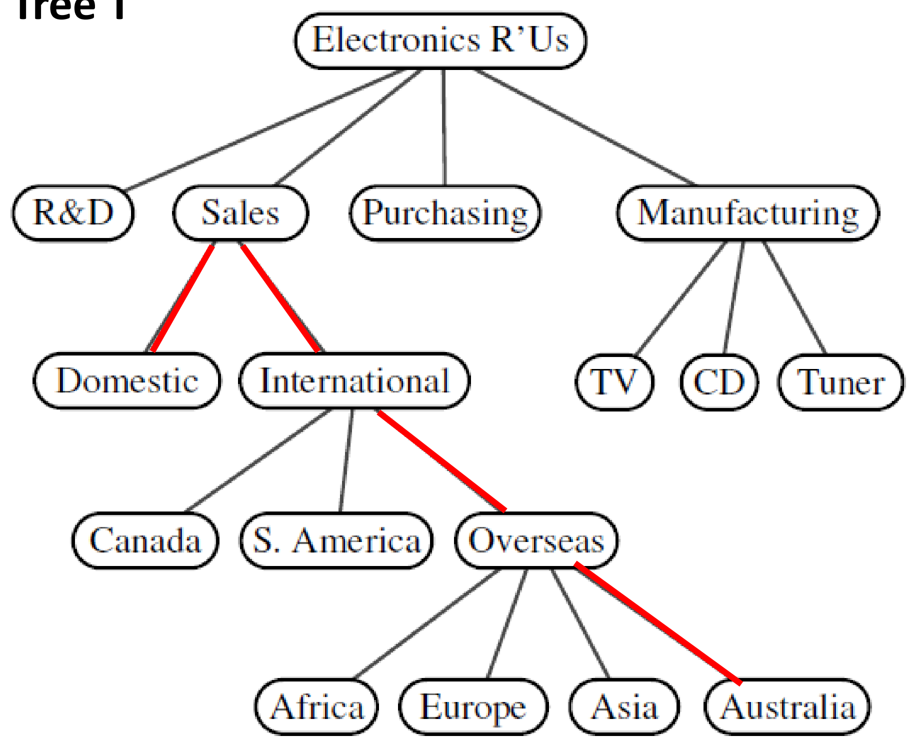


some of the **edges**

Trees

An **path** in a tree T is a <u>sequence of nodes</u> where, for any given two consecutive nodes u and v, (u,v) form an edge.

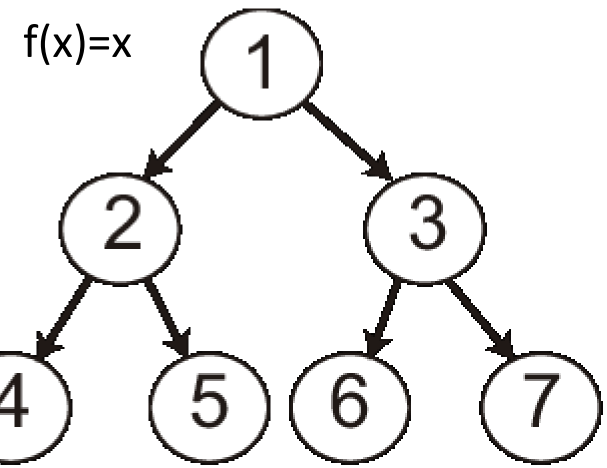
Tree T

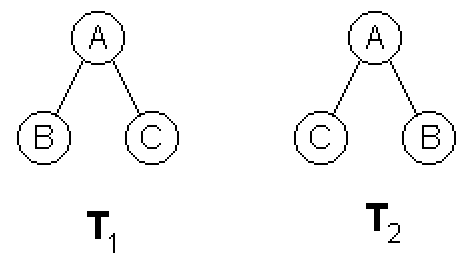


one of the possible **paths**

Trees

A tree is said to be **ordered** if siblings of any node is ordered according to some function f(x), where x is a child node.





If T_1 and T_2 are ordered trees then $T_1 \neq T_2$ else $T_1 = T_2$.

Shamelessly borrowed from: https://cs.lmu.edu/~ray/notes/orderedtrees/

Trees

The **depth** of node u is the number of proper ancestors of n.

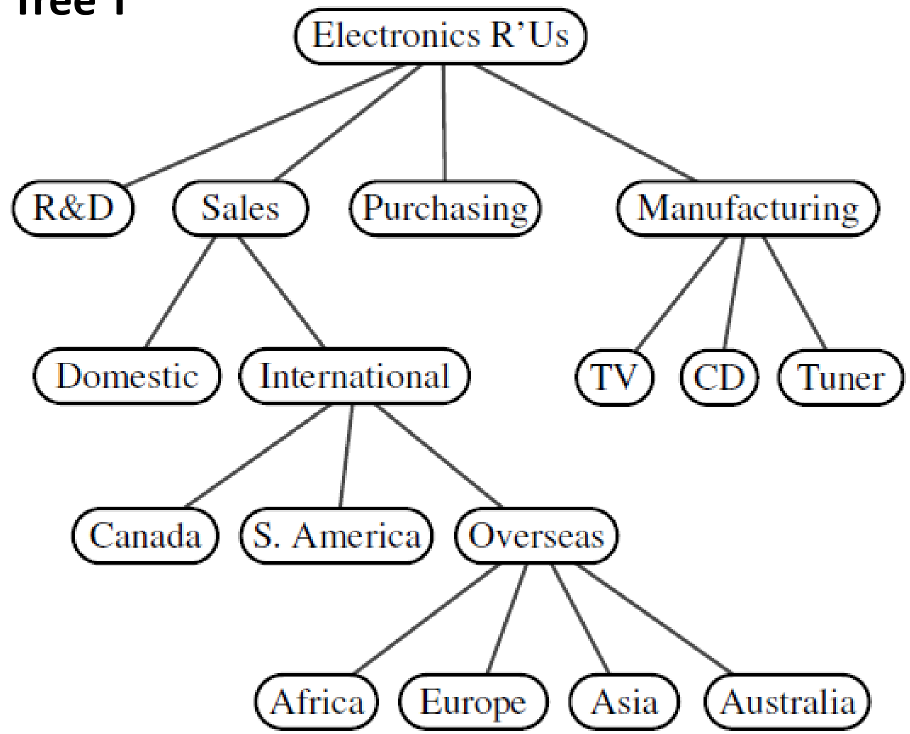
Recursive Definition:

- root's depth is 0.
- The depth of u is one plus the depth of u's parent.

Run Time for position p is $O(d_p+1)$, where d_p is depth p.

Worst-case O(n). Why?

Tree T



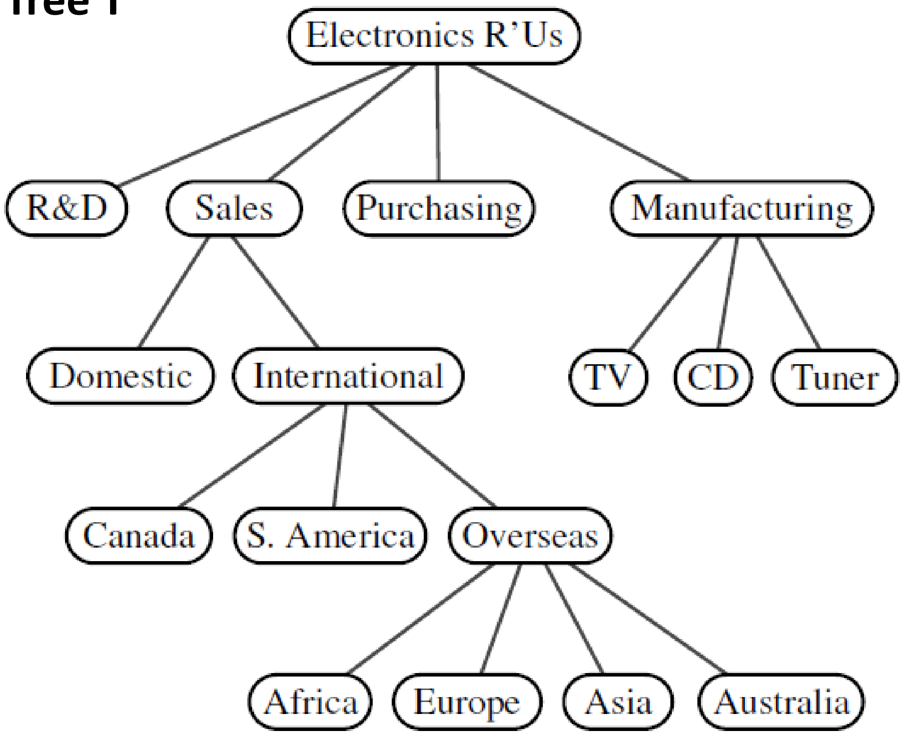
depth of «Overseas» is 3
depth of root is 0

Trees

The **height** of tree T is the largest depth of any node in T.

If we were to check every node (n), that would cost us n * (O(d_p+1)). In the worst-case scenario, O(d_p+1) is O(n). So checking every single node would be O(n^2).

Tree T



height of T is 4

Trees

Height

Checking every single node would be $O(n^2)$.

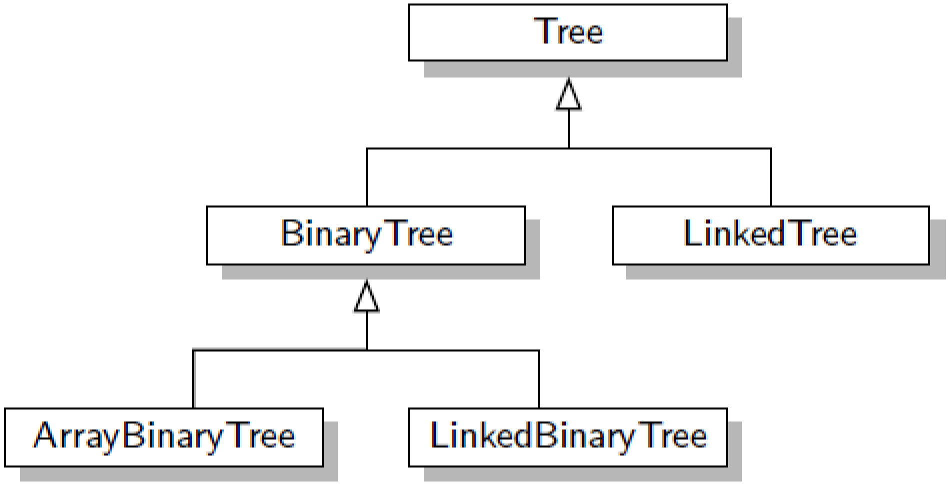
```
def _height2(self, p):                    # time is linear in size of subtree
  """Return the height of the subtree rooted at Position p."""
  if self.is_leaf(p):
    return 0
  else:
    return 1 + max(self._height2(c) for c in self.children(p))
```

For each position p, number of op's: $O(c_p+1)$ (Why?)

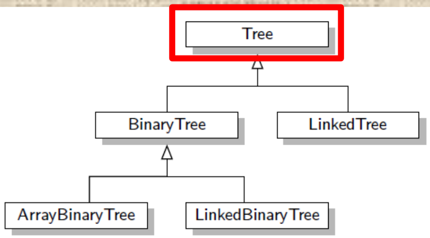
In total, for all of the p's: $O(\Sigma_p(c_p+1)) = O(n + \Sigma_p c_p)$

$\Sigma_p c_p = n-1$ (Why?)

Trees - Implementation



Tree Abstract Data Type



Here is the list of all operations that has to be supported by all types of trees.

p denotes the position (node) of a tree.

p.element()

T.root()

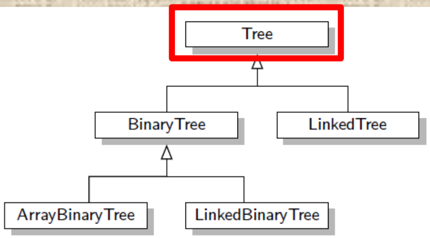
T.is_root(p)

T.parent(p)

T.num_children(p)

T.children(p)

Tree Abstract Data Type



Here is the list of all operations that has to be supported by all types of trees (continued).

p denotes the position (node) of a tree.

T.is_leaf(p)

len(T)

T.is_empty()

T.positions()

iter(T)

T.depth(p)

T.height() and T.height(p)

Tree - Implementation

```
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__
```

Tree - Implementation

```
# ---------- 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')
```

Tree - Implementation

```
# --------- 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
```

Tree - Implementation

```
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))

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))
```

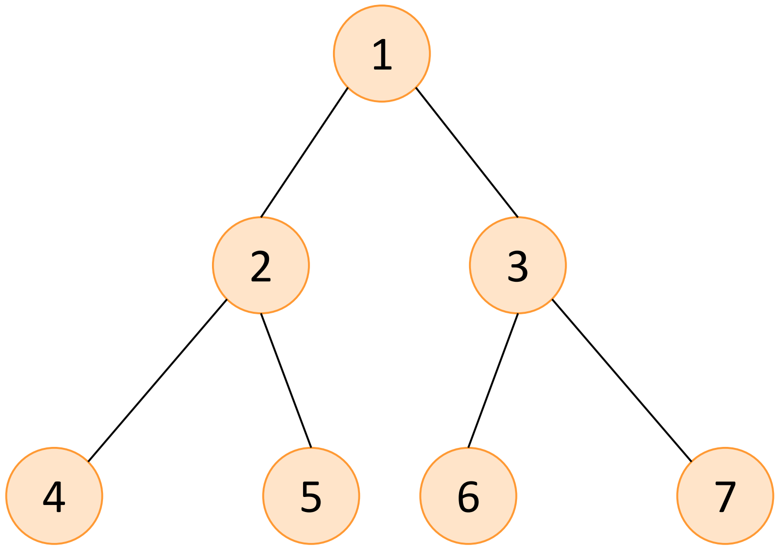
Tree - Implementation

```
def height(self, p=None):
  """Return the height of the subtree rooted at Position p.

  If p is None, return the height of the entire tree.
  """
  if p is None:
    p = self.root()
  return self._height2(p)           # start _height2 recursion
```

Binary Trees

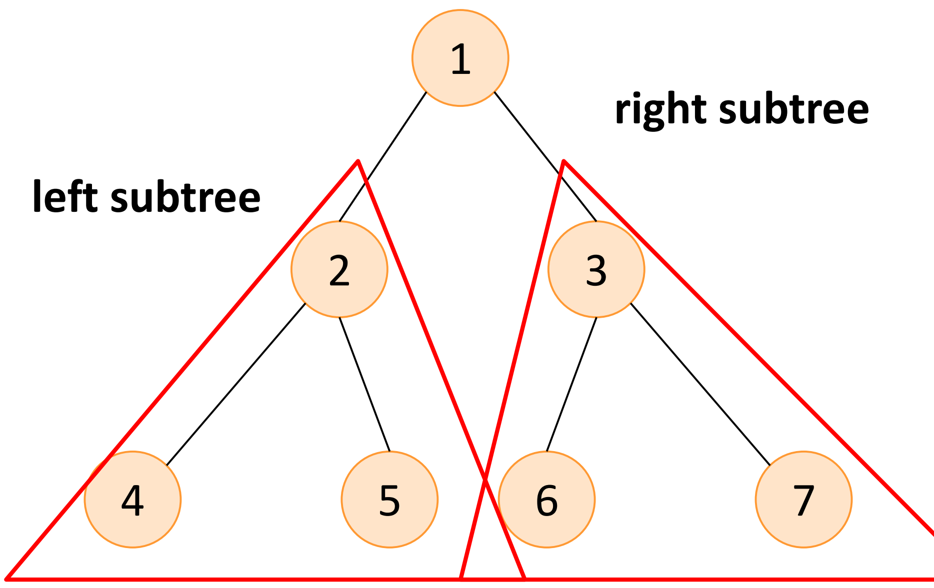
Binary tree is an <u>ordered</u> tree s.t.:

- Each node has <u>at most</u> two children
- Each node is labeled as left/right child
- Left child precedes the right child in the children order.



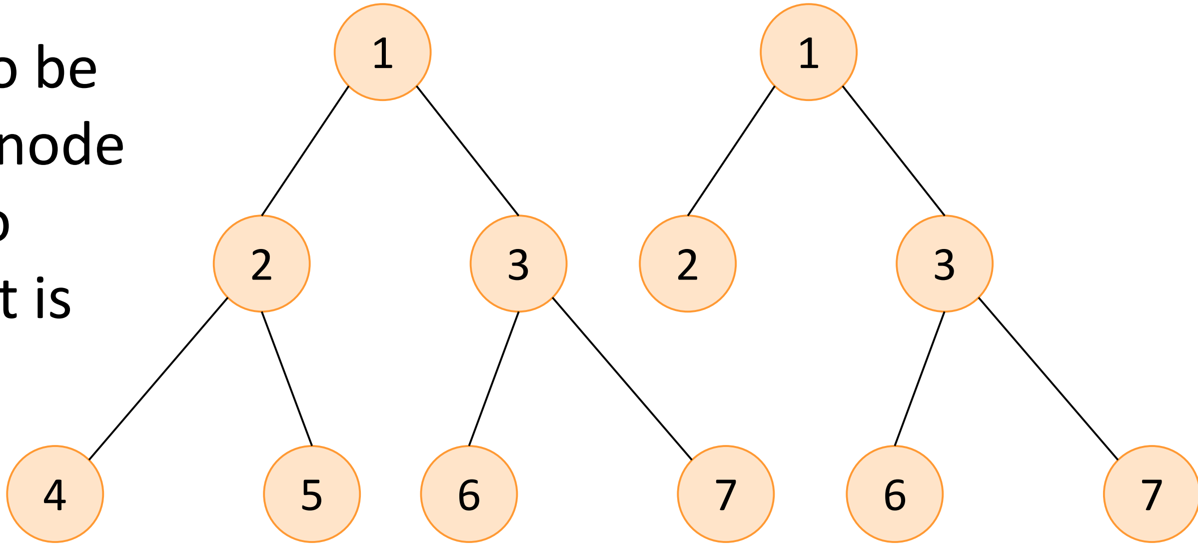
Binary Trees

Subtree rooted at an left or right child of an internal node is called **left or right subtree** of that node.

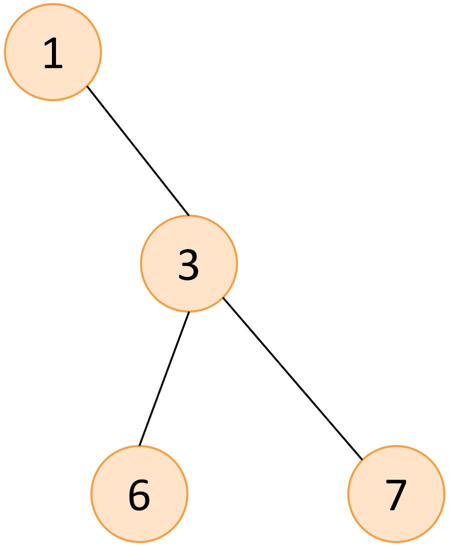


Binary Trees

A binary tree is said to be **proper** or **full** if each node has either zero or two children. Otherwise, it is called **improper**.



proper binary trees

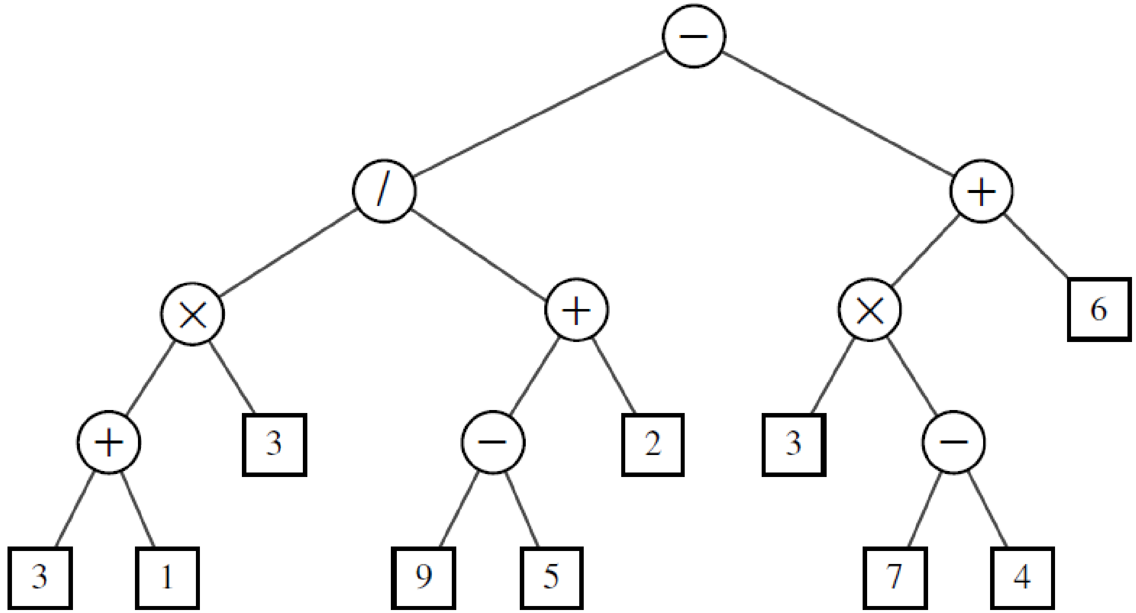


improper binary tree

Binary Trees

Representing arithmetic operations with binary trees.

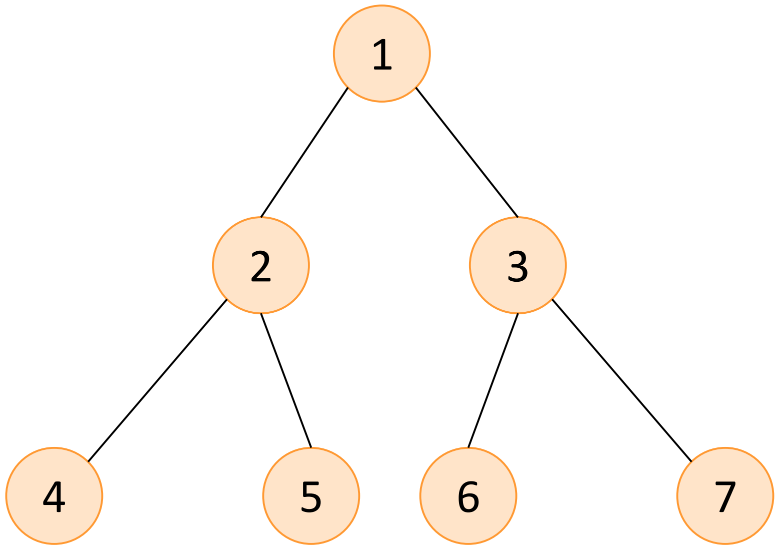
- Leaf nodes hold value
- Internal nodes hold arithmetic operators



Binary Trees

Recursive Definition of Binary Tree T:

- T is empty, or
- T consists of
  - Root node storing an element
  - A binary tree as the left subtree of T
  - A binary tree as the right subtree of T



Properties of Binary Trees

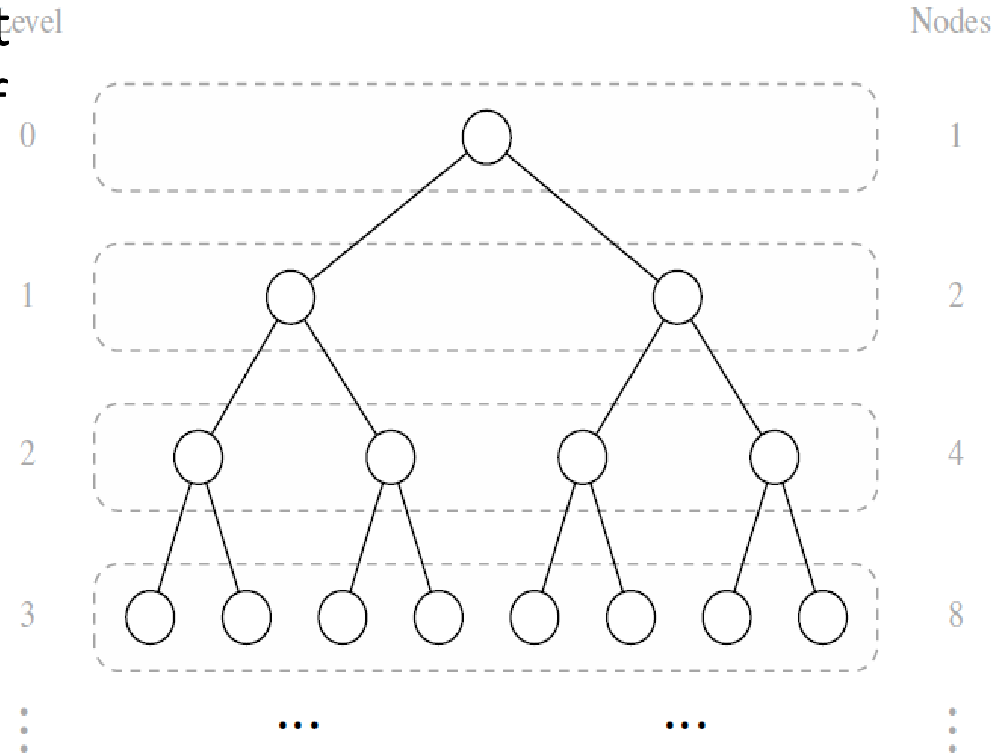
The set of all nodes of T at depth d is called **level d** of T.

Level 0 can have at most one node.

Level 1 → 2^1 (Cum: 2^{1+1}-1)

Level 2 → 2^2 (Cum: 2^{2+1}-1)

Level d → 2^d (Cum: 2^{d+1}-1)

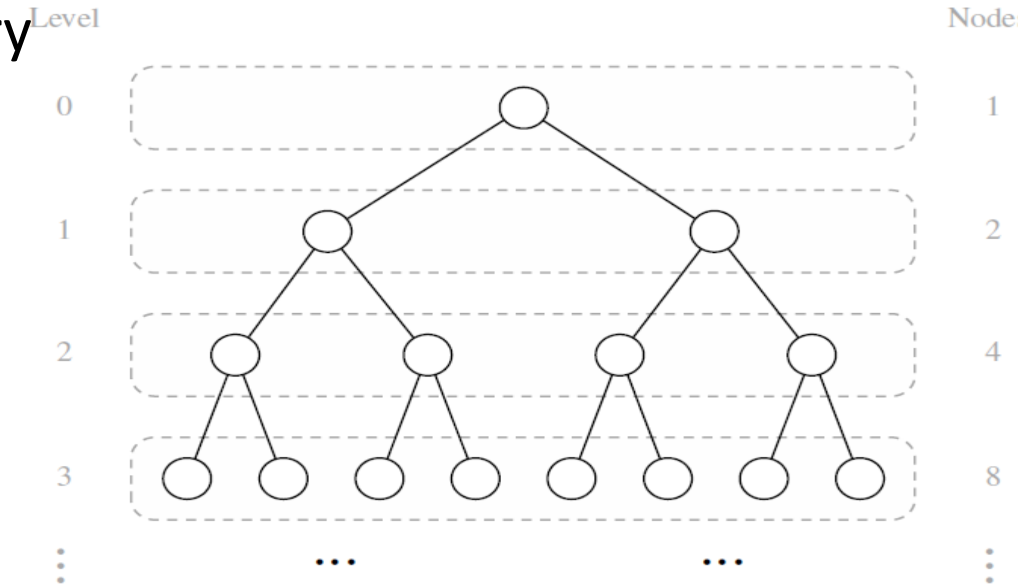


Properties of Binary Trees

Given height h of a binary tree T,

Minimum number of nodes is h+1. (Why?)

Maximum number of nodes is 2^{h+1}-1. (Why?)



So,

$h+1 <= n <= 2^{h+1}-1$

Tree of height t

$2^0 + 2^1 + \cdots + 2^h = 2^{h+1} - 1$

(max # of nodes)

$\underset{h=0}{1} + \underset{h=1}{1} + \underset{h=2}{1} + \cdots + \underset{h=h}{1} = h + 1$

(min # of nodes)

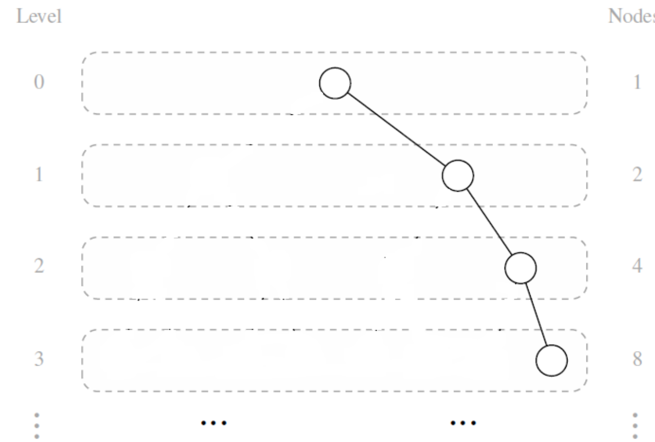
Properties of Binary Trees

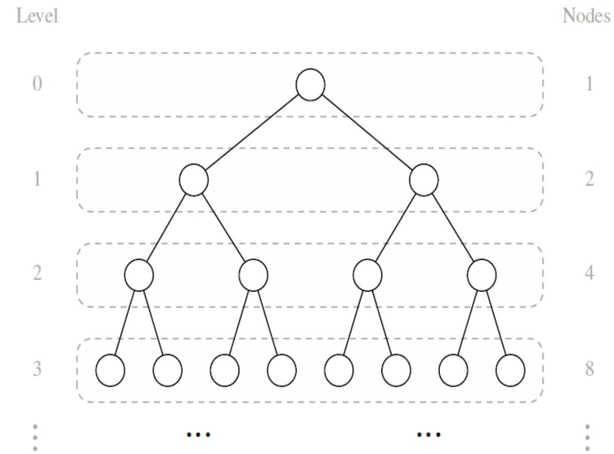
Given height h of a binary tree T,

Minimum number of external nodes is 1.
Maximum number of external nodes is 2^h

So,

$1 <= n_e <= 2^h$





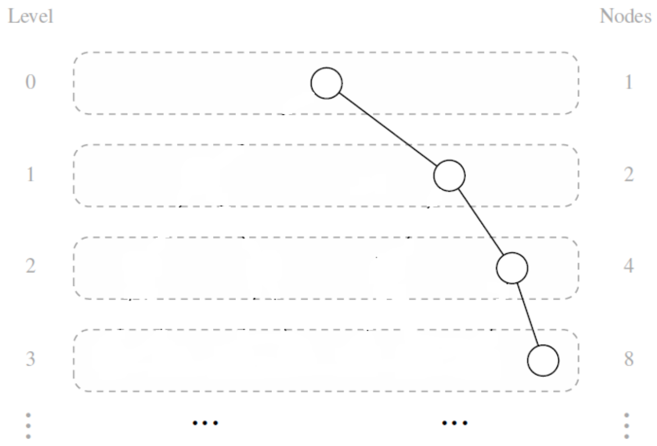
Properties of Binary Trees

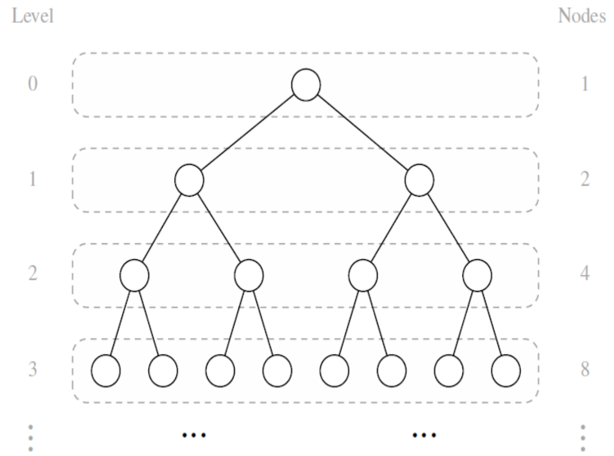
Given height h of a binary tree T,

Minimum number of internal nodes is h.
Maximum number of internal nodes is 2^h-1

So,

$h <= n_i <= 2^h-1$





Properties of Binary Trees

Given a binary tree T with n nodes,

Minimum height is log(n+1)-1. (Why?)

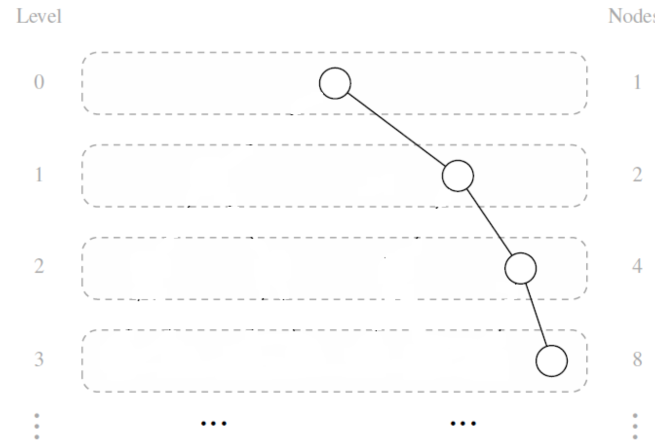
Maximum height is n-1

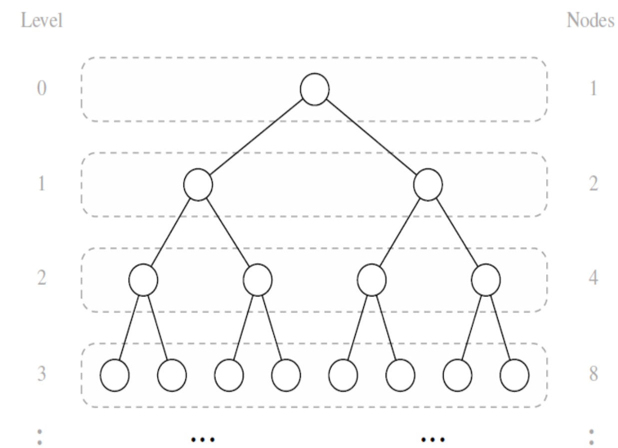
So,

log(n+1)-1 <= h <= n-1

$$2^{h+1}-1=n$$
$$2^{h+1}=n+1$$
$$h+1=\log_2(n+1)$$
$$h=\log_2(n+1)-1$$

$$h+1=n$$
$$h=n-1$$





Properties of Binary Trees

If T is a proper binary tree, then following inequalities hold:

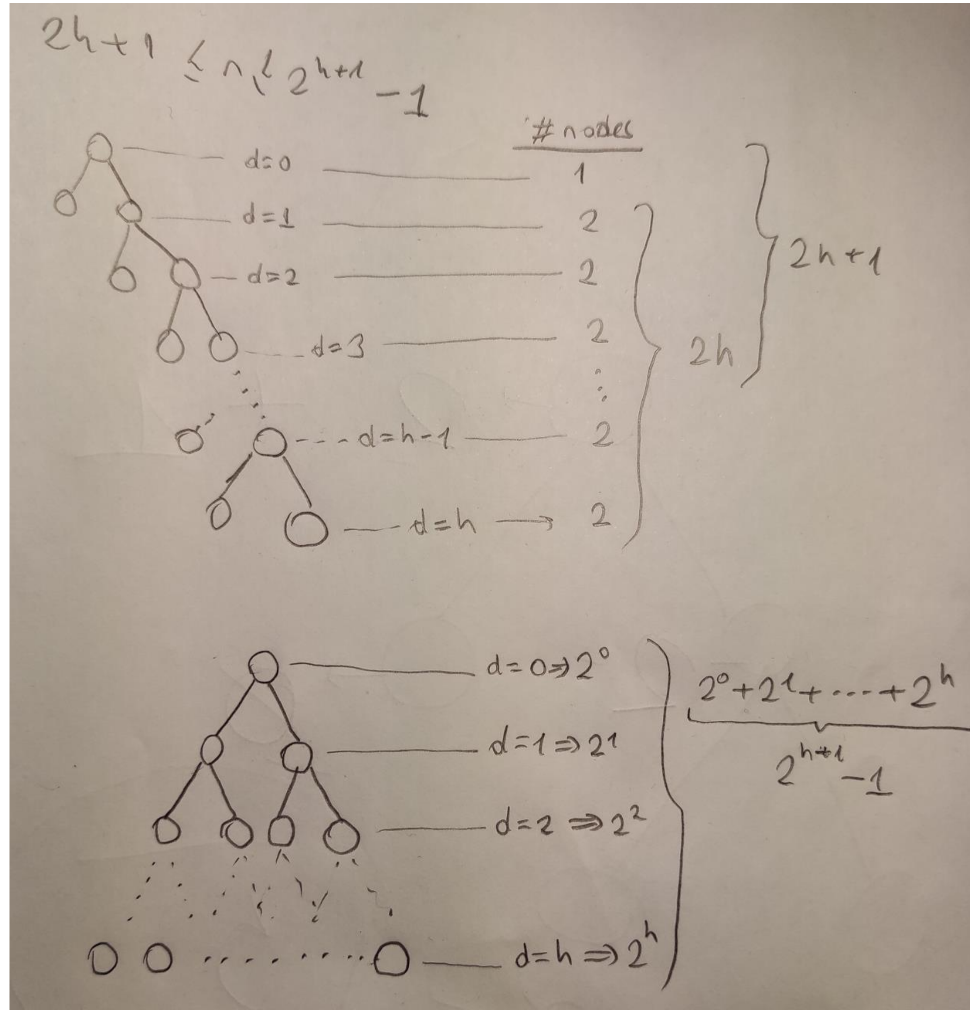
$2h+1 <= n <= 2^{h+1}-1$

$h+1 <= n_e <= 2^h$

$h <= n_i <= 2^h-1$

$\log(n+1)-1 <= h <= (n-1)/2$

Why? (Take it as a HW!)



Binary Tree ADT

In addition to Tree ADT, Binary Tree ADT has the following operations:

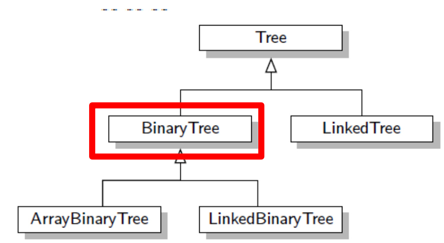
T.left(p) (return the left childs position)

T.right(p) (return the left childs position)

T.sibling(p) (return the position of the sibling)

BinaryTree Abstract Class

```
class BinaryTree(Tree):
  """Abstract base class representing a binary tree struc

  # -------------------- additional abstract methods -------
  def left(self, p):
    """Return a Position representing p's left child.

    Return None if p does not have a left child.
    """
    raise NotImplementedError('must be implemented by subclass')

  def right(self, p):
    """Return a Position representing p's right child.

    Return None if p does not have a right child.
    """
    raise NotImplementedError('must be implemented by subclass')
```



BinaryTree Abstract Class

```
def sibling(self, p):
  """Return a Position representing p's sibling (or None if no sibling)."""
  parent = self.parent(p)
  if parent is None:                          # p must be the root
    return None                               # root has no sibling
  else:
    if p == self.left(parent):
      return self.right(parent)               # possibly None
    else:
      return self.left(parent)                # possibly None
```

BinaryTree Abstract Class

```
def children(self, p):
  """Generate an iteration of Positions representing p's children."""
  if self.left(p) is not None:
    yield self.left(p)
  if self.right(p) is not None:
    yield self.right(p)
```

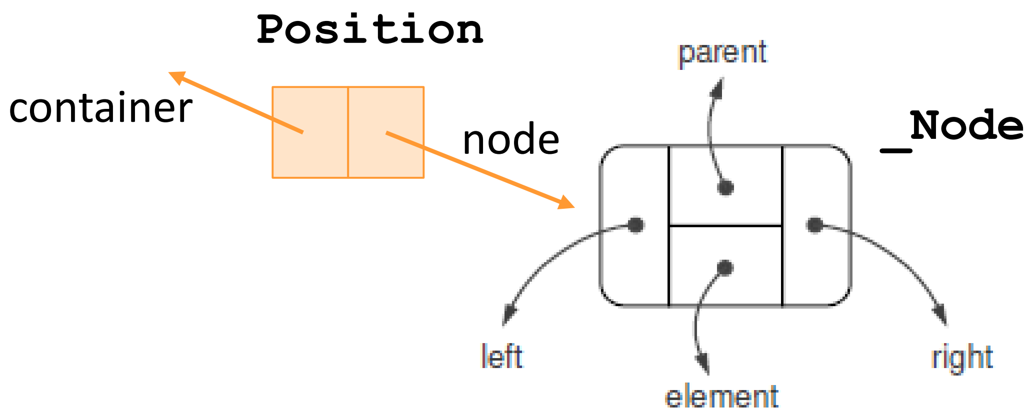
BinaryTree Implementation

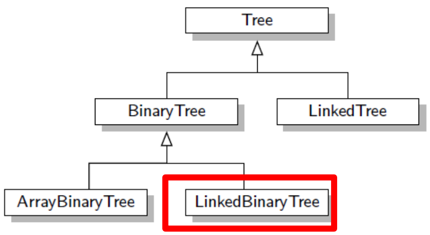
Two choices for internal tree representation:

Linked Structure

Array-based Representation

LinkedBinaryTree





Node representation of linked structure for binary tree.

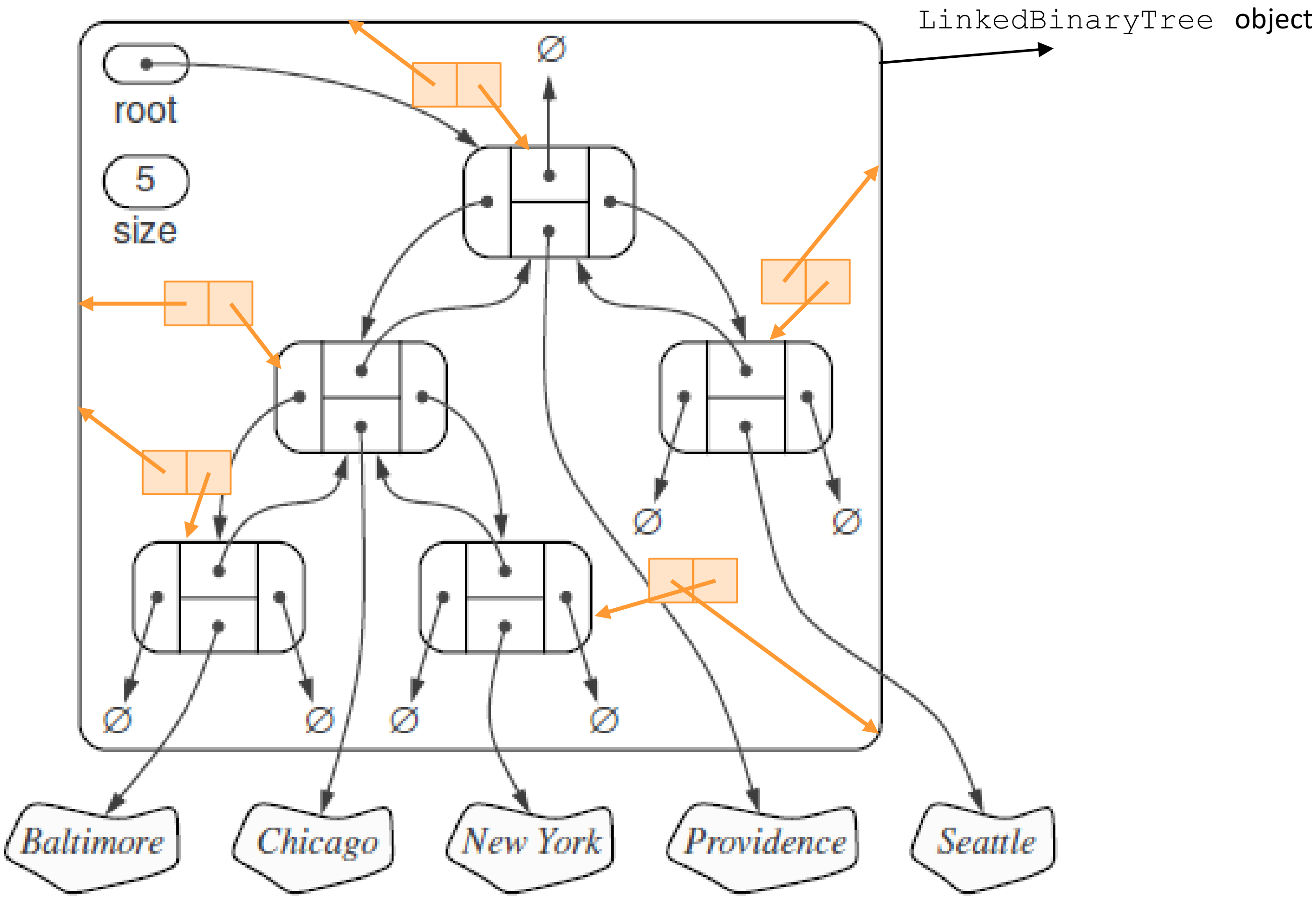
If node is root, then parent is `None`.

If node does not have left or right child, then the relevant pointer(s) are `None`.

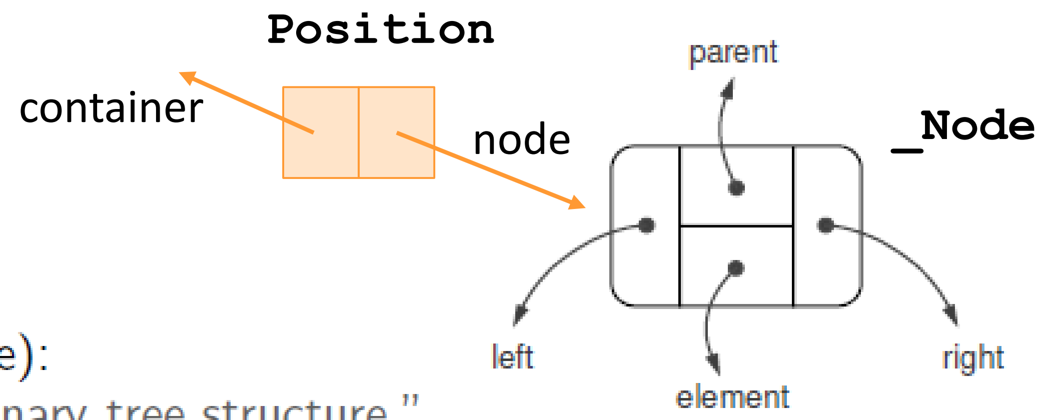
Position is merely an interface for the internal `_Node` class.

It also identifies the tie between the `LinkedBinaryTree` object and the `_Node` object.

LinkedBinaryTree

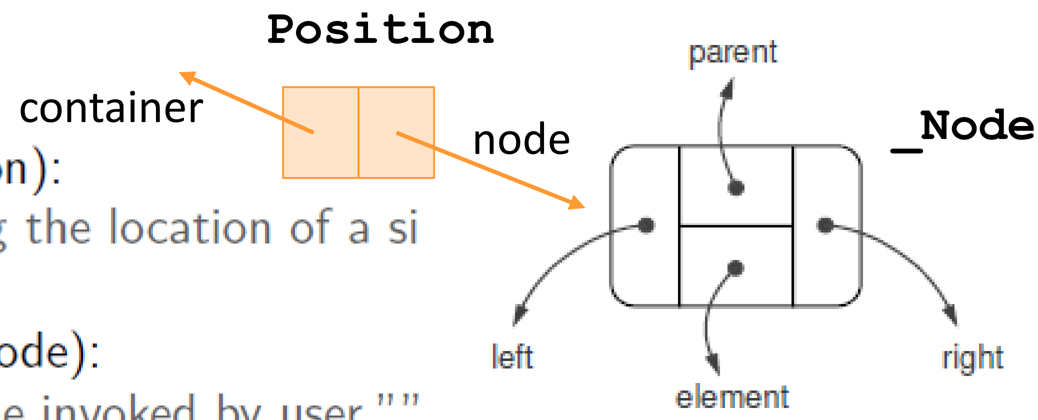


LinkedBinaryTree




```
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

```

LinkedBinaryTree




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

  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
```

Recall that `__ne__` was implemented in `Tree` base class

LinkedBinaryTree

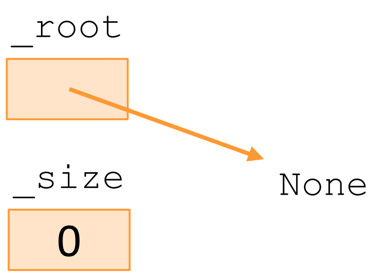
```
def _validate(self, p):
  """Return associated node, if position is valid."""
  if not isinstance(p, self.Position):
    raise TypeError('p must be proper Position type')
  if p._container is not self:
    raise ValueError('p does not belong to this container')
  if p._node._parent is p._node:  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

LinkedBinaryTree

```
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
```

LinkedBinaryTree

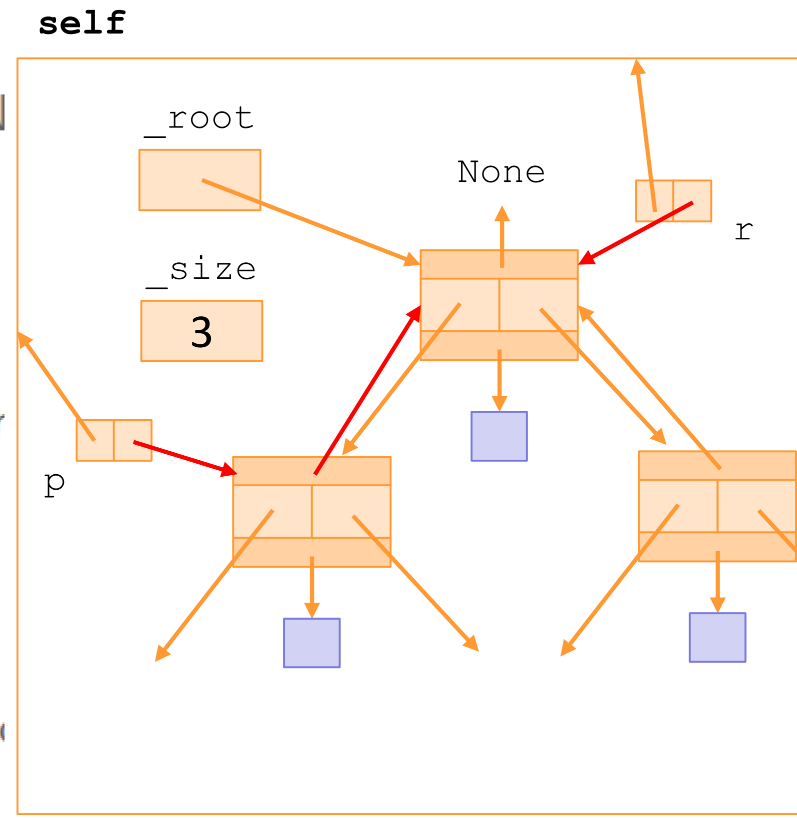
```
#------------------------ binary tree constructor ------------------------
def __init__(self):
  """Create an initially empty binary tree."""
  self._root = None
  self._size = 0
```




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

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

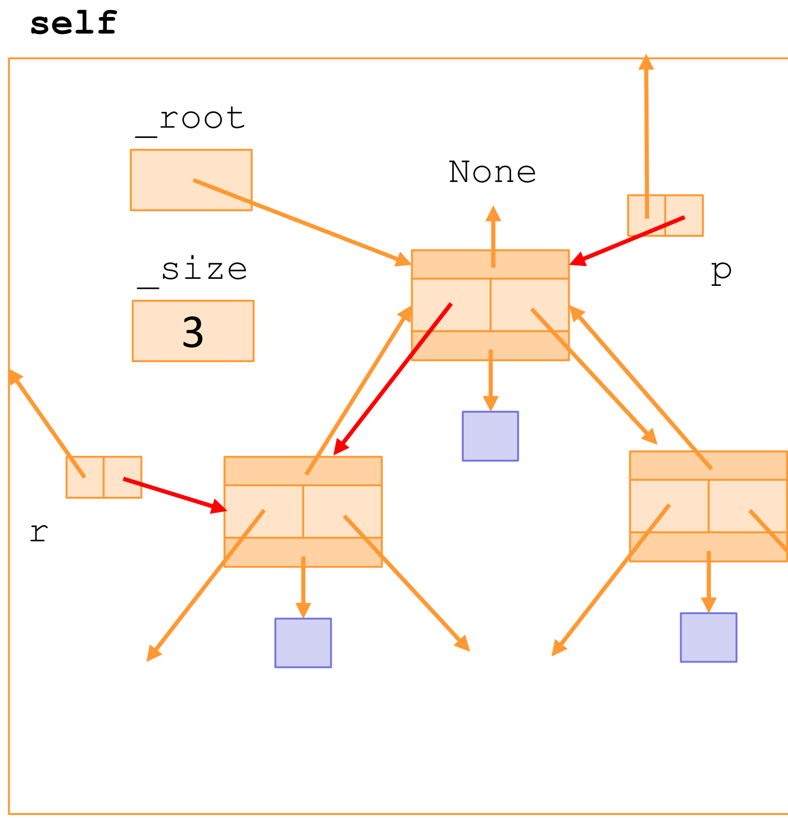
LinkedBinaryTree

```
def parent(self, p):
  """Return the Position of p's parent (or N
  node = self._validate(p)
  return self._make_position(node._parent)

def left(self, p):
  """Return the Position of p's left child (or
  node = self._validate(p)
  return self._make_position(node._left)

def right(self, p):
  """Return the Position of p's right child (
  node = self._validate(p)
  return self._make_position(node._right)
```



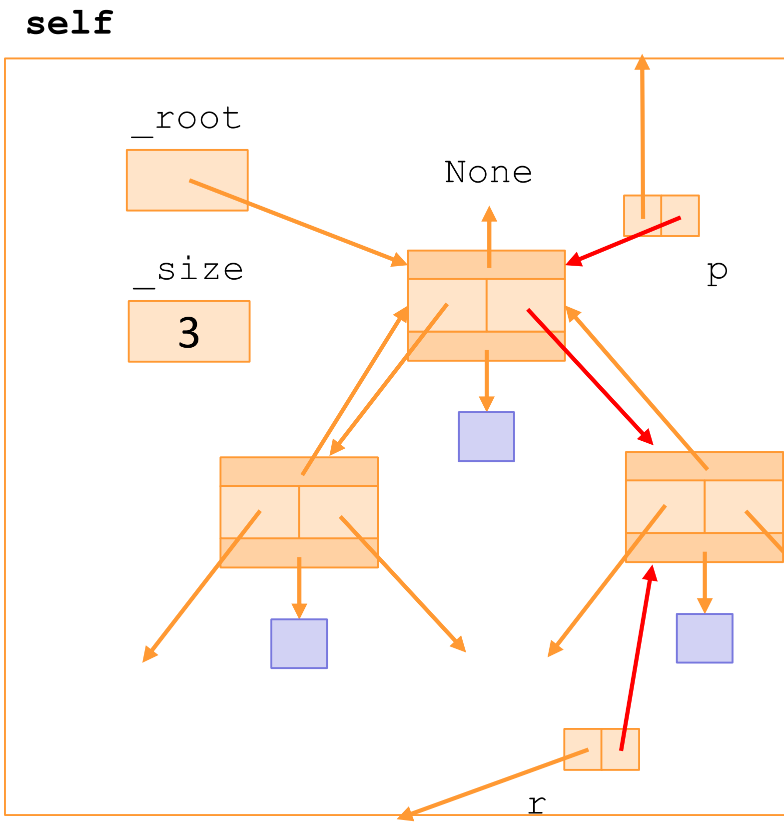
LinkedBinaryTree

```
def parent(self, p):
  """Return the Position of p's parent (or N
  node = self._validate(p)
  return self._make_position(node._parent)

def left(self, p):
  """Return the Position of p's left child (or
  node = self._validate(p)
  return self._make_position(node._left)

def right(self, p):
  """Return the Position of p's right child (
  node = self._validate(p)
  return self._make_position(node._right)
```



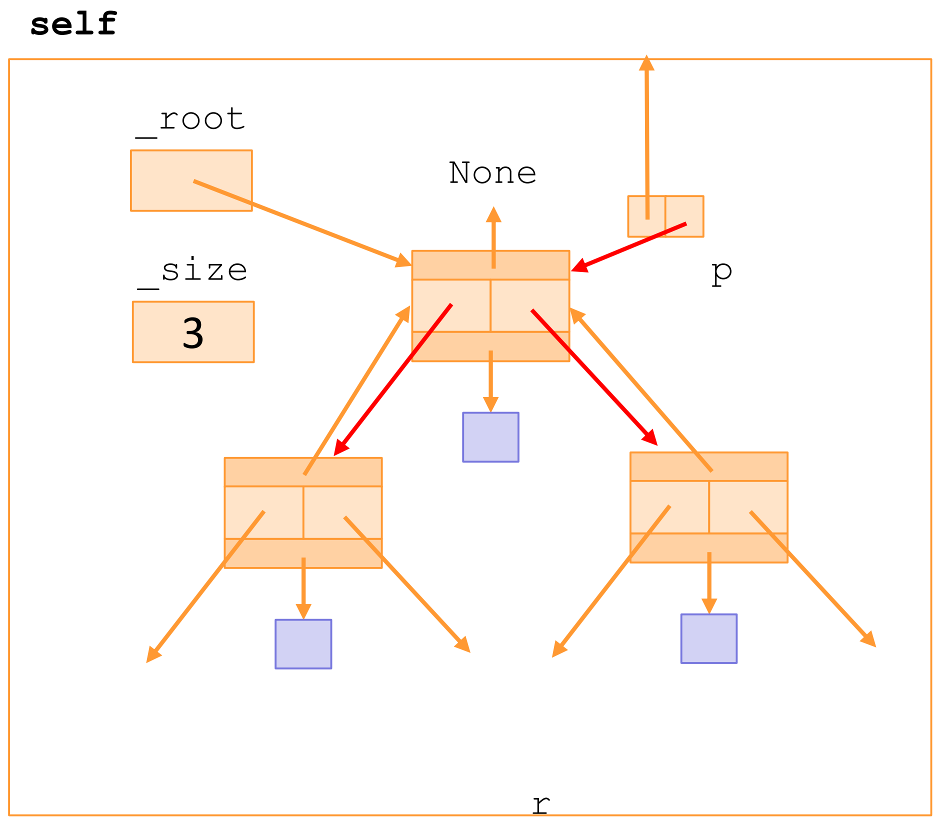
LinkedBinaryTree

```
def parent(self, p):
  """Return the Position of p's parent (or N
  node = self._validate(p)
  return self._make_position(node._parent)

def left(self, p):
  """Return the Position of p's left child (or
  node = self._validate(p)
  return self._make_position(node._left)

def right(self, p):
  """Return the Position of p's right child (
  node = self._validate(p)
  return self._make_position(node._right)
```

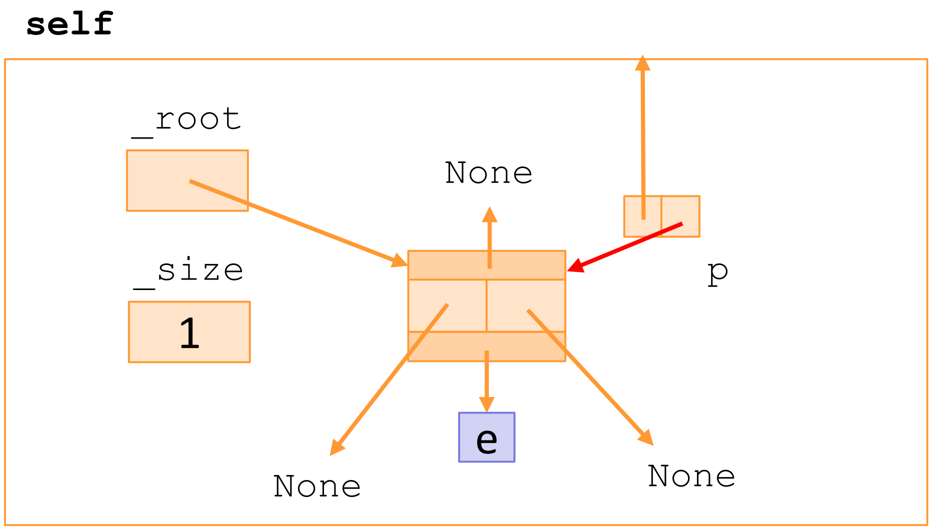


LinkedBinaryTree

```
def num_children(self, p):
  """Return the number of chil
  node = self._validate(p)
  count = 0
  if node._left is not None:
    count += 1
  if node._right is not None:
    count += 1
  return count
```



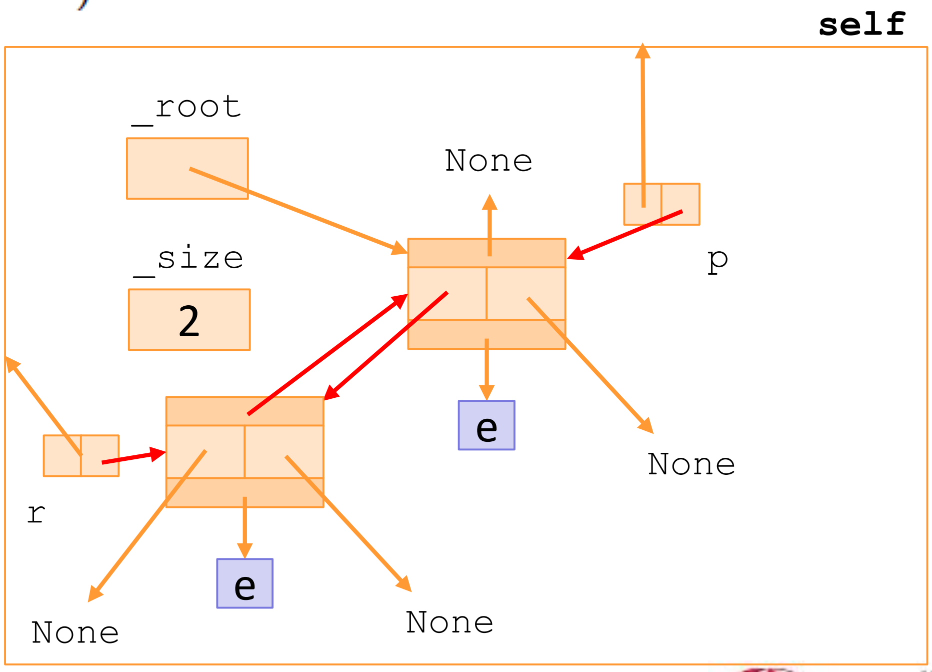
LinkedBinaryTree




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

    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)
```

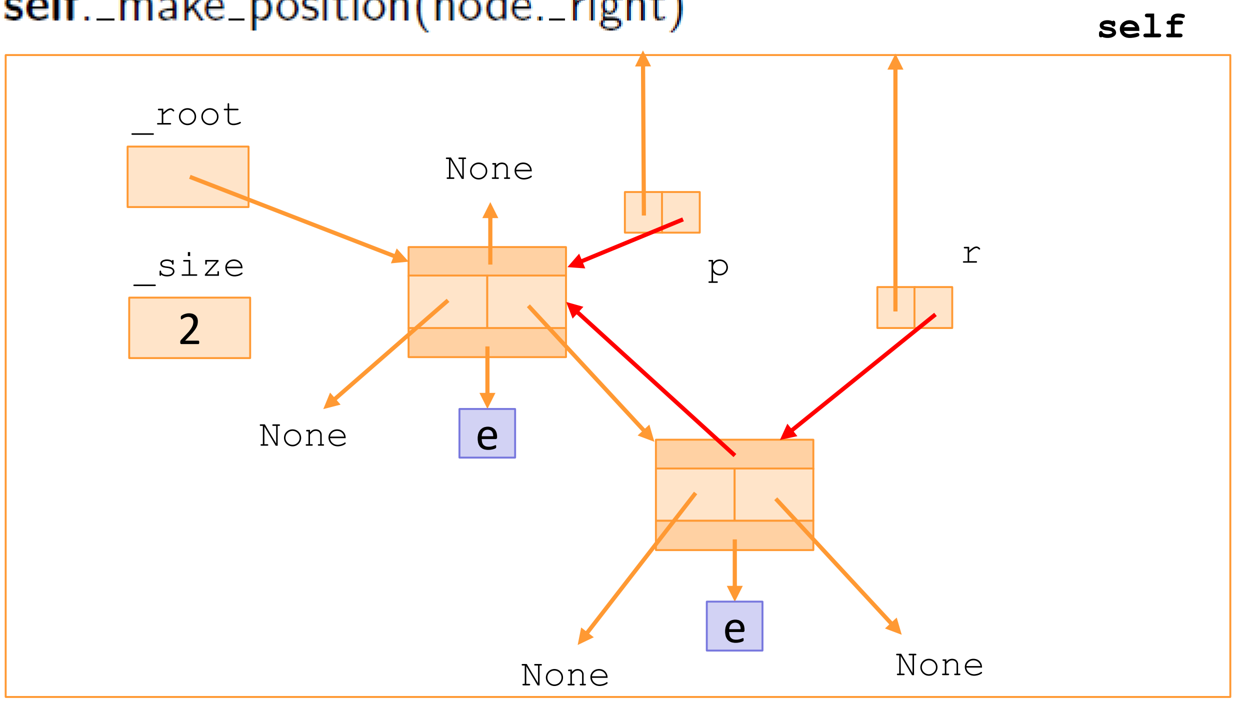
LinkedBinaryTree

```
def _add_left(self, p, e):
    node = self._validate(p)
    if node._left is not None: raise ValueError('Left child exists')
    self._size += 1
    node._left = self._Node(e, node)
    return self._make_position(node._left)
```



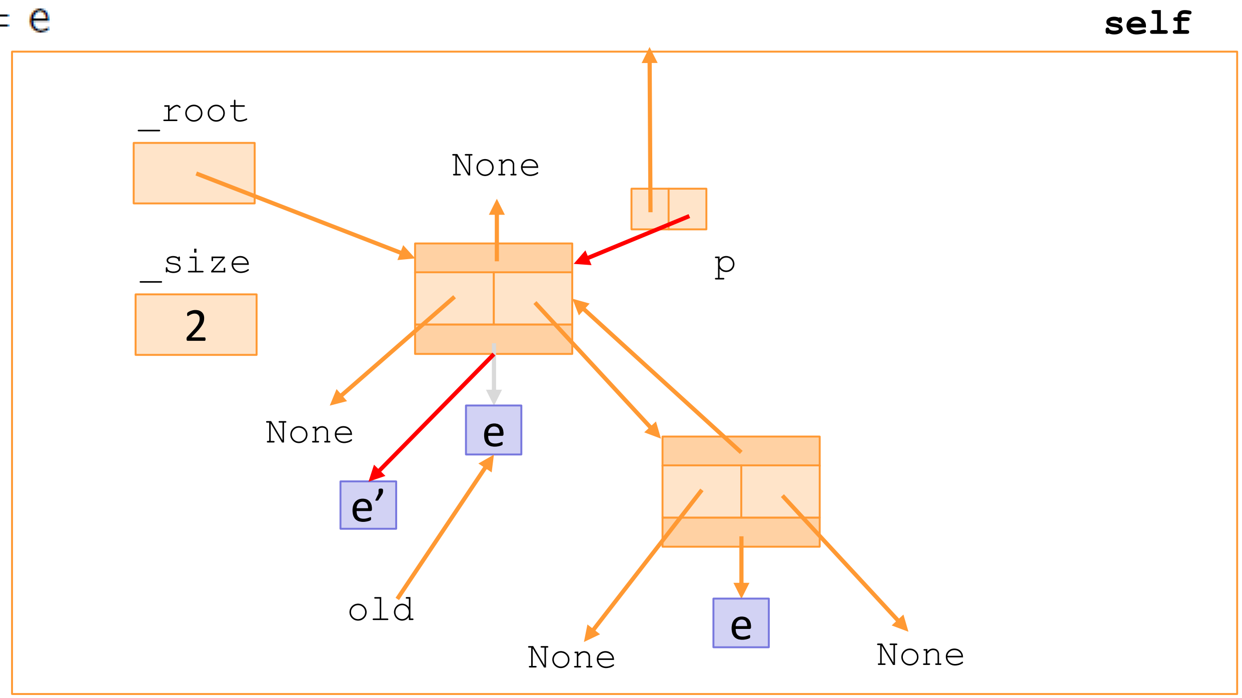
LinkedBinaryTree

```
def _add_right(self, p, e):
    node = self._validate(p)
    if node._right is not None: raise ValueError('Right child exists')
    self._size += 1
    node._right = self._Node(e, node)
    return self._make_position(node._right)
```



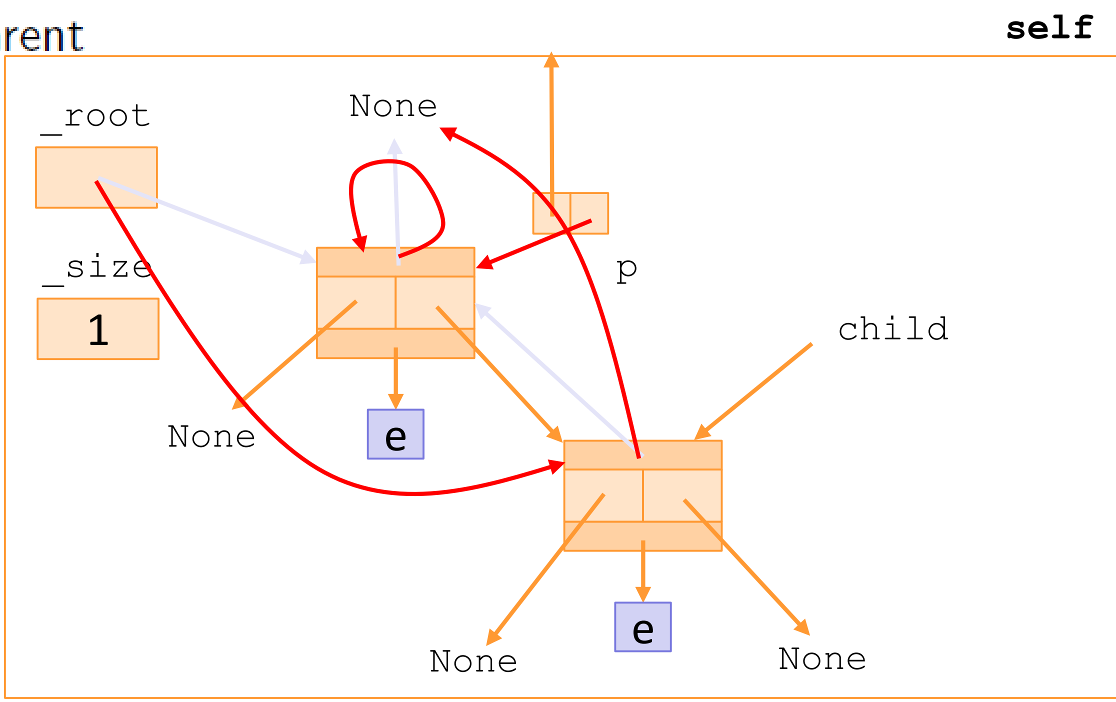
LinkedBinaryTree

```
def _replace(self, p, e):
    node = self._validate(p)
    old = node._element
    node._element = e
    return old
```



LinkedBinaryTree

```
def _delete(self, p):
    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
    if child is not None:
        child._parent = node._parent
    if node is self._root:
        self._root = child
    else:
        parent = node._parent
        if node is parent._left:
            parent._left = child
        else:
            parent._right = child
    self._size -= 1
    node._parent = node
    return node._element
```

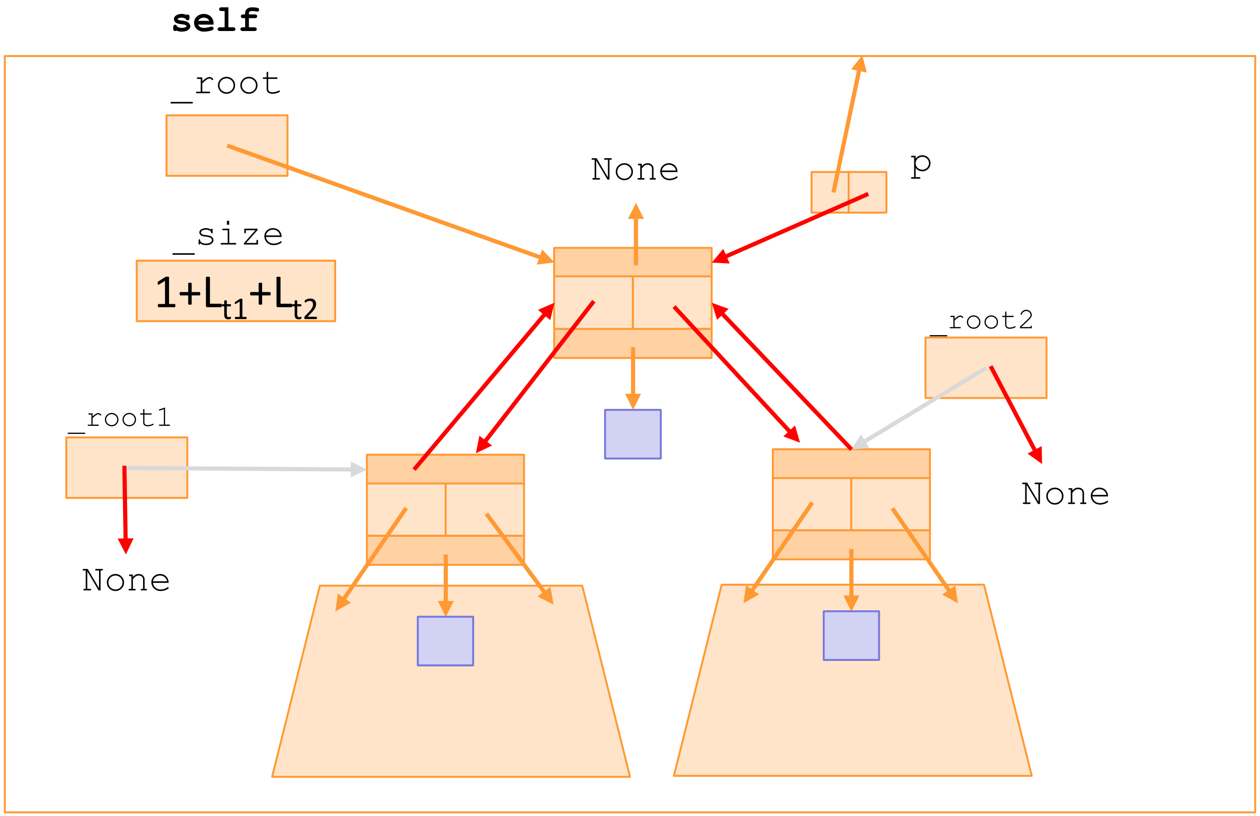


LinkedBinaryTree

```
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
```

LinkedBinaryTree

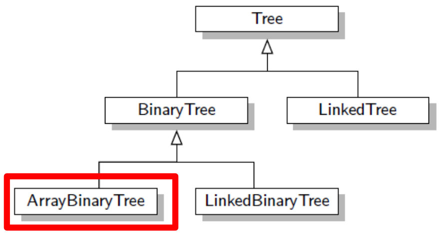
```
if not t1.is_empty():
    t1._root._parent = node
    node._left = t1._root
    t1._root = None
    t1._size = 0
if not t2.is_empty():
    t2._root._parent = node
    node._right = t2._root
    t2._root = None
    t2._size = 0
```



Performance of LinkedBinaryTree

Operation	Running Time
len, is_empty	$O(1)$
root, parent, left, right, sibling, children, num_children	$O(1)$
is_root, is_leaf	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$
add_root, add_left, add_right, replace, delete, attach	$O(1)$

ArrayBinaryTree



For storing positions in custom node objects, we can use an array-based structure.

Addresses of each node of tree can be stored in a list.

Given a position p, let f(p) is the index of the p in the array.

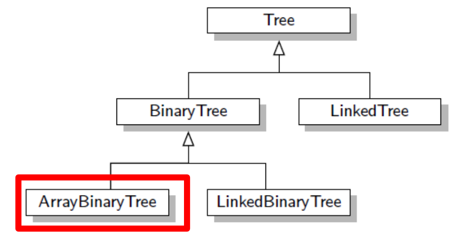
If p is a left child of q, then f(p) = 2f(q)+1.

If p is a right child of q, then f(p) = 2f(q)+2.

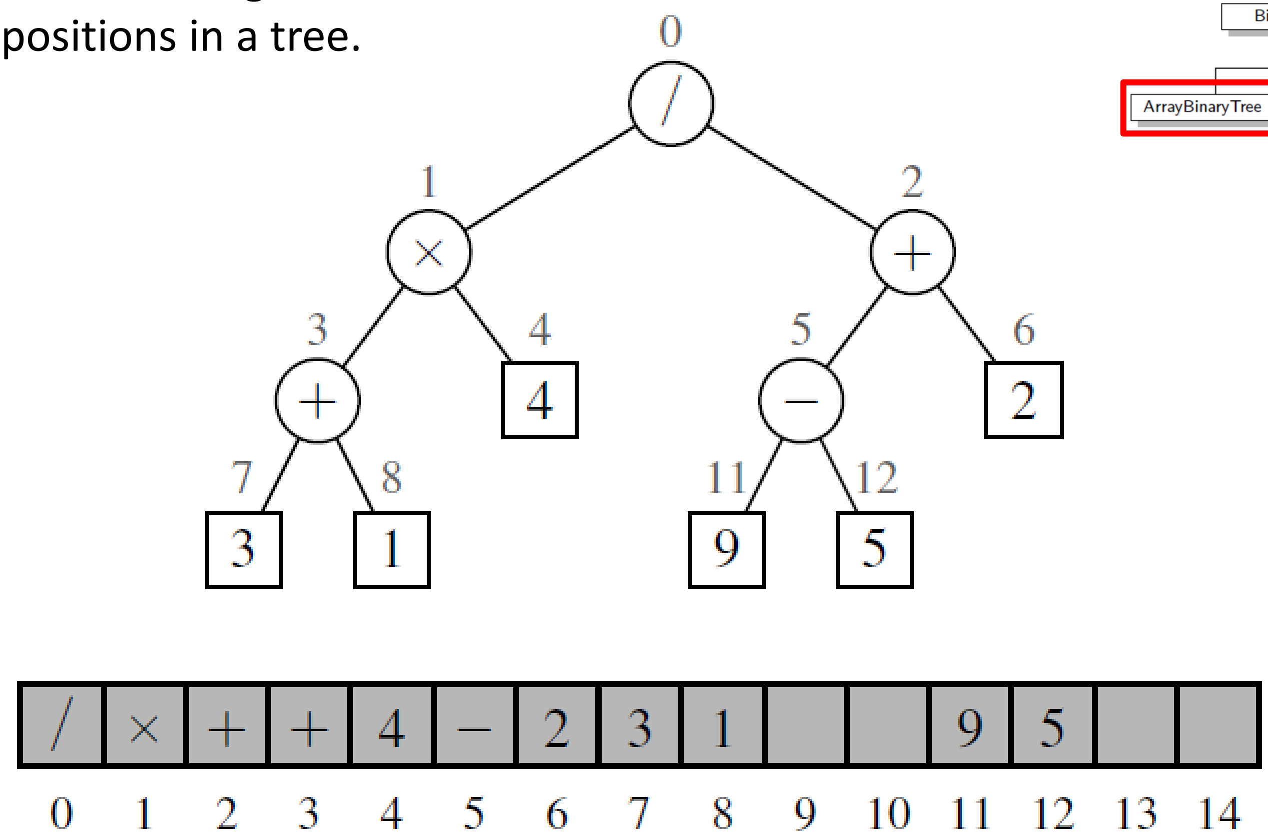
Level Numbering

With such mechanism, we can easily calculate positions of parent, left, and right children.

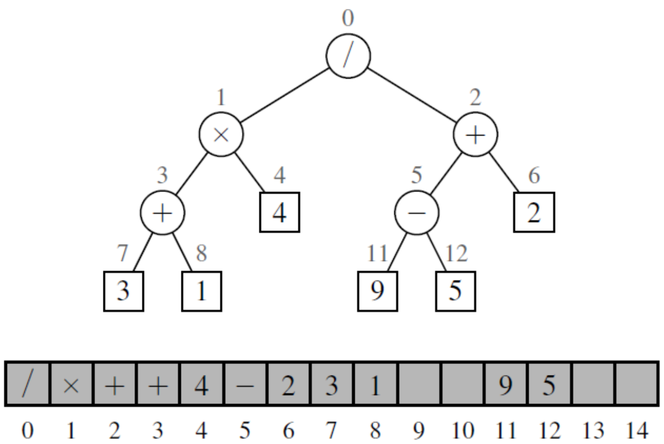
ArrayBinaryTree



So, f(p) can be much larger than the number of positions in a tree.



ArrayBinaryTree



Let n be the number of nodes in T, and f_M be the maximum value of f(p).

Array requires $N=f_M+1$ capacity.

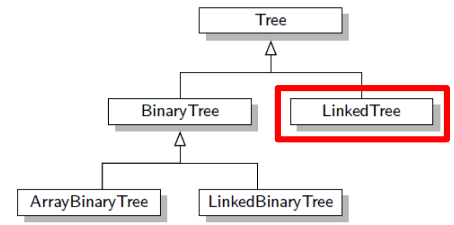
In the worst case, N could be 2^n-1. (Why?)

(Hint: Try to add new nodes "only as right child".)

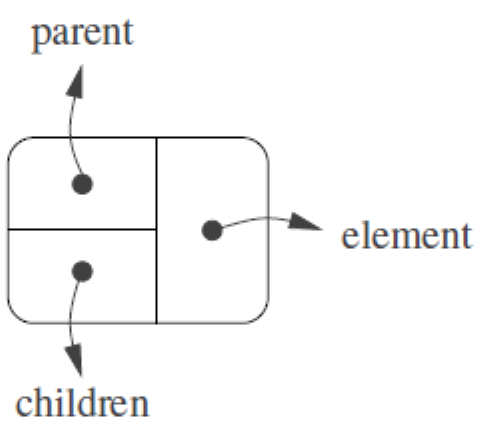
Deleting a node takes O(n) time, in the worst case. Why?

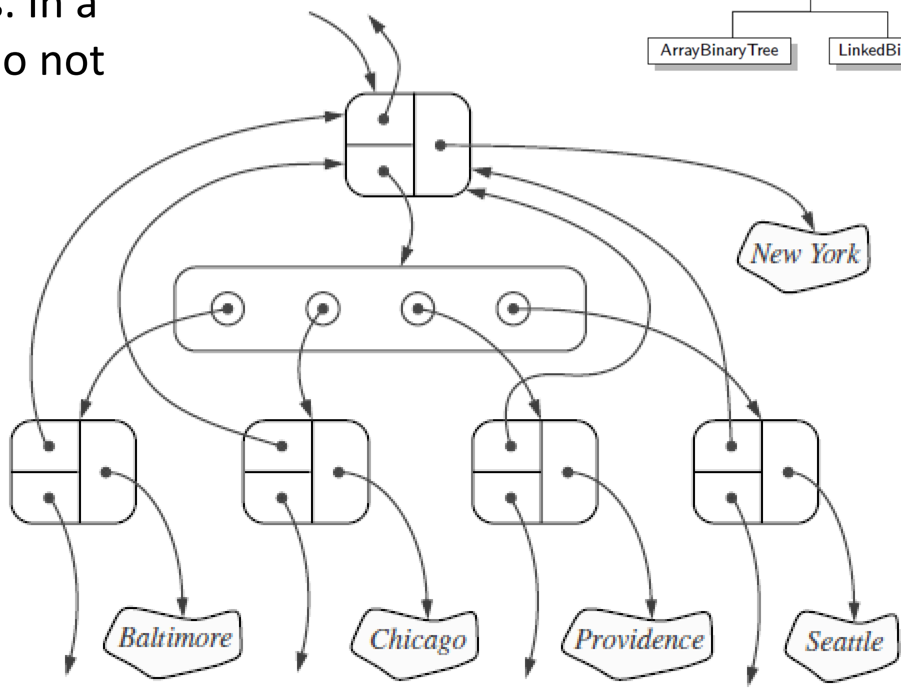
(Hint: Try to find a case where deleting a node would require shifting of all nodes.)

LinkedTree



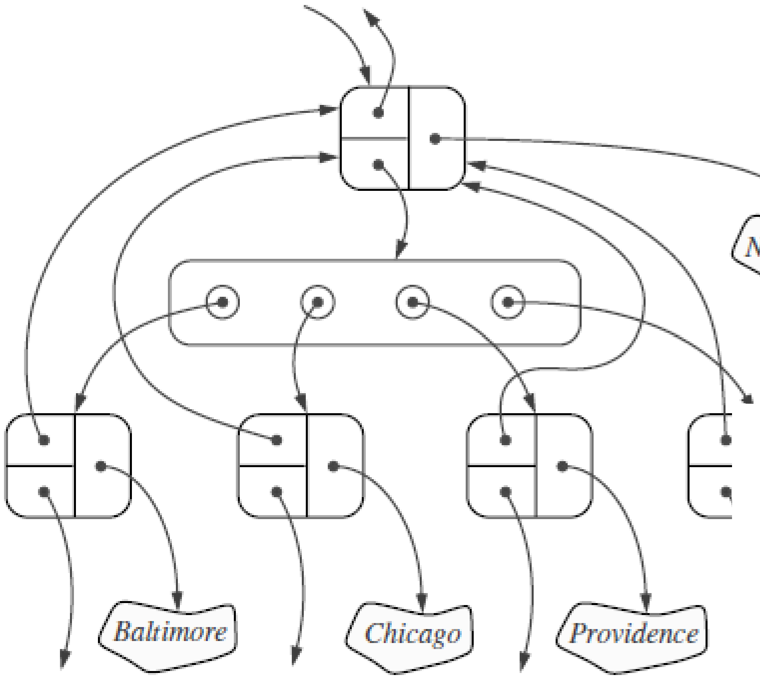
Binary tree is a special tree where each node can have at most two nodes. In a more general tree definition we do not have such a restriction.





We can store pointers to the children of a node to an array-based structure (e.g., list).

LinkedTree



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 Traversals

Traversal of a tree means visiting and (possibly) doing something with each node of a tree.

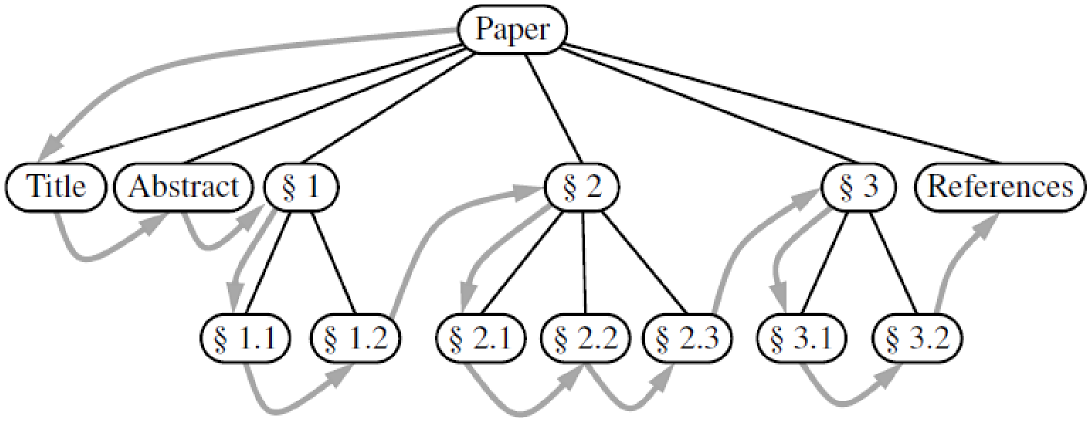
- Preorder Traversal
- Postorder Traversal
- Breadth-first Traversal
- Inorder Traversal (applicable to only binary trees)

Preorder Traversal

Given a root node, initially the root node is visited and then the same operation is repeated for each subtrees that are rooted at the children of root node. (Complexity is O(n). Why?)

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

Selfish parents!

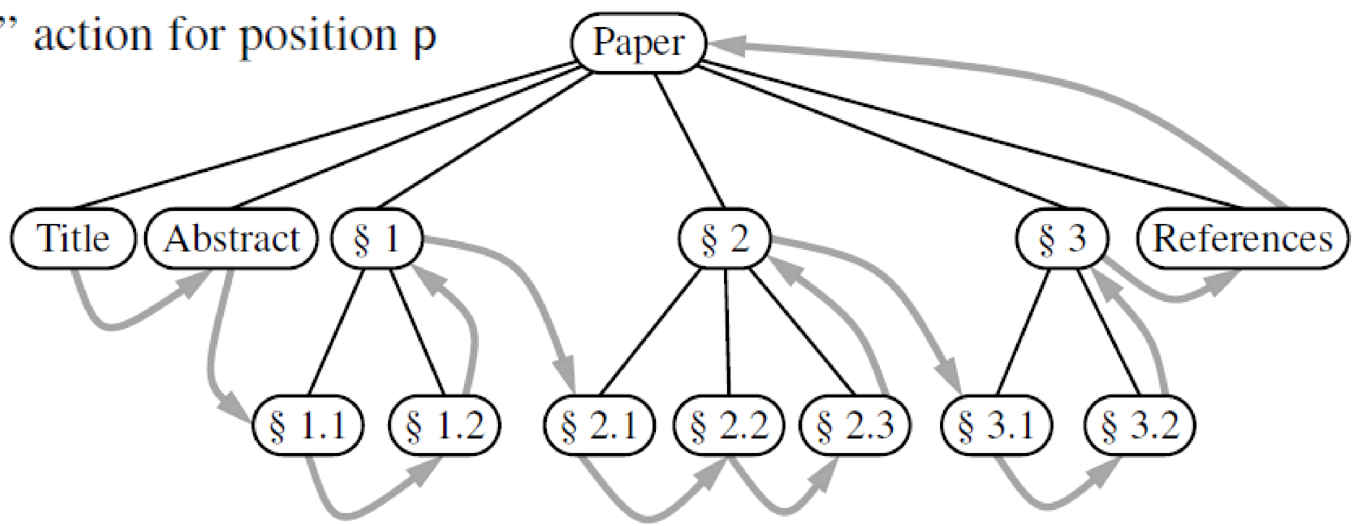


Postorder Traversal

Given a root node, first each of the subtrees that are rooted at the children of root node are visited than the root node is visited. (Complexity is O(n). Why?)

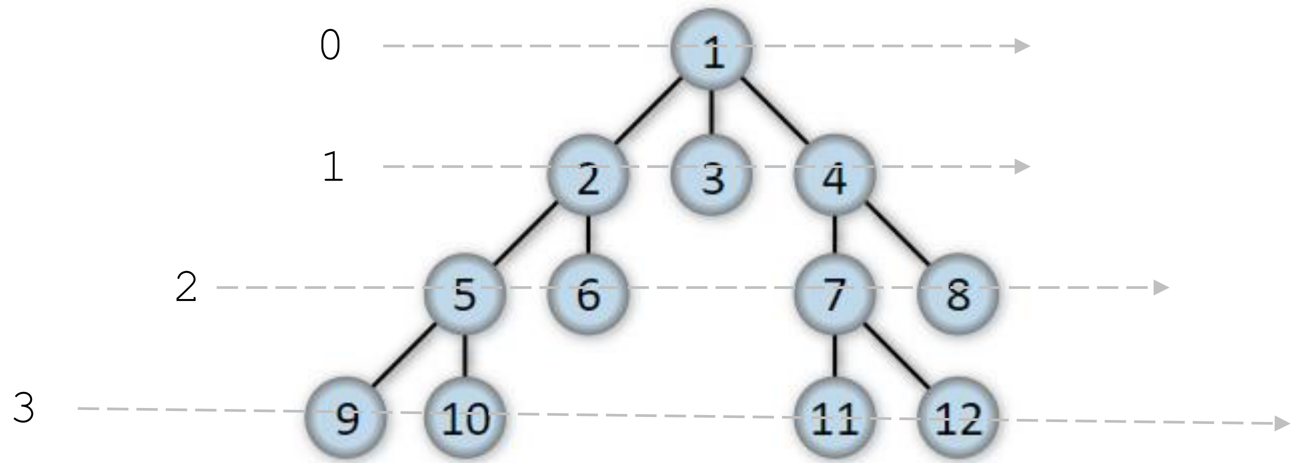
Algorithm postorder(T, p):
  for each child c in T.children(p) **do**
    postorder(T, c)
  perform the "visit" action for position p

Altruist parents!
(Su küçüğün.)



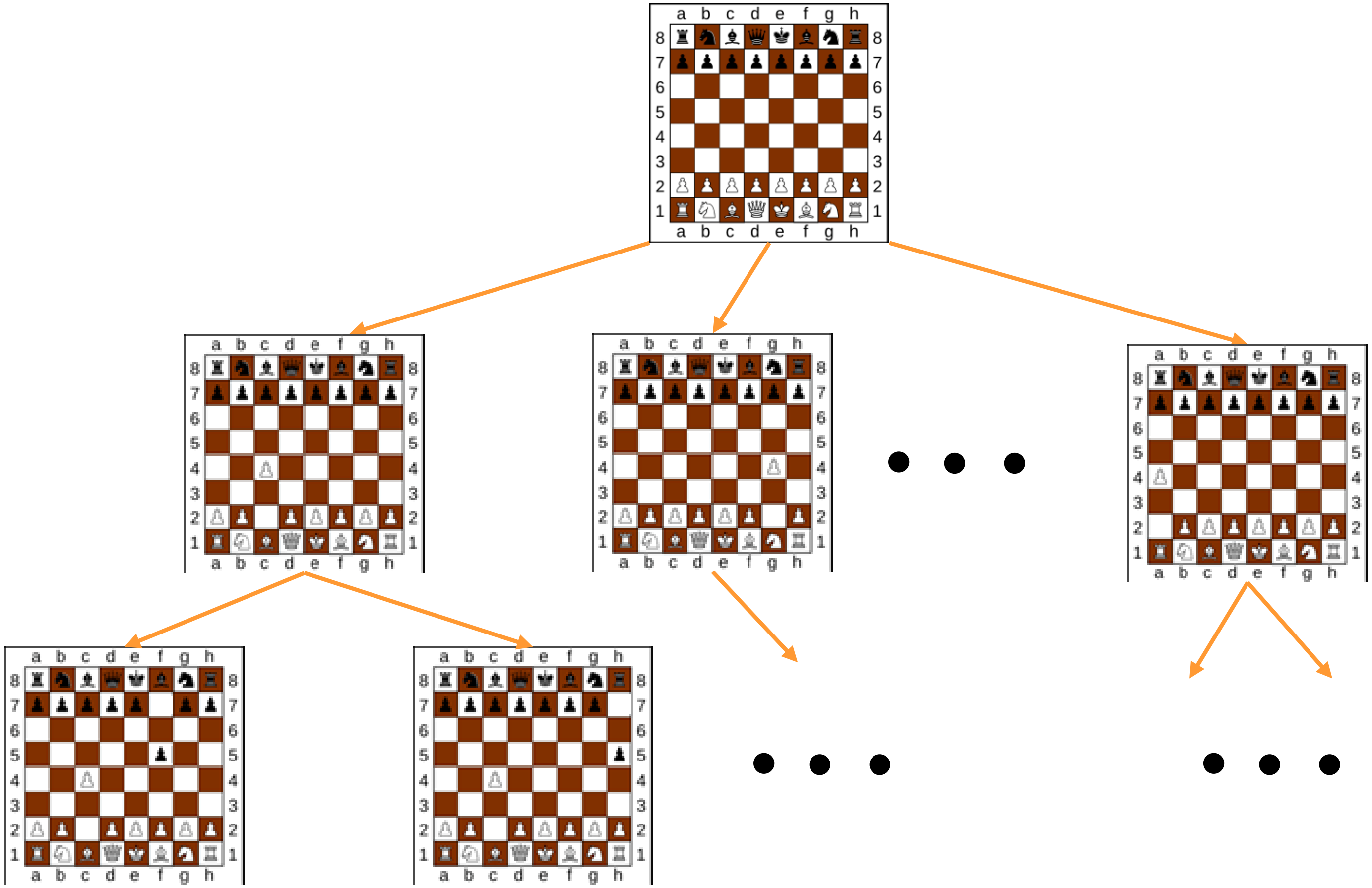
Breadth-first Traversal

Starting from depth=0, visit all nodes at depth d and then continue with the nodes at depth d+1.



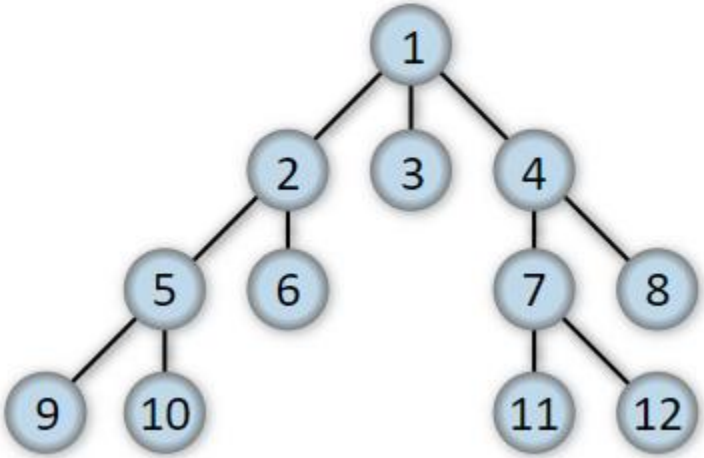
Breadth-first Traversal

A well-known example is game trees.



Breadth-first Traversal

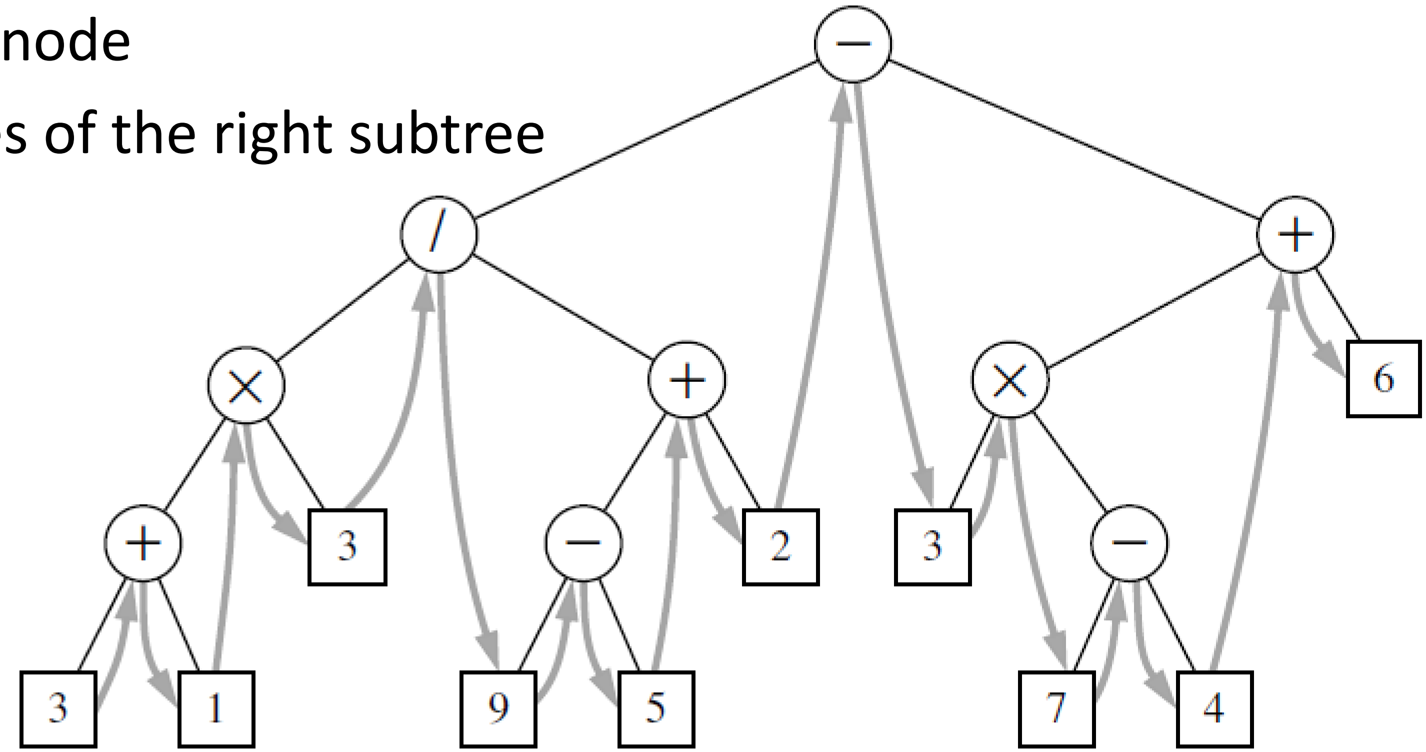
Algorithm breadthfirst(T):
  Initialize queue Q to contain T.root( )
  while Q not empty **do**
    p = Q.dequeue( )
    perform the "visit" action for position p
    for each child c in T.children(p) **do**
      Q.enqueue(c)   {add p's children to



Inorder Traversal (Binary Tree)

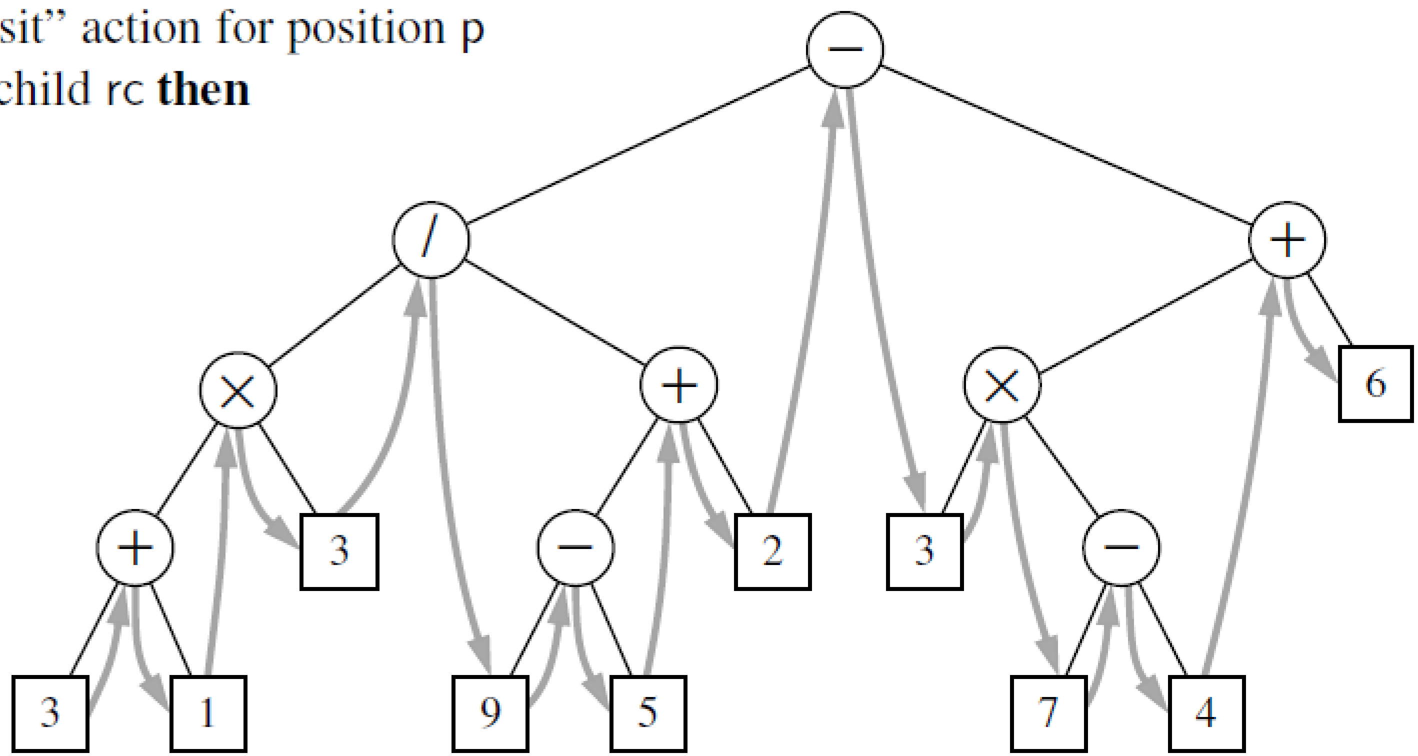
The nodes of a binary tree is visited in the following order:

1. Nodes of the left subtree
2. Root node
3. Nodes of the right subtree



Inorder Traversal (Binary Tree)

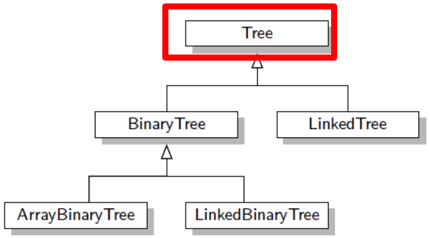
Algorithm inorder(p):
  if p has a left child lc **then**
    inorder(lc)
  perform the "visit" action for position p
  if p has a right child rc **then**
    inorder(rc)



Traversal Implementations

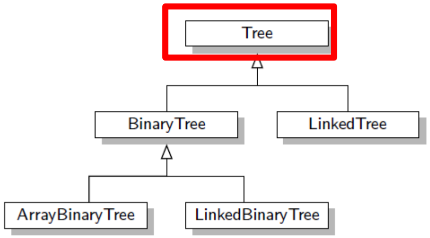
Here we will be implementing the **T.positions()** and **iter(T)** methods of tree ADT.

- T.positions(): Iteration on all <u>positions</u>
- iter(T): Iteration on <u>elements</u> of tree



Traversal Implementations

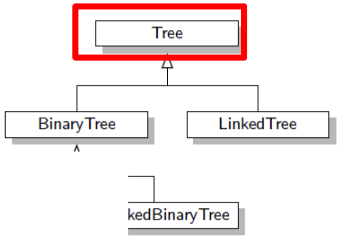
iter(T)




```
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
```

Traversal Implementations

T.positions()

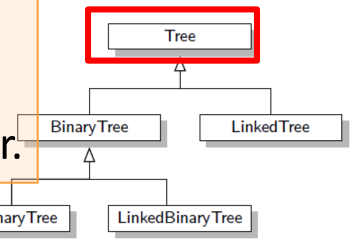



```
def positions(self):
  """Generate an iteration of the tree's positions."""
  return self.preorder( )           # return entire preorder iteration
```

A generator that returns positions according to some defined order.
The ordering logic of the generation is defined in this <u>generator</u>.

Traversal Implementations

A generator for the whole tree
Just a wrapper for the internal `_subtree_preorder` generator.



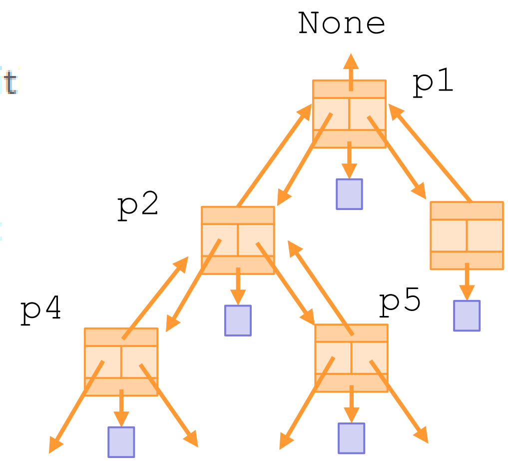

```
def preorder(self):
  """Generate a preorder iteration of positions in the tree."""
  if not self.is_empty():
    for p in self._subtree_preorder(self.root()):      # start recursio
      yield p

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
```

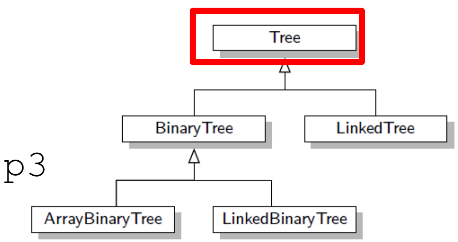
A generator that could be used for any intermediate node (i.e., traversing a subtree)

Traversal Implementations

```python
def _subtree_preorder(self, p):
  """Generate a preorder iteration of posit
  yield p
  for c in self.children(p):
    for other in self._subtree_preorder(c):
      yield other
```

```python
g = t._subtree_preorder(self, p1)
next(g) # Iteration 1 --> yields p1
next(g) # Iteration 2 --> yields p2
next(g) # Iteration 3 --> yields p4
next(g) # Iteration 4 --> yields p5
next(g) # Iteration 5 --> yields p3
```



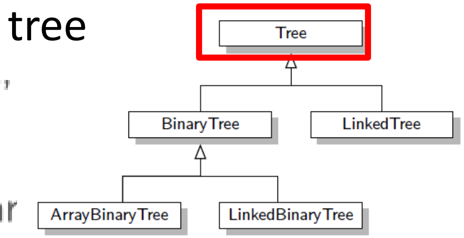


CALL STACK

		call: _subtree_preorder(p4) > yield p for c in self.children(p): for other in self._subtree_preorder(c) yield(other)	call: _subtree_preorder(p5) > yield p for c in self.children(p): for other in self._subtree_preorder(c) yield(other)	
	call: _subtree_preorder(p2) > yield p for c in self.children(p): for other in self._subtree_preorder(c) yield(other)	call: _subtree_preorder(p2) yield p for c in self.children(p): for other in self._subtree_preorder(c) > yield(other)	call: _subtree_preorder(p2) yield p for c in self.children(p): for other in self._subtree_preorder(c) > yield(other)	call: _subtree_preorder(p3) > yield p for c in self.children(p): for other in self._subtree_preorder(c) yield(other)
call: _subtree_preorder(p1) > yield p for c in self.children(p): for other in self._subtree_preorder(c) yield(other)	call: _subtree_preorder(p1) yield p for c in self.children(p): for other in self._subtree_preorder(c) > yield(other)	call: _subtree_preorder(p1) yield p for c in self.children(p): for other in self._subtree_preorder(c) > yield(other)	call: _subtree_preorder(p1) yield p for c in self.children(p): for other in self._subtree_preorder(c) > yield(other)	call: _subtree_preorder(p1) yield p for c in self.children(p): for other in self._subtree_preorder(c) > yield(other)
First Yield	Second Yield	Third Yield	Fourth Yield	Fifth Yield

Traversal Implementations

A generator for the whole tree

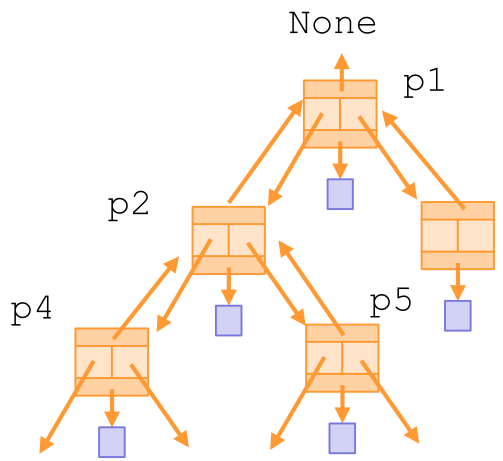


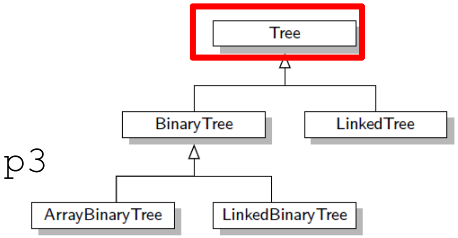

```
def postorder(self):
  """Generate a postorder iteration of positions in the tree."""
  if not self.is_empty():
    for p in self._subtree_postorder(self.root()):        # star
      yield p

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
```

A generator that could be used with any intermediate node (i.e., traversing a subtree)

Traversal Implementations

```
def _subtree_postorder(self, p):
  for c in self.children(p):
    for other in self._subtree_postorder(c):
      yield other
  yield p
```






```
g = t._subtree_postorder(self, p1)
next(g) # Iteration 1 --> yields p4
next(g) # Iteration 2 --> yields p5
next(g) # Iteration 3 --> yields p2
next(g) # Iteration 4 --> yields p3
next(g) # Iteration 5 --> yields p1
```

CALL STACK

call: _subtree_preorder (p4)
for c in self.children(p):
for other in self._subtree_postorder(c):
yield other
>yield p
	call: _subtree_preorder (p5)
	for c in self.children(p):
	for other in self._subtree_postorder(c):
	yield other
	>yield p			

call: _subtree_preorder (p2)
for c in self.children(p):
for other in self._subtree_postorder(c):
>yield other
yield p
	call: _subtree_preorder (p2)
	for c in self.children(p):
	for other in self._subtree_postorder(c):
	>yield other
	yield p
		call: _subtree_preorder (p2)
		for c in self.children(p):
		for other in self._subtree_postorder(c):
		yield other
		> yield p
			call: _subtree_preorder (p3)
			for c in self.children(p):
			for other in self._subtree_postorder(c):
			yield other
			> yield p	

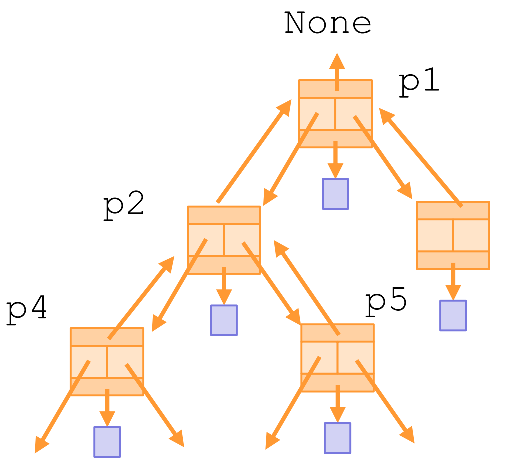
call: _subtree_preorder (p1)
for c in self.children(p):
for other in self._subtree_postorder(c):
>yield other
yield p
	call: _subtree_preorder (p1)
	for c in self.children(p):
	for other in self._subtree_postorder(c):
	>yield other
	yield p
		call: _subtree_preorder (p1)
		for c in self.children(p):
		for other in self._subtree_postorder(c):
		>yield other
		yield p
			call: _subtree_preorder (p1)
			for c in self.children(p):
			for other in self._subtree_postorder(c):
			>yield other
			yield p
				call: _subtree_preorder (p1)
				for c in self.children(p):
				for other in self._subtree_postorder(c):
				yield other
				> yield p

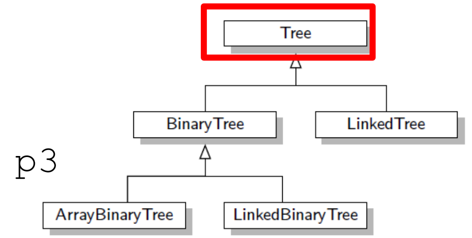
First Yield	Second Yield	Third Yield	Fourth Yield	Fifth Yield

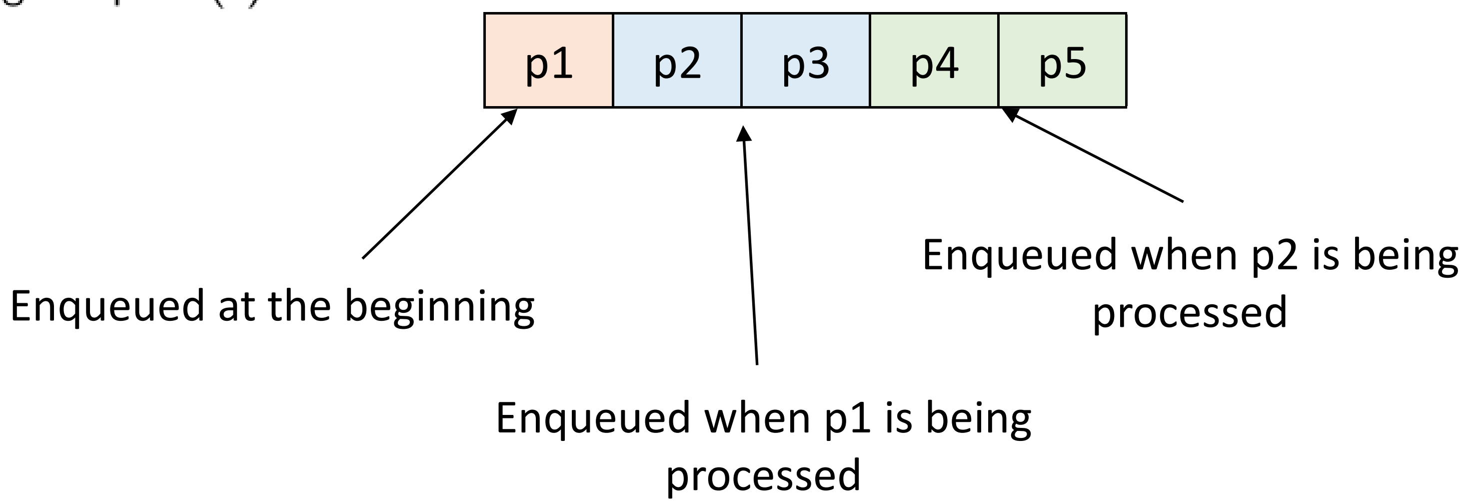
CALL STACK

Traversal Implementations

```
def breadthfirst(self):
  """Generate a breadth-first iter
  if not self.is_empty( ):
    fringe = LinkedQueue( )
    fringe.enqueue(self.root( ))
    while not fringe.is_empty( ):
      p = fringe.dequeue( )
      yield p
      for c in self.children(p):
        fringe.enqueue(c)
```

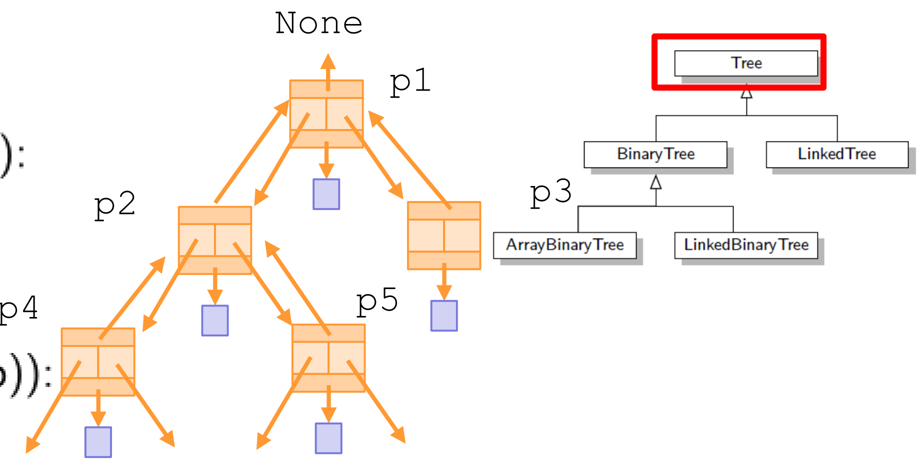






Traversal Implementations

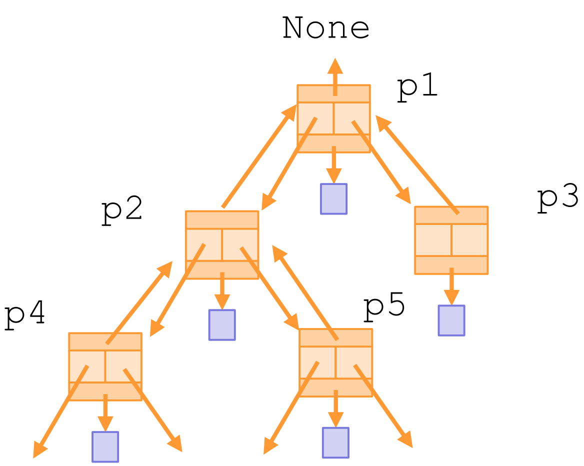
```
def _subtree_inorder(self, p):
    if self.left(p) is not None:
        for other in self._subtree_inorder(self.left(p)):
            yield other
    yield p
    if self.right(p) is not None:
        for other in self._subtree_inorder(self.right(p)):
            yield other
```




```
g = t._subtree_inorder(self, p1)
next(g) # Iteration 1 --> yields p4
next(g) # Iteration 2 --> yields p2
next(g) # Iteration 3 --> yields p5
next(g) # Iteration 4 --> yields p1
next(g) # Iteration 5 --> yields p3
```

Traversal Implementations

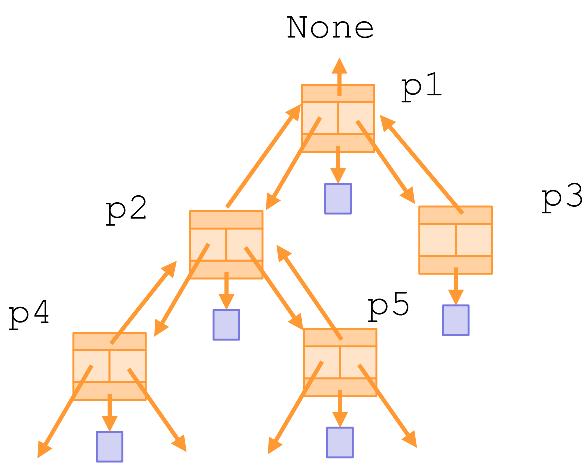
```
g = t._subtree_inorder(self, p1)
next(g) # Iteration 1 --> yields p4
next(g) # Iteration 2 --> yields p2
next(g) # Iteration 3 --> yields p5
next(g) # Iteration 4 --> yields p1
next(g) # Iteration 5 --> yields p3
```




```
_subtree_inorder(p1):
if self.left(p) is not None:
    for other in self._subtree_inorder(self.left(p)):
        >>> yield other
yield p
if self.right(p) is not None:
    for other in self._subtree_inorder(self.right(p)):
        yield other

_subtree_inorder(p2):
if self.left(p) is not None:
    for other in self._subtree_inorder(self.left(p)):
     >>> yield other
yield p
if self.right(p) is not None:
    for other in self._subtree_inorder(self.right(p)):
        yield other

_subtree_inorder(p4):
if self.left(p) is not None:
    for other in self._subtree_inorder(self.left(p)):
        yield other
>>> yield p
if self.right(p) is not None:
    for other in self._subtree_inorder(self.right(p)):
        yield other
```

Traversal Implementations

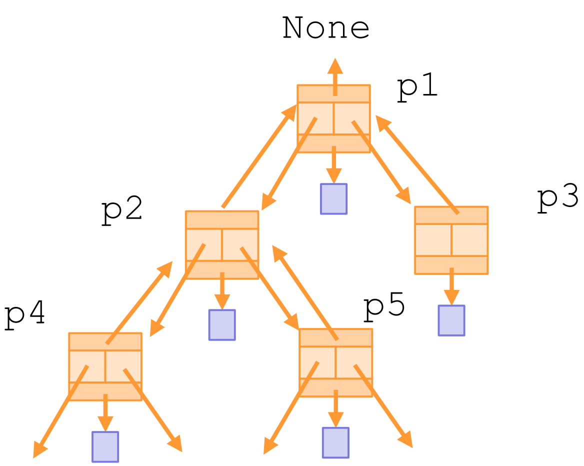
```
g = t._subtree_inorder(self, p1)
next(g) # Iteration 1 --> yields p4
next(g) # Iteration 2 --> yields p2
next(g) # Iteration 3 --> yields p5
next(g) # Iteration 4 --> yields p1
next(g) # Iteration 5 --> yields p3
```




```
_subtree_inorder(p1):
if self.left(p) is not None:
    for other in self._subtree_inorder(self.left(p)):
        >>> yield other
yield p
if self.right(p) is not None:
    for other in self._subtree_inorder(self.right(p)):
        yield other

_subtree_inorder(p2):
if self.left(p) is not None:
    for other in self._subtree_inorder(self.left(p)):
        yield other
>>> yield p
if self.right(p) is not None:
    for other in self._subtree_inorder(self.right(p)):
        yield other
```

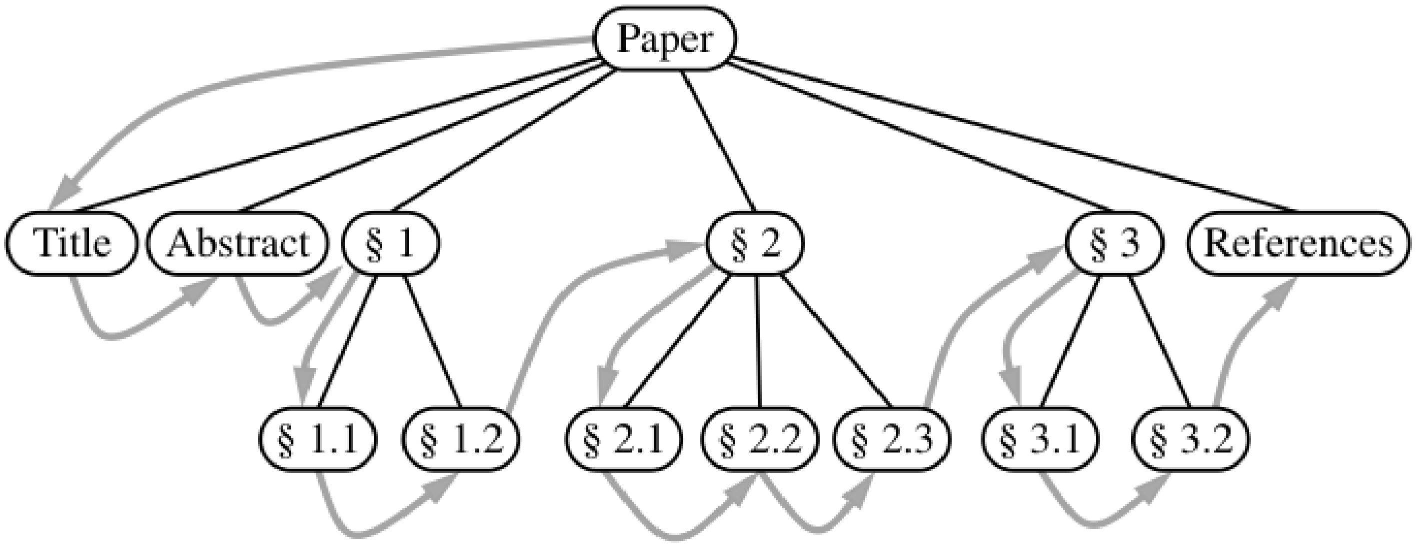
Traversal Implementations

```
g = t._subtree_inorder(self, p1)
next(g) # Iteration 1 --> yields p4
next(g) # Iteration 2 --> yields p2
next(g) # Iteration 3 --> yields p5
next(g) # Iteration 4 --> yields p1
next(g) # Iteration 5 --> yields p3
```




```
_subtree_inorder(p1):
if self.left(p) is not None:
    for other in self._subtree_inorder(self.left(p)):
        >>> yield other
yield p
if self.right(p) is not None:
    for other in self._subtree_inorder(self.right(p)):
        yield other

_subtree_inorder(p2):
if self.left(p) is not None:
    for other in self._subtree_inorder(self.left(p)):
        yield other
yield p
if self.right(p) is not None:
    for other in self._subtree_inorder(self.right(p)):
        >>> yield other

_subtree_inorder(p5):
if self.left(p) is not None:
    for other in self._subtree_inorder(self.left(p)):
        yield other
>>> yield p
if self.right(p) is not None:
    for other in self._subtree_inorder(self.right(p)):
        yield other
```

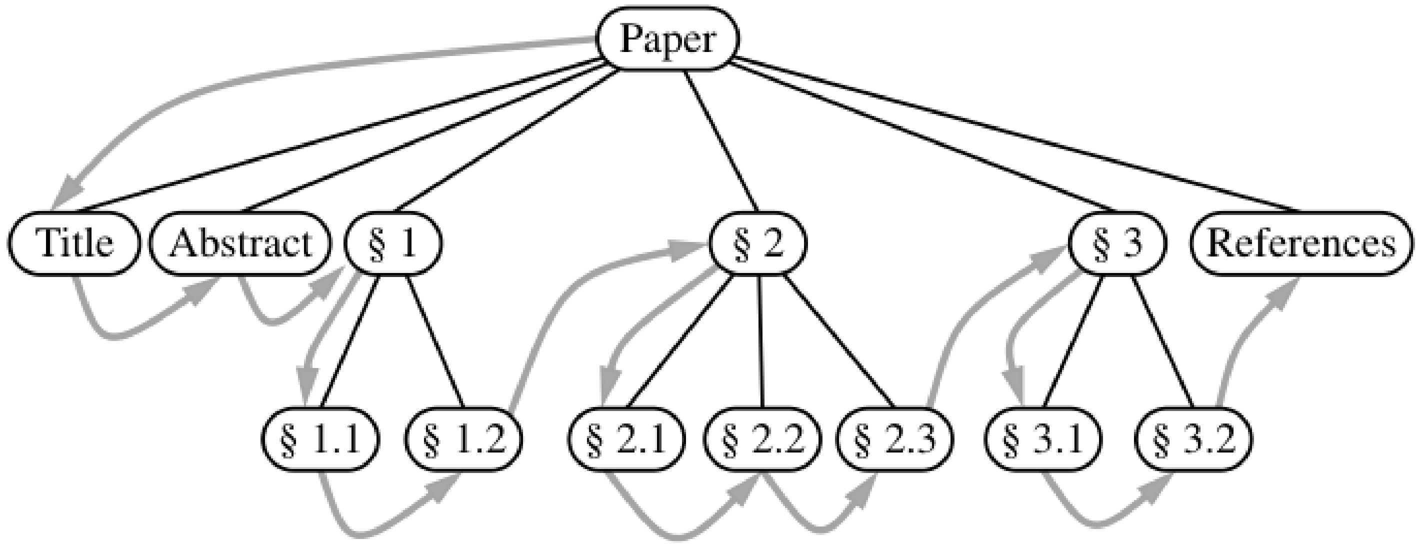
Example Application of Tree




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

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

Example Application of Tree

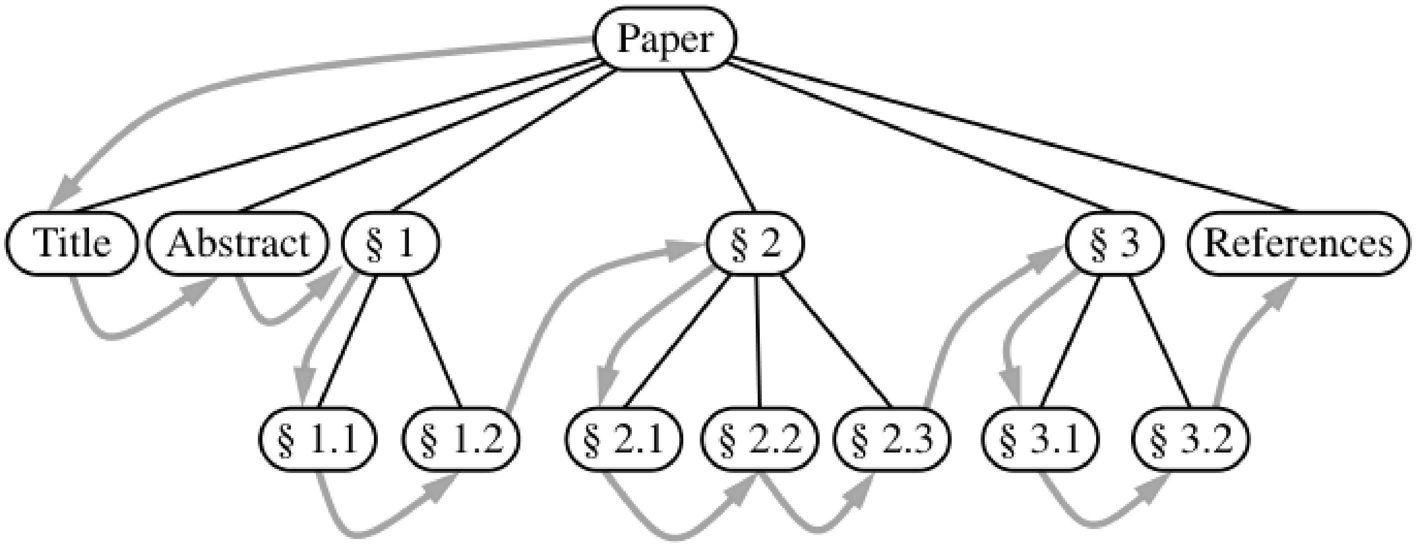



```
Paper
  Title
  Abstract
  §1
    §1.1
    §1.2
  §2
    §2.1
    ...
```

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

What about the complexity?

Example Application of Tree




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

A better approach

```python
def preorder_indent(T, p, d):
    print(2 * d * ' ' + str(p.element()))
    for c in T.children(p):
        preorder_indent(T, c, d+1)
```

What about the complexity?