

# Trees

Data Structures & Algorithms with Python

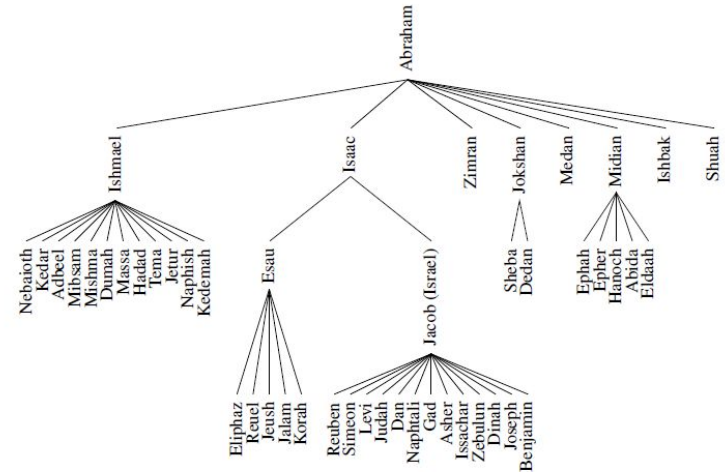
Lecture 8

# Overview

- Introduction to Trees
  - Definitions, Properties
  - Tree ADT - Computing Depth & Height
- Binary Trees
- Implementing Trees
  - Using Linked Lists
  - Using Arrays
- Tree Traversal Algorithms
- Summary

# Introduction to Trees

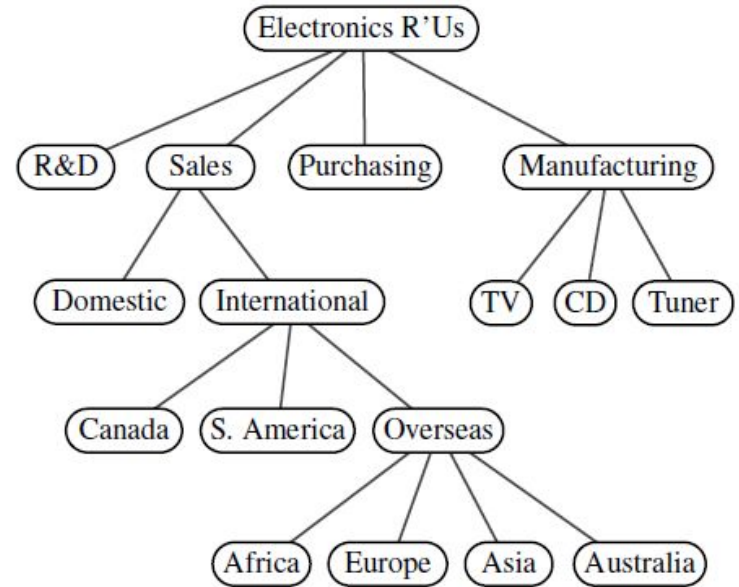
- A nonlinear data structure.
- Provide faster algorithms for search and retrieval of data compared to linear data structures - arrays or linked-lists.
- Provide a natural organization of data which is inherently hierarchical.



A family tree showing some descendants of Abraham, as recorded in Genesis, chapters 25–36.

# Tree Definitions and Properties

- A **tree** is an abstract data type that stores elements hierarchically.
- With the exception of the top element, each element in a tree has a **parent** element and zero or more **children** elements.
- The top element of tree is called the **root** of the tree.



# Formal Definition

- Formally, we define a **tree**  $T$  as a set of nodes storing elements such that the nodes have a **parent-child** relationship that satisfies the following properties:
  - If  $T$  is nonempty, 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  $w$ .
- Note that according to our definition, a tree can be empty, meaning that it does not have any nodes. This convention also allows us to define a tree recursively.
- Two nodes that are children of the same parent are **siblings**.
- A node  $v$  is **external** if  $v$  has no children. A node  $v$  is **internal** if it has one or more children. External nodes are also known as **leaves**.
- A node  $u$  is an **ancestor** of a node  $v$  if  $u = v$  or  $u$  is an ancestor of the parent of  $v$ .
- Conversely, we say that a node  $v$  is a **descendant** of a node  $u$  if  $u$  is an ancestor of  $v$ .



# The Tree ADT

- We define a tree ADT using the concept of a ***position*** as an abstraction for a node of a tree.
- An element is stored at each position, and positions satisfy parent-child relationships that define the tree structure. A position object for a tree supports the method:
  - `p.element()`: Return the element stored at position p.
- The tree ADT then supports the following ***accessor*** methods:
  - `T.root()`: Return the position of the root of tree T, or None if T is empty.
  - `T.is_root(p)`: Return True if position p is the root of Tree T.
  - `T.parent(p)`: Return the position of the parent of position p, or None if p is the root of T.
  - `T.num_children(p)`: Return the number of children of position p.
  - `T.children(p)`: Generate an iteration of the children of position p.
  - `T.is_leaf(p)`: Return True if position p does not have any children.
  - `len(T)`: Return the number of positions (and hence elements) that are contained in tree T.
  - `T.is_empty()`: Return True if tree T does not contain any positions.
  - `T.positions()`: Generate an iteration of all positions of tree T.
  - `iter(T)`: Generate an iteration of all elements stored within tree T.

# Duck Typing

- Duck Typing is a way of programming in which an object passed into a function or method supports all method signatures and attributes expected of that object at run time.
- The object's type itself is not important. Rather, the object should support all methods/attributes called on it. For this reason, duck typing is sometimes seen as "a way of thinking rather than a type system".
- In duck typing, we don't declare the argument types in function prototypes or methods. This implies that compilers can't do type checking. What really matters is if the object has the particular methods/attributes at run time.
- It originates from an old saying: *"If it walks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck"*. Even a non-duck entity that behaves like a duck can be considered a duck because emphasis is on behaviour.
- By analogy, for computing languages, the type of an object is not important so long as it behaves as expected.



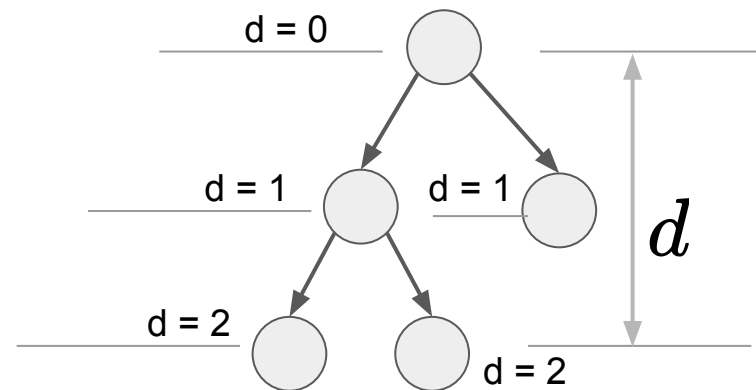
# Abstract Base Class for Tree Data structure

- We follow duck typing by providing abstract base class and several concrete sub-classes inheriting from this base class.
- The base class consists of
  - A nested position class with abstract methods.
  - Several abstract methods
  - Few concrete methods:
    - `is_root()`
    - `is_leaf()`
    - `is_empty()`
- No direct instance is created for the abstract base class

```
1 class Tree:
2     '''Abstract Base Class representing a tree structure'''
3
4     # ----- Nested Position Class -----
5     class Position:
6         '''An abstraction representing the location of a single element'''
7
8         def element(self):
9             '''return the element stored at this position'''
10            raise NotImplementedError('must be implemented by subclass')
11
12        def __eq__(self, other):
13            '''Return True if other Position represents the same location'''
14            raise NotImplementedError('must be implemented by subclass')
15
16        def __ne__(self, other):
17            '''Return True if the other does not represent the same location.'''
18            return not(self == other)
19
20        #----- Abstract Methods that concrete subclass must support ----
21
22        def root(self):
23            '''Returns Position representing the tree's root (or None if Empty)'''
24            raise NotImplementedError('must be implemented by subclass')
25
26        def parent(self, p):
27            '''Returns Position representing p's parent (or None if Empty)'''
28            raise NotImplementedError('must be implemented by subclass')
29
30        def num_children(self, p):
31            '''Return the number of children that Position p has.'''
32            raise NotImplementedError('must be implemented by subclass')
33
34        def children(self, p):
35            '''Return the number of children that Position p has.'''
36            raise NotImplementedError('must be implemented by subclass')
37
38        def __len__(self):
39            '''Return the total number of elements in the tree.'''
40            raise NotImplementedError('must be implemented by subclass')
41
42        # ----- concrete methods implemented in this class -----
43
44        def is_root(self, p):
45            '''Return True if Position p represents the root of the tree.'''
46            return self.root() == p
47
48        def is_leaf(self, p):
49            '''Return True if Position p does not have any children.'''
50            return self.num_children(p) == 0
51
52        def is_empty(self):
53            '''Return True if the tree is empty.'''
54            return len(self) == 0
55
```

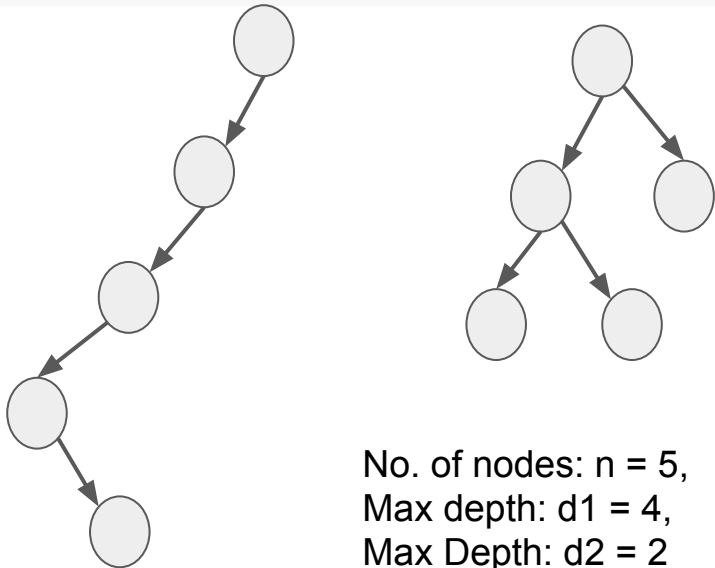
## Computing Depth

- Let  $p$  be the position of a node of a tree  $T$ .
- The **depth** of  $p$  is the number of ancestors of  $p$ , excluding  $p$  itself.
- The depth of the root of  $T$  is 0 as the root has no ancestors.
- The depth of  $p$  can also be recursively defined as follows:
  - If  $p$  is the root, then the depth of  $p$  is 0.
  - Otherwise, the depth of  $p$  is one plus the depth of the parent of  $p$ .



For each non-root node,  
we add 1 to the depth of  
its parent.

```
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))
```

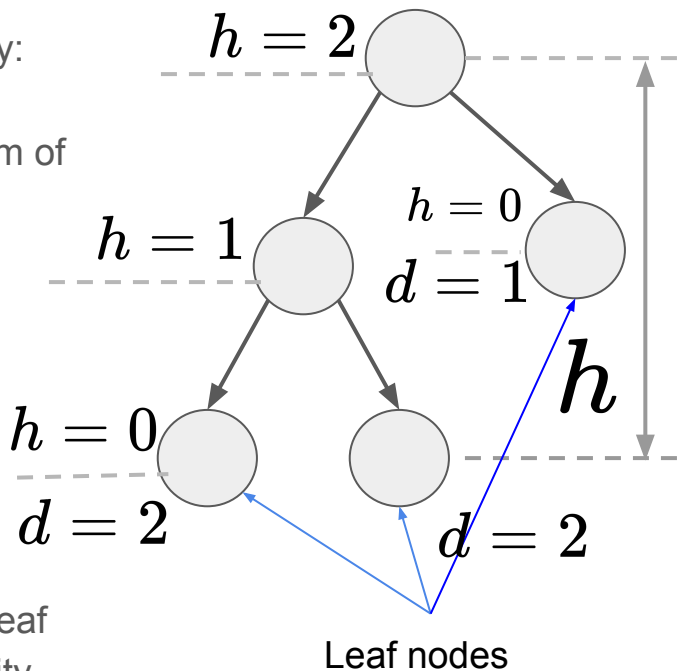


No. of nodes:  $n = 5$ ,  
 Max depth:  $d_1 = 4$ ,  
 Max Depth:  $d_2 = 2$

- The running time of  **$T.\text{depth}(p)$**  for position  $p$  is  $O(d_p + 1)$  where  $d_p$  denotes the depth of  $p$  in the tree  $T$ , because the algorithm performs a constant-time recursive step for each ancestor of  $p$ .
- **$T.\text{depth}(p)$**  runs in  $O(n)$  worst-case time, where  $n$  is the total number of positions of  $T$ , because a position of  $T$  may have depth  $n-1$  if all nodes form a single branch.

# Computing Height

- The height of a position  $p$  in a tree  $T$  is also defined recursively:
  - If  $p$  is a leaf, then the height of  $p$  is 0.
  - Otherwise, the height of  $p$  is one more than the maximum of the heights of  $p$ 's children.
- The **height** of a nonempty tree  $T$  is the height of the root of  $T$ .
- **Proposition:** *The height of a nonempty tree  $T$  is equal to the maximum of the depths of its leaf positions.*
- We provide two implementations for computing height:
  - `_height1(p)` that calls algorithm `depth(p)` on each leaf position of  $T$  leading to  $O(n^2)$  worst-case time complexity.
  - `_height2(p)` that calls itself recursively to provide linear or  $O(n)$  time-complexity



- There are  $n$  number of positions in the tree.
- Let us assume there are  $L$  leaf positions.
- Computation time:

$$O(n + \sum_{p \in L} (d_p + 1))$$

- Each call to `depth()` function has a constant time  $O(d_p + 1)$  which in the worst case can be  $(n-1)$  or  $O(n)$  time-complexity

- So,  $\sum_{p \in L} (d_p + 1) \propto n^2$
- So the worst-case time complexity of `_height1()` function is

$$O(n + n^2) \approx O(n^2)$$

```
def _height1(self):
    ...
    Return the height of the tree
    works but O(n^2) worst-case time
    ...
    return(max([self.depth(p)
                for p in self.positions() if self.is_leaf(p)]))
```

$\sim O(n^2)$

```
depths = []
for p in range(n): (executed n times)
    if is_leaf(p):
        d = depth(p) (executed L times)
        depths.append(d)
H = max(depths)
```

Case 1:  $n = 5, L = 3$

$$C_1 = 1$$

depth(2)

$$C_3 = 1$$

$$C_2 = 1 + 2 = 3$$

depth(4)

$$C_4 = 1 + 3 = 4$$

$$C_5 = 1 + 3 = 4$$

depth(5)

$$T = \sum_{i=1}^5 C_i = 13$$

$$T \leq 5^2; T \sim O(n^2)$$

As  $n$  increases, the upper bound on the computation time will be quadratic wrt  $n$ .

Case II:  $n = 5, L = 1$

$$C_1 = 1$$

$$C_2 = 1$$

$$C_3 = 1$$

$$C_4 = 1$$

depth(5)

$$C_5 = 1 + 5 = 6$$

$$T = \sum_{i=1}^5 C_i = 10$$

- `_height2()` provides  $O(n)$  worst-case time performance.
- It progresses in top-down fashion.
- `children(p)` runs in  $O(c_p + 1)$  time, where  $c_p$  denotes the number of children of  $p$ .
- `_height2(p)` spends  $O(c_p + 1)$  time at each position  $p$  to compute the maximum. So its overall running time is

$$\begin{aligned}
 O\left(\sum_p (c_p + 1)\right) &= O\left(n + \sum_p c_p\right) \\
 &= O(n + n - 1) \\
 &\sim O(n)
 \end{aligned}$$

```
def height(self, p = None):
    """
    Return the height of 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
```

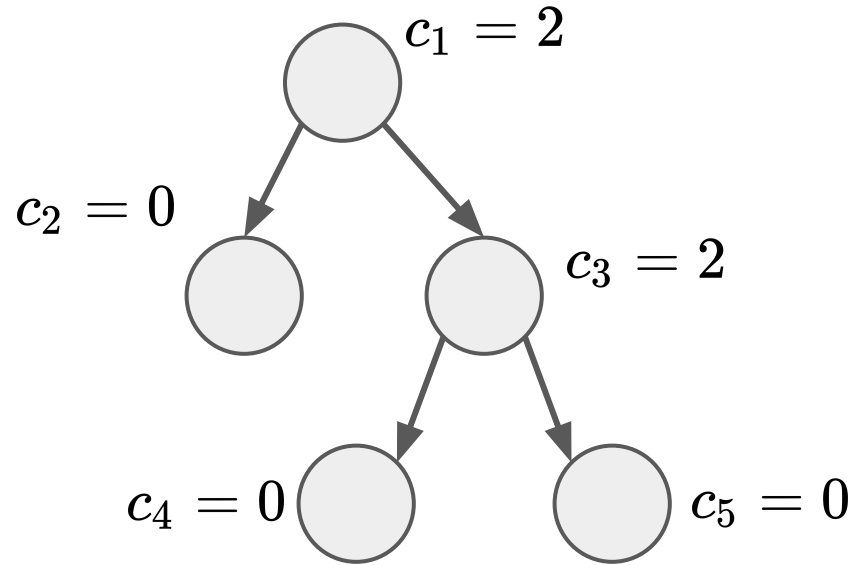
```
def _height2(self, p):
    """
    Return the height of the tree
    time is linear in size of sub-tree
    """
    if self.is_leaf(p):
        return 0
    else:
        return 1 + max(self._height2(c) for c in self.children(p))
```

**Proposition:** Let  $T$  be a tree with  $n$  positions, and let  $c_p$  denote the number of children of a position  $p$  of  $T$ . Then, summing over the positions of  $T$ ,

$$\sum_p c_p = n - 1$$

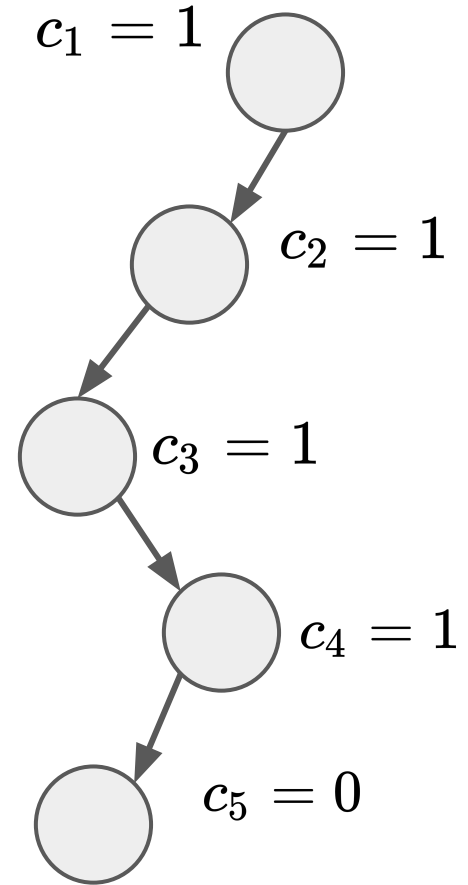
**Justification:** Each position of  $T$ , with the exception of the root, is a child of another position, and thus contributes one unit to the above sum.

$$n = 5$$



Total number of Children In  
the tree:

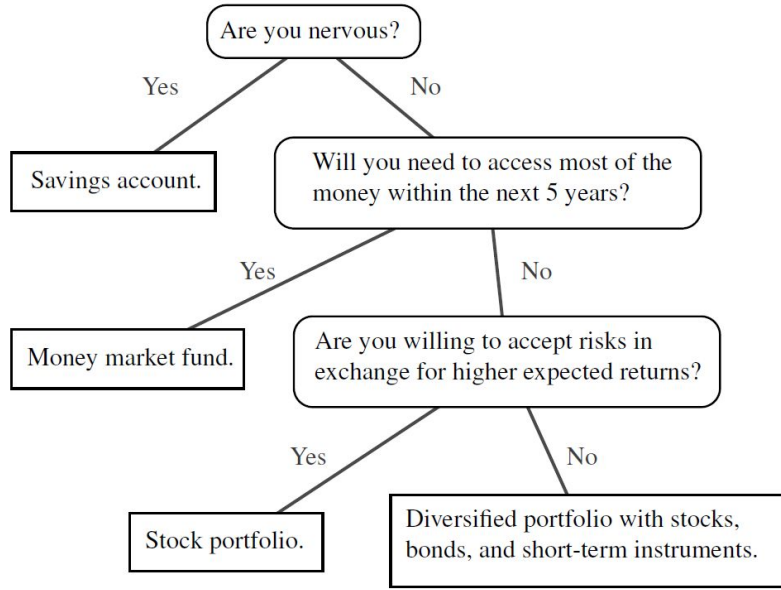
$$\sum_p c_p = 4$$



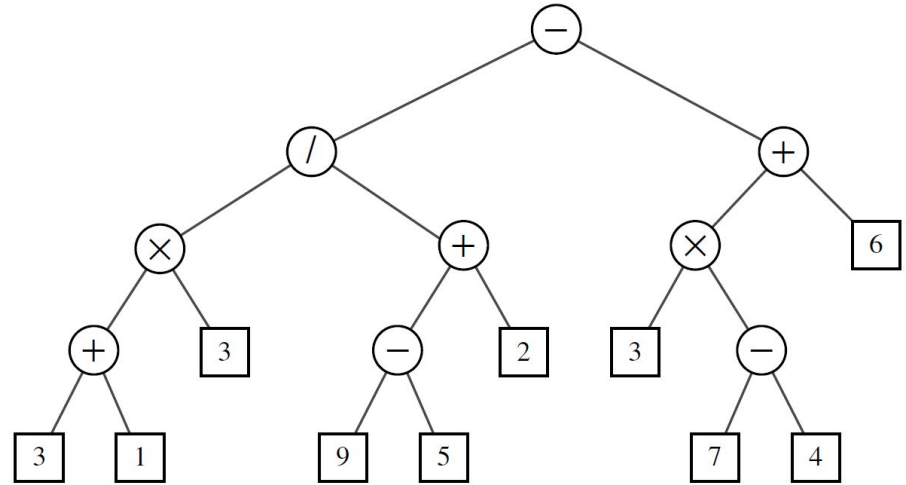


# Binary Trees

- A **binary tree** is an ordered tree with the following properties:
  - Every node has at most two children.
  - Each child node is labeled as being either a left child or a right child.
  - A left child precedes a right child in the order of children of a node.
- The subtree rooted at a left or right child of an internal node  $v$  is called a **left subtree** or **right subtree**, respectively, of  $v$ .
- A binary tree is **proper** if each node has either zero or two children. Such trees are also referred to as **full** binary trees.
- Thus, in a proper binary tree, every internal node has exactly two children.
- A binary tree that is not proper is **improper**.
- Examples:
  - **Decision tree** is a proper binary tree obtained by deciding outcomes based on a series of yes-or-no questions.
  - Arithmetic expressions can be represented by a binary tree where the leaves are associated with variables and constants and the internal nodes are the operators.



A decision tree providing investment advice.



Tree representing arithmetic expression:  
 $((((3 + 1) \times 3) / ((9 - 5) + 2)) - ((3 \times (7 - 4)) + 6)).$

## Examples of Binary Tree

- **A recursive binary tree definition:** A binary tree is either empty or consists of:
  - A node  $r$ , called the root of  $T$ , that stores an element
  - A binary tree (possibly empty), called the left subtree of  $T$
  - A binary tree (possibly empty), called the right subtree of  $T$
- The Binary Tree ADT: As an abstract data type, a binary tree is a specialization of a tree that supports three additional accessor methods:
  - **$T.\text{left}(p)$**  : Return the position that represents the left child of  $p$ , or `None` if  $p$  has no left child.
  - **$T.\text{right}(p)$**  : Return the position that represents the right child of  $p$ , or `None` if  $p$  has no right child.
  - **$T.\text{sibling}(p)$**  : Return the position that represents the sibling of  $p$ , or `None` if  $p$  has no sibling.

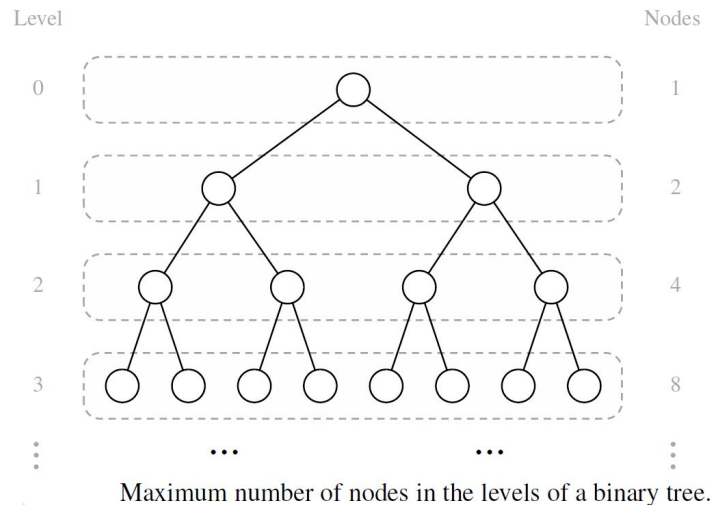
# The Binary Tree Abstract Base Class Implementation

- **BinaryTree** class still remains *abstract*.
- It inherits all properties from the **Tree** class (e.g., **parent()**, **is\_leaf()**, **root()** etc.)
- Provides two declarations for two new abstract methods:
  - **left(p)** - position of p's left child
  - **right(p)** - position of p's right child
- Provides concrete implementation of two methods:
  - **sibling(p)** - other child of p's parent
  - **children(p)** - generator for the ordered children of a node.

```
1 class BinaryTree(Tree):
2     '''Abstract base class representing a binary tree structure.'''
3
4     # ----- additional abstract methods -----
5     def left(self, p):
6         '''
7         Return a Position representing p's left child.
8         Return None if p does not have a left child.
9         '''
10        raise NotImplementedError('Must be implemented by subclass')
11
12    def right(self, p):
13        '''
14        Return a Position representing p's right child.
15        Return None if p does not have a right child
16        '''
17        raise NotImplementedError('Must be implemented by subclass')
18
19    # ----- concrete methods implemented in this class -----
20
21    def sibling(self, p):
22        '''Return a Position representing p's sibling (or None if no sibling).'''
23        parent = self.parent(p)
24        if parent is None:          # p must be the root
25            return None            # root has no sibling
26        else:
27            if p == self.left(parent):
28                return self.right(parent)    # possibly None
29            else:
30                return self.left(parent)     # possibly None
31
32
33    def children(self, p):
34        '''Generate an iteration of Positions representing p's children'''
35        if self.left(p) is not None:
36            yield self.left(p)
37        if self.right(p) is not None:
38            yield self.right(p)
```

# Properties of Binary Trees

- We denote the set of all nodes of a tree  $T$  at the same depth  $d$  as **level**  $d$  of  $T$ .
- In general, level  $d$  has at most  $2^d$  nodes.
- The maximum number of nodes on the levels of a binary tree grows exponentially as we go down the tree.



**Proposition 8.8:** *Let  $T$  be a nonempty binary tree, and let  $n$ ,  $n_E$ ,  $n_I$  and  $h$  denote the number of nodes, number of external nodes, number of internal nodes, and height of  $T$ , respectively. Then  $T$  has the following properties:*

1.  $h + 1 \leq n \leq 2^{h+1} - 1$
2.  $1 \leq n_E \leq 2^h$
3.  $h \leq n_I \leq 2^h - 1$
4.  $\log(n + 1) - 1 \leq h \leq n - 1$

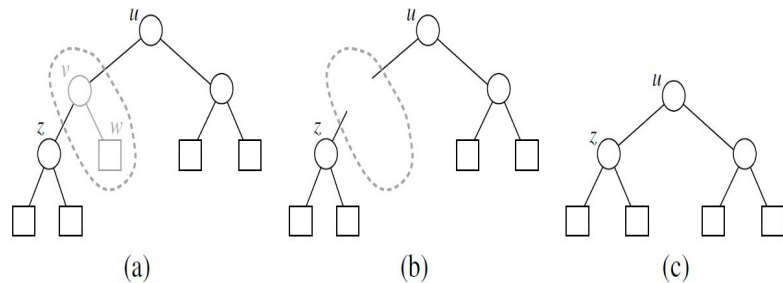
*Also, if  $T$  is proper, then  $T$  has the following properties:*

1.  $2h + 1 \leq n \leq 2^{h+1} - 1$
2.  $h + 1 \leq n_E \leq 2^h$
3.  $h \leq n_I \leq 2^h - 1$
4.  $\log(n + 1) - 1 \leq h \leq (n - 1)/2$

**Proposition 8.9:** *In a nonempty proper binary tree  $T$ , with  $n_E$  external nodes and  $n_I$  internal nodes, we have  $n_E = n_I + 1$ .*

Justification: We make two piles - one for external nodes and another for internal nodes.

Case 1: If  $T$  has one node  $v$  then  $v$  is put into external pile and none for internal node.



Case 2: If  $T$  has more than one node, we remove a pair of (one external node and one internal node) at one time. In the end only one node will be left which will be external.

Note that the above relationship does not hold, in general, for improper binary trees and non-binary trees,

# Implementing Trees

- Linked Structure for Binary trees
- Array-based Representation for Binary Trees
- Linked Structure for General Trees



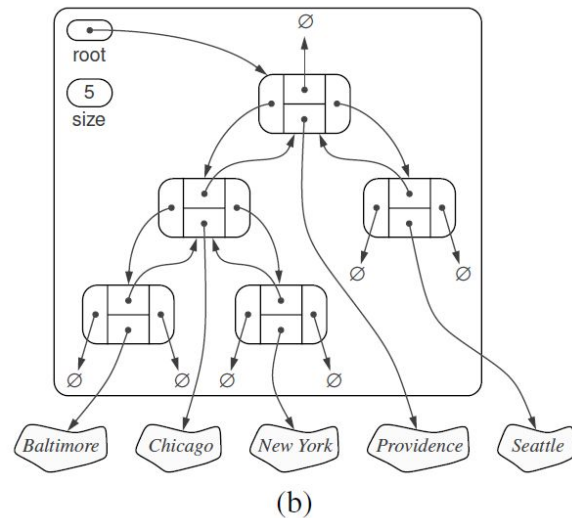
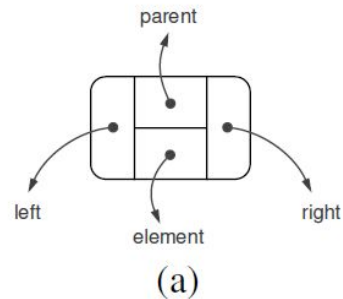
# Linked Structure for Binary Trees

Each node maintains the references to the following:

- The element stored at a position p.
- Left child node
- Right child node
- Parent node.

The tree T maintains two variables:

- Variable for storing reference to the root node.
- Variable `size` that represents the total number of nodes of T.



# Python Implementation of a Linked Binary Tree Structure

- A sub-class of `BinaryTree` class.
- It defines two nested classes:
  - A nonpublic `_Node` class to represent node.
  - A public `Position` class that wraps the node - inherited from `Tree.Position` class
- It provides following two utility methods:
  - `_validity` utility for robustly checking the validity of a given position while unwrapping it.
  - `_make_position` utility for wrapping a node as a position to return to a caller.
- Operations for updating Linked Binary Tree
  - `T._add_root(e)`
  - `T._add_left(e)`
  - `T._add_right(e)`
  - `T._replace(p, e)`
  - `T._delete(p)`
  - `T._attach(p, T1, T2)`

$\sim O(1)$   
Worst-case time

```
3 # Linked Binary tree
4 class LinkedBinaryTree(BinaryTree):
5     '''Linked representation of a binary tree structure.'''
6
7     class _Node:
8         __slots__ = '_element', '_parent', '_left', '_right'
9         def __init__(self, element, parent=None, left=None, right=None):
10             self._element = element
11             self._parent = parent
12             self._left = left
13             self._right = right
14
15     class Position(BinaryTree.Position):
16         '''An abstraction representing the location of a single element.'''
17
18         def __init__(self, container, node):
19             '''Constructor should not be invoked by user.'''
20             self._container = container
21             self._node = node
22
23         def element(self):
24             '''Return the element stored at this position.'''
25             return self._node._element
26
27         def __eq__(self, other):
28             '''Return True if other is a position representing the same location.'''
29             return type(other) is type(self) and other._node is self._node
30
31     # ----- hidden utility functions for LinkedBinary Tree -----
32     def _validate(self, p):
33         '''Return associated node, if position is valid.'''
34         if not isinstance(p, self.Position):
35             raise TypeError('p must be proper Position type')
36         if p._container is not self:
37             raise ValueError('p does not belong to this container')
38         if p._node._parent is p._node: # convention for deprecated nodes
39             raise ValueError('p is no longer valid')
40         return p._node
41
42     def _make_position(self, node):
43         '''Return Position instance for a given node (or None if no node)'''
44         return self.Position(self, node) if node is not None else None
45
```

```

#----- 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 tree is empty).'''
    return self._make_position(self._root)

def parent(self, p):
    '''return the position 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 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 P's right child (or None if no right child).'''
    node = self._validate(p)
    return self._make_position(node._right)

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:
        count += 1
    if node._right is not None:
        count += 1
    return count

```

```

# ----- NonPublic tree update Methods -----
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)

def _add_left(self, p, e):
    '''
    Create a new left child for Position P, storing element e.
    Return the position of a 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)

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)

def _replace(self, p, e):
    '''
    replace the element at position P with e and return the old element.'''
    node = self._validate(p)
    old = node._element
    node._element = e
    return old

```

```

148 def _delete(self, p):
149     '''
150     Delete the node at Position p, and replace it with its child, if any.
151     Return the element that had been stored at Position p.
152     Raise ValueError if Position p is invalid or p has two children.
153     '''
154     node = self._validate(p)
155     if self.num_children(p) == 2: raise ValueError('p has two children')
156     child = node._left if node._left else node._right
157     if child is not None:
158         child._parent = node._parent # child's grandparent becomes parent
159     if node is self._root:
160         self._root = child # child becomes root if its parent is deleted
161     else:
162         parent = node._parent
163         if node is parent._left:
164             parent._left = child
165         else:
166             parent._right = child
167     self._size -= 1
168     node._parent = node # convention for deprecated node
169     return node._element
170
171 def _attach(self, p, t1, t2):
172     '''Attach trees t1 and t2 as left and right subtrees of external p.'''
173
174     node = self._validate(p)
175
176     if not self.is_leaf(p): raise ValueError('position must be leaf')
177     if not type(self) is type(t1) is type(t2): # all 3 tree must be same type
178         raise TypeError('Tree types must match')
179     self._size += len(t1) + len(t2)
180
181     if not t1.is_empty(): # attach t1 as left subtree of node
182         t1._root._parent = node
183         node._left = t1._root
184         t1._root = None # set t1 instance to empty
185         t1._size = 0
186     if not t2.is_empty(): # attach t2 as right subtree of node
187         t2._root._parent = node
188         node._right = t2._root
189         t2._root = None # set t2 instance to empty
190         t2._size = 0
191

```

```

86 def _print_children(self, p, arr):
87     if p is not None:
88         arr += str(p.element())+' ': '
89         if self.num_children(p) > 0:
90             for c in self.children(p):
91                 arr += str(c.element()) + '\t'
92             arr += '\n'
93             for c in self.children(p):
94                 arr = self._print_children(c, arr)
95             arr += '\n'
96     return arr
97
98 def __str__(self):
99     ''' Provide a string representation of the tree'''
100
101     start = self.root()
102     arr = ''
103     arr = self._print_children(start, arr)
104     return arr
105

```



Above function prints a binary tree.

Can you make it better by providing a more intuitive output?

```

192 #####
193
194 P = LinkedBinaryTree()
195 P._add_root('Providence')
196 P._add_left(P.root(), 'Chicago')
197 P._add_right(P.root(), 'Seattle')
198 P._add_left(P.left(P.root()), 'Baltimore')
199 P._add_right(P.left(P.root()), 'New York')
200 print(P)
201 print('\n-----\n')
202
203 Q = LinkedBinaryTree()
204 Q._add_root('-')
205 Q._add_left(Q.root(), '/')
206 Q._add_right(Q.root(), '+')
207 Q._add_left(Q.left(Q.root()), 'X')
208 Q._add_right(Q.left(Q.root()), '+')
209
210 Q._add_left(Q.right(Q.root()), 'X')
211 Q._add_right(Q.right(Q.root()), 6)
212
213 Q._add_left(Q.left(Q.left(Q.root()))), '+')
214 Q._add_right(Q.left(Q.left(Q.root()))), '3')
215 Q._add_left(Q.left(Q.left(Q.left(Q.root())))), '3')
216 Q._add_right(Q.left(Q.left(Q.left(Q.root())))), '1')
217
218 Q._add_left(Q.right(Q.left(Q.root()))), '-')
219 Q._add_right(Q.right(Q.left(Q.root()))), '2')
220 Q._add_left(Q.left(Q.right(Q.left(Q.root())))), '9')
221 Q._add_right(Q.left(Q.right(Q.left(Q.root())))), '5')
222
223 Q._add_left(Q.left(Q.right(Q.root()))), '3')
224 Q._add_right(Q.left(Q.right(Q.root()))), '-')
225
226 Q._add_left(Q.right(Q.left(Q.right(Q.root())))), '7')
227 Q._add_right(Q.right(Q.left(Q.right(Q.root())))), '4')
228
229 print(Q)

```

```

➡ Providence: Chicago      Seattle
Chicago: Baltimore        New York
Baltimore:
New York:

Seattle:

```

```

-----
-: /      +
/: X      +
X: +      3
+: 3      1
3:
1:

3:

+: -      2
-: 9      5
9:
5:

2:

+: X      6
X: 3      -
3:
-: 7      4
7:
4:

6:

```

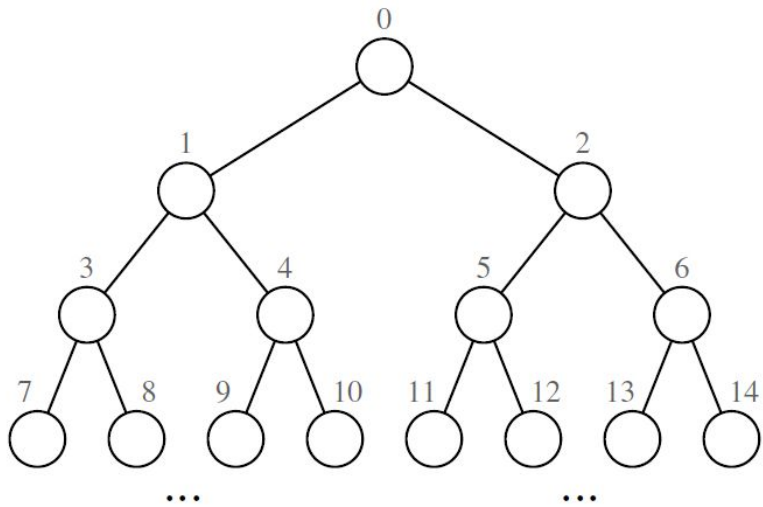


## Performance of the Linked Binary Tree Implementation

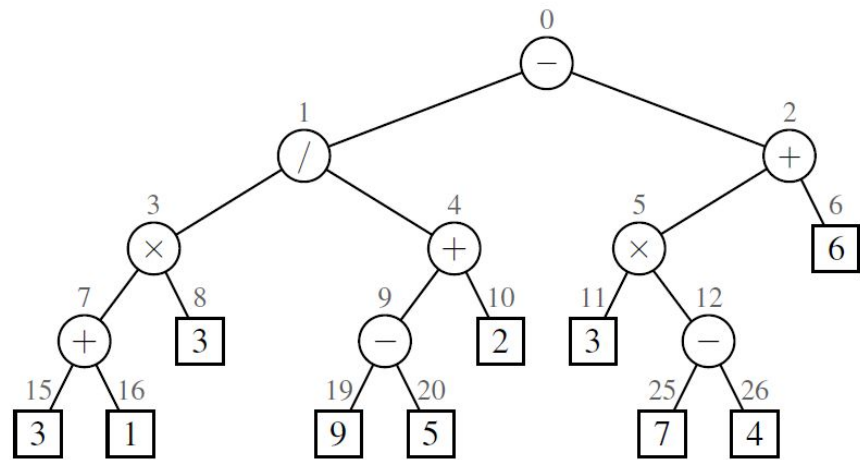
Operation	Running Time
len, is_empty	$O(1)$
root, parent, left, right, sibling, children, num_children	$O(1)$
is_root, is_leaf	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$
add_root, add_left, add_right, replace, delete, attach	$O(1)$

## Array-Based Representation of a Binary Tree

- An alternative representation of a binary tree  $T$  is based on a way of numbering the positions of  $T$ .
- 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  $f(p) = 2 f(q)+1$ .
  - If  $p$  is the right child of position  $q$ , then  $f(p) = 2 f(q)+2$ .
- The numbering function  $f$  is known as a **level numbering** of the positions in a binary tree  $T$ .
  - it numbers the positions on each level of  $T$  in increasing order from left to right.
- The level numbering function  $f$  suggests a representation of a binary tree  $T$  by means of an array-based structure  $A$  (such as a Python list), with the element at position  $p$  of  $T$  stored at index  $f(p)$  of the array.



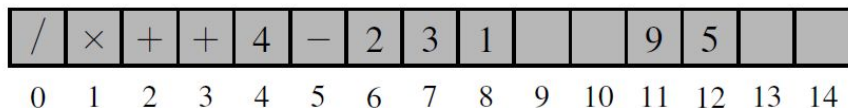
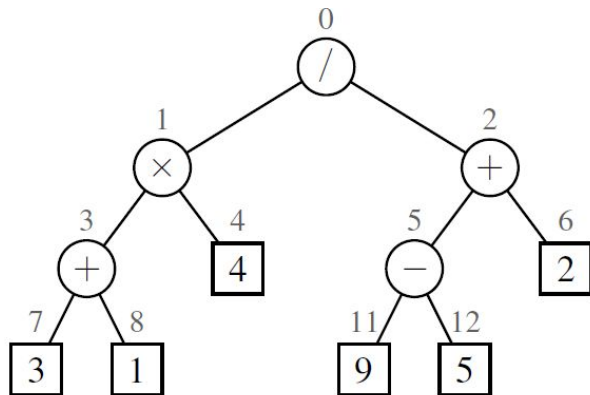
Binary Tree level numbering scheme



An Example



- The level numbering function  $f$  suggests a representation of a binary tree  $T$  by means of an array-based structure  $A$  (such as a Python list), with the element at position  $p$  of  $T$  stored at index  $f(p)$  of the array.



Advantage:

- The position  $p$  can be represented by the single integer  $f(p)$ .
- The position-based methods such as root, parent, left, right can be implemented using simple arithmetic operation on  $f(p)$ .
- Left child of  $p$  has an index  $2f(p)+1$ , the right child has index  $2f(p)+2$  and the parent of  $p$  has index  $\lfloor (f(p) - 1)/2 \rfloor$ .

## Space Usage of Array-based Trees

- Let  $n$  be the number of nodes of  $T$ .
- Let  $f\_M$  be the maximum value of  $f(p)$  over all nodes of  $T$ .
- Array  $A$  requires length  $N = 1 - f\_M$  with array elements ranging from  $A[0]$  to  $A[f\_M]$ .
- The array  $A$  may have a number of empty cells that do not refer to existing nodes of  $T$ .
- In the worst case,  $N = 2^n - 1$
- Worst case space requirement is  $O(2^n)$
- For general binary trees, the exponential worst-case space requirement of this representation is prohibitive.

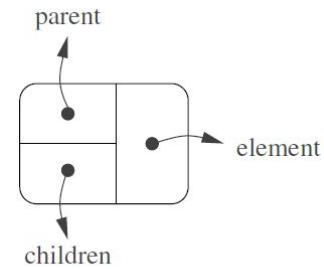
## Disadvantages of Array-based Tree implementation

- Array-based trees have ***exponential worst-case space requirement*** making them less desirable.
- *Some update operations for trees cannot be efficiently supported* in array representation.
  - Example: deleting a node and promoting its child takes  $O(n)$  time because it is not just the child that moves locations within the array, but all descendants of that child.

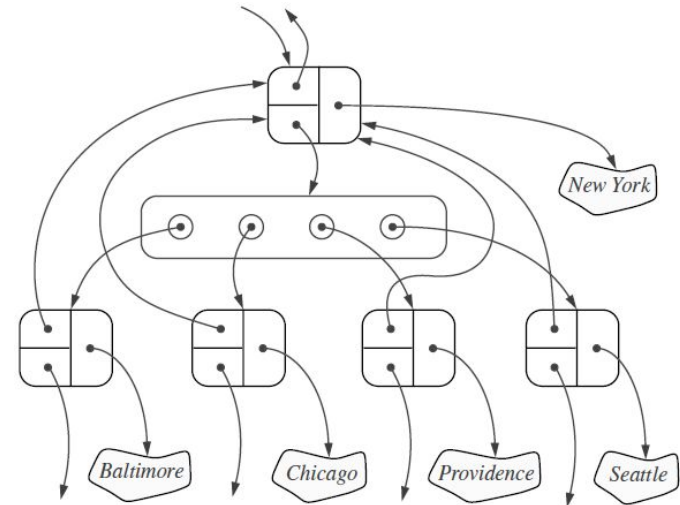
# Linked Structure for General Trees

- For a general tree, there is no a priori limit on the number of children that a node may have.
- One way to realize a general tree  $T$  as a linked structure is to have each node store a single container of references to its children.
- For example a children field of a node can be a Python list of references to the children of the node

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	$O(n)$



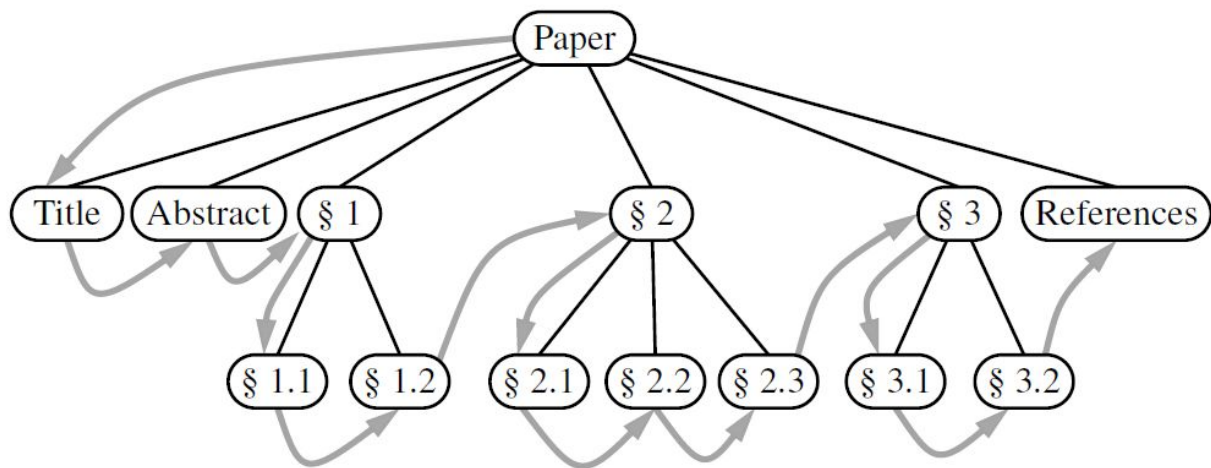
(a)



(b)

# Tree Traversal Algorithms

- A ***traversal*** of a tree  $T$  is a systematic way of accessing, or “visiting”, all the positions of  $T$ .
- In a ***preorder traversal*** of a tree  $T$ , *the root of  $T$  is visited first* and then the subtrees rooted at its children are traversed recursively.
- If the tree is ordered, then the subtrees are traversed according to the order of the children.
- In a ***postorder traversal*** of a tree  $T$ , *the subtrees rooted at the children of the root are first traversed*, and then the root is visited.



Preorder traversal of an ordered tree, where the children of each position are ordered from left to right.

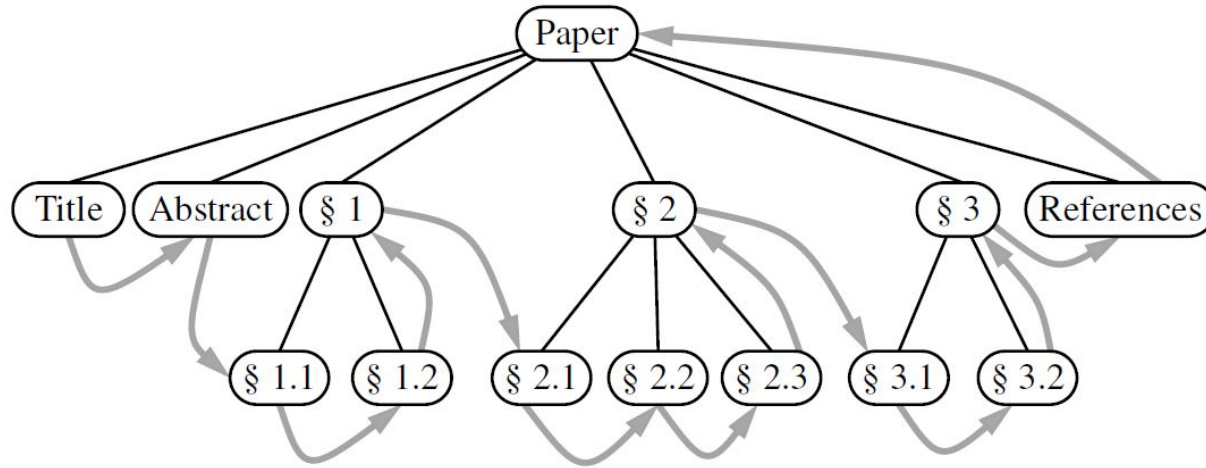
**Algorithm** preorder( $T, p$ ):

perform the “visit” action for position  $p$

**for** each child  $c$  in  $T.children(p)$  **do**

preorder( $T, c$ )

{recursively traverse the subtree rooted at  $c$ }



Postorder traversal of the ordered tree

**Algorithm** postorder( $T$ ,  $p$ ):

**for** each child  $c$  in  $T$ .children( $p$ ) **do**

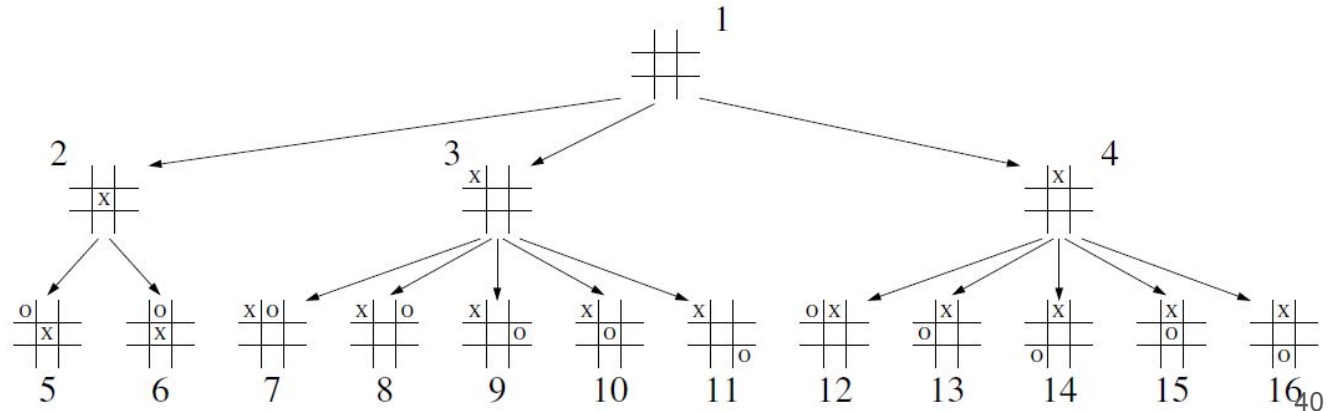
    postorder( $T$ ,  $c$ )                      {recursively traverse the subtree rooted at  $c$ }

  perform the “visit” action for position  $p$

# Breadth-First Tree Traversal

- **Breadth-first traversal:** traverse a tree so that we visit all the positions at depth  $d$  before we visit the positions at depth  $d+1$ .
- A breadth-first traversal is a common approach used in software for playing games.
- A game tree represents the possible choices of moves that might be made by a player (or computer) during a game, with the root of the tree being the initial configuration for the game.

Partial game tree for Tic-Tac-Toe showing the numbers as orders in which the positions are visited in a breadth-first traversal.





**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  $p$

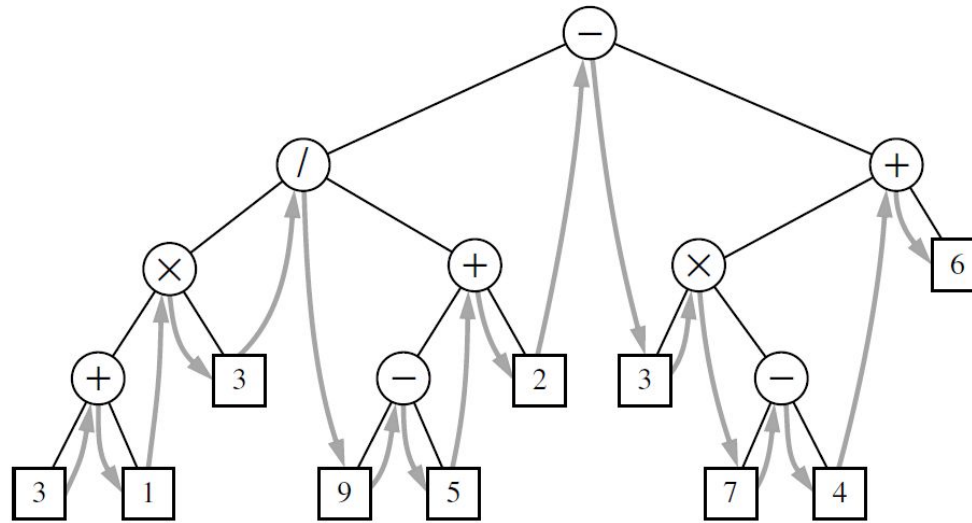
**for** each child  $c$  in  $T.children(p)$  **do**

Q.enqueue(c)     {add p's children to the end of the queue for later visits}

- The process is not recursive.
- A queue is used to produce a FIFO (i.e., first-in first-out) semantics for the order in which we visit nodes.
- The overall running time is  $O(n)$ , due to the  $n$  calls to enqueue and  $n$  calls to dequeue.

# Inorder Traversal of a Binary Tree

- The standard preorder, postorder, and breadth-first traversals described for general trees, can be directly applied to binary trees.
- ***Inorder traversal*** is applicable only to binary trees.
- During an inorder traversal, we visit a position between the recursive traversals of its left and right subtrees.
- The inorder traversal of a binary tree T can be informally viewed as visiting the nodes of T ***“from left to right.”***
- For every position p, the inorder traversal visits p after all the positions in the left subtree of p and before all the positions in the right subtree of p.



Inorder traversal of a binary tree.

**Algorithm** inorder(p):

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

    inorder(lc)

    {recursively traverse the left subtree of p}

perform the “visit” action for position p

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

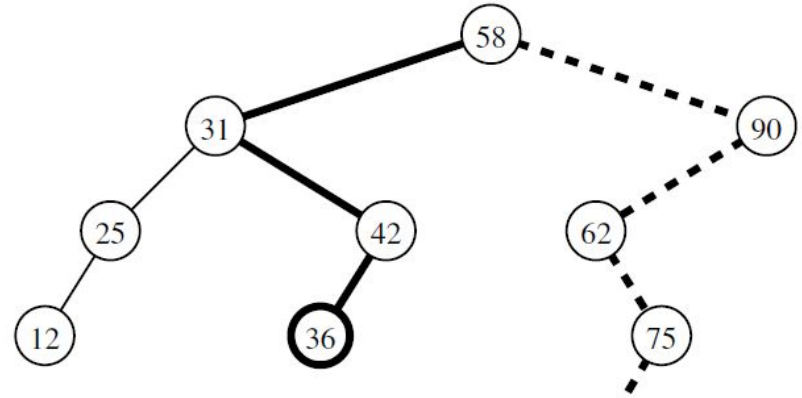
    inorder(rc)

    {recursively traverse the right subtree of p}

# Binary Search Trees

- An important application of the inorder traversal algorithm arises when we store an ordered sequence of elements in a binary tree - a ***binary search tree***.
- Consider the following example:  
Let  $S$  be a set whose unique elements have an order relation, e.g,  $S$  could be a set of integers.  
A binary search tree for  $S$  is a binary tree  $T$  such that, for each position  $p$  of  $T$ :
  - Position  $p$  stores an element of  $S$ , denoted as  $e(p)$ .
  - Elements stored in the left subtree of  $p$  (if any) are less than  $e(p)$ .
  - Elements stored in the right subtree of  $p$  (if any) are greater than  $e(p)$ .
- The above properties assure that an inorder traversal of a binary search tree  $T$  visits the elements in nondecreasing order.

- At each internal position  $p$  encountered, we compare our **search value  $v$**  with the element  $e(p)$  stored at  $p$ .
- If  $v < e(p)$ , then the search continues in the left subtree of  $p$ .
- If  $v = e(p)$ , then the search terminates successfully.
- If  $v > e(p)$ , then the search continues in the right subtree of  $p$ .
- Finally, if we reach an empty subtree, the search terminates unsuccessfully.
- In other words, a binary search tree can be viewed as a ***binary decision tree***.
- The running time of searching in a binary search tree  $T$  is proportional to the height of  $T$ .
- Recall from [Proposition 8.8](#) that the height of a binary tree with  $n$  nodes can be as small as  $\log(n+1)-1$  or as large as  $n-1$ .



A binary search tree storing integers. The solid path is traversed when searching (successfully) for 36. The dashed path is traversed when searching (unsuccessfully) for 70.



Thus, binary search trees are most efficient when they have small height.

# Implementing Tree Traversals in Python

We define the following three public methods for tree traversal:

- `T.preorder()`
- `T.postorder()`
- `T.breadthfirst()`
- `T.positions()` # uses preorder traversal by default.

It is possible to access the elements of the tree as shown below:

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

```
1 class Tree:  
2     '''Abstract Base Class representing a tree structure'''  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23  
24  
25  
26  
27  
28  
29  
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
560  
561  
562  
563  
564  
565  
566  
567  
568  
569  
570  
571  
572  
573  
574  
575  
576  
577  
578  
579  
580  
581  
582  
583  
584  
585  
586  
587  
588  
589  
590  
591  
592  
593  
594  
595  
596  
597  
598  
599  
600  
601  
602  
603  
604  
605  
606  
607  
608  
609  
610  
611  
612  
613  
614  
615  
616  
617  
618  
619  
620  
621  
622  
623  
624  
625  
626  
627  
628  
629  
630  
631  
632  
633  
634  
635  
636  
637  
638  
639  
640  
641  
642  
643  
644  
645  
646  
647  
648  
649  
650  
651  
652  
653  
654  
655  
656  
657  
658  
659  
660  
661  
662  
663  
664  
665  
666  
667  
668  
669  
670  
671  
672  
673  
674  
675  
676  
677  
678  
679  
680  
681  
682  
683  
684  
685  
686  
687  
688  
689  
690  
691  
692  
693  
694  
695  
696  
697  
698  
699  
700  
701  
702  
703  
704  
705  
706  
707  
708  
709  
710  
711  
712  
713  
714  
715  
716  
717  
718  
719  
720  
721  
722  
723  
724  
725  
726  
727  
728  
729  
730  
731  
732  
733  
734  
735  
736  
737  
738  
739  
740  
741  
742  
743  
744  
745  
746  
747  
748  
749  
750  
751  
752  
753  
754  
755  
756  
757  
758  
759  
760  
761  
762  
763  
764  
765  
766  
767  
768  
769  
770  
771  
772  
773  
774  
775  
776  
777  
778  
779  
780  
781  
782  
783  
784  
785  
786  
787  
788  
789  
790  
791  
792  
793  
794  
795  
796  
797  
798  
799  
800  
801  
802  
803  
804  
805  
806  
807  
808  
809  
810  
811  
812  
813  
814  
815  
816  
817  
818  
819  
820  
821  
822  
823  
824  
825  
826  
827  
828  
829  
830  
831  
832  
833  
834  
835  
836  
837  
838  
839  
840  
841  
842  
843  
844  
845  
846  
847  
848  
849  
850  
851  
852  
853  
854  
855  
856  
857  
858  
859  
860  
861  
862  
863  
864  
865  
866  
867  
868  
869  
870  
871  
872  
873  
874  
875  
876  
877  
878  
879  
880  
881  
882  
883  
884  
885  
886  
887  
888  
889  
890  
891  
892  
893  
894  
895  
896  
897  
898  
899  
900  
901  
902  
903  
904  
905  
906  
907  
908  
909  
910  
911  
912  
913  
914  
915  
916  
917  
918  
919  
920  
921  
922  
923  
924  
925  
926  
927  
928  
929  
930  
931  
932  
933  
934  
935  
936  
937  
938  
939  
940  
941  
942  
943  
944  
945  
946  
947  
948  
949  
950  
951  
952  
953  
954  
955  
956  
957  
958  
959  
960  
961  
962  
963  
964  
965  
966  
967  
968  
969  
970  
971  
972  
973  
974  
975  
976  
977  
978  
979  
980  
981  
982  
983  
984  
985  
986  
987  
988  
989  
990  
991  
992  
993  
994  
995  
996  
997  
998  
999  
1000  
1001  
1002  
1003  
1004  
1005  
1006  
1007  
1008  
1009  
1010  
1011  
1012  
1013  
1014  
1015  
1016  
1017  
1018  
1019  
1020  
1021  
1022  
1023  
1024  
1025  
1026  
1027  
1028  
1029  
1030  
1031  
1032  
1033  
1034  
1035  
1036  
1037  
1038  
1039  
1040  
1041  
1042  
1043  
1044  
1045  
1046  
1047  
1048  
1049  
1050  
1051  
1052  
1053  
1054  
1055  
1056  
1057  
1058  
1059  
1060  
1061  
1062  
1063  
1064  
1065  
1066  
1067  
1068  
1069  
1070  
1071  
1072  
1073  
1074  
1075  
1076  
1077  
1078  
1079  
1080  
1081  
1082  
1083  
1084  
1085  
1086  
1087  
1088  
1089  
1090  
1091  
1092  
1093  
1094  
1095  
1096  
1097  
1098  
1099  
1100  
1101  
1102  
1103  
1104  
1105  
1106  
1107  
1108  
1109  
1110  
1111  
1112  
1113  
1114  
1115  
1116  
1117  
1118  
1119  
1120  
1121  
1122  
1123  
1124  
1125  
1126  
1127  
1128  
1129  
1130  
1131  
1132  
1133  
1134  
1135  
1136  
1137  
1138  
1139  
1140  
1141  
1142  
1143  
1144  
1145  
1146  
1147  
1148  
1149  
1150  
1151  
1152  
1153  
1154  
1155  
1156  
1157  
1158  
1159  
1160  
1161  
1162  
1163  
1164  
1165  
1166  
1167  
1168  
1169  
1170  
1171  
1172  
1173  
1174  
1175  
1176  
1177  
1178  
1179  
1180  
1181  
1182  
1183  
1184  
1185  
1186  
1187  
1188  
1189  
1190  
1191  
1192  
1193  
1194  
1195  
1196  
1197  
1198  
1199  
1200  
1201  
1202  
1203  
1204  
1205  
1206  
1207  
1208  
1209  
1210  
1211  
1212  
1213  
1214  
1215  
1216  
1217  
1218  
1219  
1220  
1221  
1222  
1223  
1224  
1225  
1226  
1227  
1228  
1229  
1230  
1231  
1232  
1233  
1234  
1235  
1236  
1237  
1238  
1239  
1240  
1241  
1242  
1243  
1244  
1245  
1246  
1247  
1248  
1249  
1250  
1251  
1252  
1253  
1254  
1255  
1256  
1257  
1258  
1259  
1260  
1261  
1262  
1263  
1264  
1265  
1266  
1267  
1268  
1269  
1270  
1271  
1272  
1273  
1274  
1275  
1276  
1277  
1278  
1279  
1280  
1281  
1282  
1283  
1284  
1285  
1286  
1287  
1288  
1289  
1290  
1291  
1292  
1293  
1294  
1295  
1296  
1297  
1298  
1299  
1300  
1301  
1302  
1303  
1304  
1305  
1306  
1307  
1308  
1309  
1310  
1311  
1312  
1313  
1314  
1315  
1316  
1317  
1318  
1319  
1320  
1321  
1322  
1323  
1324  
1325  
1326  
1327  
1328  
1329  
1330  
1331  
1332  
1333  
1334  
1335  
1336  
1337  
1338  
1339  
1340  
1341  
1342  
1343  
1344  
1345  
1346  
1347  
1348  
1349  
1350  
1351  
1352  
1353  
1354  
1355  
1356  
1357  
1358  
1359  
1360  
1361  
1362  
1363  
1364  
1365  
1366  
1367  
1368  
1369  
1370  
1371  
1372  
1373  
1374  
1375  
1376  
1377  
1378  
1379  
1380  
1381  
1382  
1383  
1384  
1385  
1386  
1387  
1388  
1389  
1390  
1391  
1392  
1393  
1394  
1395  
1396  
1397  
1398  
1399  
1400  
1401  
1402  
1403  
1404  
1405  
1406  
1407  
1408  
1409  
1410  
1411  
1412  
1413  
1414  
1415  
1416  
1417  
1418  
1419  
1420  
1421  
1422  
1423  
1424  
1425  
1426  
1427  
1428  
1429  
1430  
1431  
1432  
1433  
1434  
1435  
1436  
1437  
1438  
1439  
1440  
1441  
1442  
1443  
1444  
1445  
1446  
1447  
1448  
1449  
1450  
1451  
1452  
1453  
1454  
1455  
1456  
1457  
1458  
1459  
1460  
1461  
1462  
1463  
1464  
1465  
1466  
1467  
1468  
1469  
1470  
1471  
1472  
1473  
1474  
1475  
1476  
1477  
1478  
1479  
1480  
1481  
1482  
1483  
1484  
1485  
1486  
1487  
1488  
1489  
1490  
1491  
1492  
1493  
1494  
1495  
1496  
1497  
1498  
1499  
1500  
1501  
1502  
1503  
1504  
1505  
1506  
1507  
1508  
1509  
1510  
1511  
1512  
1513  
1514  
1515  
1516  
1517  
1518  
1519  
1520  
1521  
1522  
1523  
1524  
1525  
1526  
1527  
1528  
1529  
1530  
1531  
1532  
1533  
1534  
1535  
1536  
1537  
1538  
1539  
1540  
1541  
1542  
1543  
1544  
1545  
1546  
1547  
1548  
1549  
1550  
1551  
1552  
1553  
1554  
1555  
1556  
1557  
1558  
1559  
1560  
1561  
1562  
1563  
1564  
1565  
1566  
1567  
1568  
1569  
1570  
1571  
1572  
1573  
1574  
1575  
1576  
1577  
1578  
1579  
1580  
1581  
1582  
1583  
1584  
1585  
1586  
1587  
1588  
1589  
1590  
1591  
1592  
1593  
1594  
1595  
1596  
1597  
1598  
1599  
1600  
1601  
1602  
1603  
1604  
1605  
1606  
1607  
1608  
1609  
1610  
1611  
1612  
1613  
1614  
1615  
1616  
1617  
1618  
1619  
1620  
1621  
1622  
1623  
1624  
1625  
1626  
1627  
1628  
1629  
1630  
1631  
1632  
1633  
1634  
1635  
1636  
1637  
1638  
1639  
1640  
1641  
1642  
1643  
1644  
1645  
1646  
1647  
1648  
1649  
1650  
1651  
1652  
1653  
1654  
1655  
1656  
1657  
1658  
1659  
1660  
1661  
1662  
1663  
1664  
1665  
1666  
1667  
1668  
1669  
1670  
1671  
1672  
1673  
1674  
1675  
1676  
1677  
1678  
1679  
1680  
1681  
1682  
1683  
1684  
1685  
1686  
1687  
1688  
1689  
1690  
1691  
1692  
1693  
1694  
1695  
1696  
1697  
1698  
1699  
1700  
1701  
1702  
1703  
1704  
1705  
1706  
1707  
1708  
1709  
1710  
1711  
1712  
1713  
1714  
1715  
1716  
1717  
1718  
1719  
1720  
1721  
1722  
1723  
1724  
1725  
1726  
1727  
1728  
1729  
1730  
1731  
1732  
1733  
1734  
1735  
1736  
1737  
1738  
1739  
1740  
1741  
1742  
1743  
1744  
1745  
1746  
1747  
1748  
1749  
1750  
1751  
1752  
1753  
1754  
1755  
1756  
1757  
1758  
1759  
1760  
1761  
1762  
1763  
1764  
1765  
1766  
1767  
1768  
1769  
1770  
1771  
1772  
1773  
1774  
1775  
1776  
1777  
1778  
1779  
1780  
1781  
1782  
1783  
1784  
1785  
1786  
1787  
1788  
1789  
1790  
1791  
1792  
1793  
1794  
1795  
1796  
1797  
1798  
1799  
1800  
1801  
1802  
1803  
1804  
1805  
1806  
1807  
1808  
1809  
1810  
1811  
1812  
1813  
1814  
1815  
1816  
1817  
1818  
1819  
1820  
1821  
1822  
1823  
1824  
1825  
1826  
1827  
1828  
1829  
1830  
1831  
1832  
1833  
1834  
1835  
1836  
1837  
1838  
1839  
1840  
1841  
1842  
1843  
1844  
1845  
1846  
1847  
1848  
1849  
1850  
1851  
1852  
1853  
1854  
1855  
1856  
1857  
1858  
1859  
1860  
1861  
1862  
1863  
1864  
1865  
1866  
1867  
1868  
1869  
1870  
1871  
1872  
1873  
1874  
1875  
1876  
1877  
1878  
1879  
1880  
1881  
1882  
1883  
1884  
1885  
1886  
1887  
1888  
1889  
1890  
1891  
1892  
1893  
1894  
1895  
1896  
1897  
1898  
1899  
1900  
1901  
1902  
1903  
1904  
1905  
1906  
1907  
1908  
1909  
1910  
1911  
1912  
1913  
1914  
1915  
1916  
1917  
1918  
1919  
1920  
1921  
1922  
1923  
1924  
1925  
1926  
1927  
1928  
1929  
1930  
1931  
1932  
1933  
1934  
1935  
1936  
1937  
1938  
1939  
1940  
1941  
1942  
1943  
1944  
1945  
1946  
1947  
1948  
1949  
1950  
1951  
1952  
1953  
1954  
1955  
1956  
1957  
1958  
1959  
1960  
1961  
1962  
1963  
1964  
1965  
1966  
1967  
1968  
1969  
1970  
1971  
1972  
1973  
1974  
1975  
1976  
1977  
1978  
1979  
1980  
1981  
1982  
1983  
1984  
1985  
1986  
1987  
1988  
1989  
1990  
1991  
1992  
1993  
1994  
1995  
1996  
1997  
1998  
1999  
2000  
2001  
2002  
2003  
2004  
2005  
2006  
2007  
2008  
2009  
2010  
2011  
2012  
2013  
2014  
2015  
2016  
2017  
2018  
2019  
2020  
2021  
2022  
2023  
2024  
2025  
2026  
2027  
2028  
2029  
2030  
2031  
2032  
2033  
2034  
2035  
2036  
2037  
2038  
2039  
2040  
2041  
2042  
2043  
2044  
2045  
2046  
2047  
2048  
2049  
2050  
2051  
2052  
2053  
2054  
2055  
2056  
2057  
2058  
2059  
2060  
2061  
2062  
2063  
2064  
2065  
2066  
2067  
2068  
2069  
2070  
2071  
2072  
2073  
2074  
2075  
2076  
2077  
2078  
2079  
2080  
2081  
2082  
2083  
2084  
2085  
2086  
2087  
2088  
2089  
2090  
2091  
2092  
2093  
2094  
2095  
2096  
2097  
2098  
2099  
2100  
2101  
2102  
2103  
2104  
2105  
2106  
2107  
2108  
2109  
2110  
2111  
2112  
2113  
2114  
2115  
2116  
2117  
2118  
2119  
2120  
2121  
2122  
2123  
2124  
2125  
2126  
2127  
2128  
2129  
2130  
2131  
2132  
2133  
2134  
2135  
2136  
2137  
2138  
2139  
2140  
2141  
2142  
2143  
2144  
2145  
2146  
2147  
2148  
2149  
2150  
2151  
2152  
2153  
2154  
2155  
2156  
2157  
2158  
2159  
2160  
2161  
2162  
2163  
2164  
2165  
2166  
2167  
2168  
2169  
2170  
2171  
2172  
2173  
2174  
2175  
2176  
2177  
2178  
2179  
2180  
2181  
2182  
2183  
2184  
2185  
2186  
2187  
2188  
2189  
2190  
2191  
2192  
2193  
2194  
2195  
2196  
2197  
2
```

# Application of Tree Traversals

- Table of Contents
  - With Indent
  - With explicit Numbering

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

(a)

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

(b)

```

3 class LinkedTree(Tree):
4     '''Linked representation of General Tree Structure.'''
5
6     class _Node:
7         __slots__ = '_element', '_parent', '_children', '_noc'
8         def __init__(self, element, parent=None):
9             self._element = element
10            self._parent = parent
11            self._children = []
12            self._noc = 0 # number of children
13
14        class Position(Tree.Position):
15            '''An abstraction representing the location of a single element.'''
16
17            def __init__(self, container, node):
18                '''Constructor should not be invoked by user.'''
19                self._container = container
20                self._node = node
21
22            def element(self):
23                '''Return the element stored at this position.'''
24                return self._node._element
25
26            def _eq__(self, other):
27                '''Return True if other is a position representing the same location.'''
28                return type(other) is type(self) and other._node is self._node
29
30        # ----- hidden utility functions for LinkedTree -----
31        def _validate(self, p):
32            '''Return associated node, if position is valid.'''
33            if not isinstance(p, self.Position):
34                raise TypeError('p must be proper Position type')
35            if p._container is not self:
36                raise ValueError('p does not belong to this container')
37            if p._node._parent is p._node: # convention for deprecated nodes
38                raise ValueError('p is no longer valid')
39            return p._node
40
41        def _make_position(self, node):
42            '''Return Position instance for a given node (or None if no node)'''
43            return self.Position(self, node) if node is not None else None
44

```

```

51 # ----- Public Accessors -----
52 def __len__(self):
53     '''Return the total number of elements in the tree.'''
54     return self._size
55
56 def root(self):
57     '''Return the root Position of the tree (or None if tree is empty).'''
58     return self._make_position(self._root)
59
60 def parent(self, p):
61     '''return the position P's parent (or None if p is root)'''
62     node = self._validate(p)
63     return self._make_position(node._parent)
64
65 def children(self, p):
66     '''Generate an iteration of Positions representing p's children'''
67     node = self._validate(p)
68     for i in range(node._noc):
69         if node._children[i] is not None:
70             yield self._make_position(node._children[i])
71
72 def num_children(self, p):
73     '''Return the number of children of Position P.'''
74     node = self._validate(p)
75     return node._noc
76

```



```

76 # ----- Nonpublic tree update methods -----
77
78 def _add_root(self, e):
79     '''
80     Place element e at the root of an empty tree and return new Position.
81     Raise ValueError if tree nonempty.
82     '''
83     if self._root is not None: raise ValueError('Root Exists.')
84     self._size = 1
85     self._root = self._Node(e)
86     return self._make_position(self._root)
87
88 def _add_child(self, p, e):
89     '''
90     Create a new left child for Position P, storing element e.
91     Return the position of a new node.
92     Raise ValueError if Position p is invalid or p already has a left child.
93     '''
94     node = self._validate(p)
95     child = self._Node(e, node) # create a new child with node as its parent
96     node._children.append(child) # add child to the list with
97     self._size += 1
98     node._noc += 1
99     return self._make_position(child) # return current child position
100
101 def _replace(self, p, e):
102     ''' replace the element at position P with e and return the old element.'''
103     node = self._validate(p)
104     old = node._element
105     node._element = e
106     return old
107

```

```

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.
    *** NOT TEST YET ****
    '''
    node = self._validate(p)

    if node._noc > 1:
        raise ValueError('p has more than 1 child. Node can not be deleted')

    if node is self._root: # if we are deleting root node
        for i in range(node._noc):
            child = node._children[i]
            if child is not None:
                self._root = child
    else: # if not a root node
        for i in range(node._noc):
            child = node._children[i]
            if child is not None:
                child._parent = node._parent # child's grandparent becomes parent
    self._size -= 1
    node._parent._noc -= 1 # decrease the number of children for the parent
    node._parent = node # convention for deprecated node
    return node._element

def _preorder_indent(self, p, d):
    '''Print preorder representation of subtree T rooted at p at depth d.'''
    print(2*d*' ' + str(p.element()))
    if not self.is_leaf(p):
        for c in self.children(p):
            self._preorder_indent(c, d+1)

def _print_tree(self):
    p = self.root()
    d = 0
    #set_trace()
    self._preorder_indent(p, d)

```

```

A = LinkedTree()
A._add_root('Paper')
A._add_child(A.root(), 'Title')
A._add_child(A.root(), 'Abstract')
A._add_child(A.root(), 'Section1')
A._add_child(A.root(), 'Section2')
A._add_child(A.root(), 'Conclusion')
print('Root has {} children'.format(A.num_children(A.root())))
L1 = []
for child in A.children(A.root()):
    L1.append(child)

A._add_child(L1[2], 'Sec1.1')
A._add_child(L1[2], 'Sec1.2')
A._add_child(L1[2], 'Sec1.3')

print('L1[2] has {} children'.format(A.num_children(L1[2])))

A._add_child(L1[3], 'Sec2.1')
A._add_child(L1[3], 'Sec2.2')
A._add_child(L1[3], 'Sec2.3')

# Prints the labels with indent
A.print_tree_with_indent()

print("-----")

for p in A.preorder(): # preorder traversal
    print(p.element())
print("-----")

for p in A.postorder(): # postorder traversal
    print(p.element())
print("-----")

# Execute the cell containing LinkedQueue for this
for p in A.breadthfirst(): # breadthfirst traversal
    print(p.element())

```

```

Root has 5 children
L1[2] has 3 children
Paper
  Title
  Abstract
  Section1
    Sec1.1
    Sec1.2
    Sec1.3
  Section2
    Sec2.1
    Sec2.2
    Sec2.3
  Conclusion
-----
Paper
  Title
  Abstract
  Section1
    Sec1.1
    Sec1.2
    Sec1.3
  Section2
    Sec2.1
    Sec2.2
    Sec2.3
  Conclusion
-----
Title
Abstract
Sec1.1
Sec1.2
Sec1.3
Section1
  Sec2.1
  Sec2.2
  Sec2.3
Section2
Conclusion
Paper
-----

```

```

def preorder_label(self, p, d, path):
    '''Print a labeled representation of subtree 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 self.children(p):
        self.preorder_label(c, d+1, path) # child depth is d+1
    path[-1] += 1
    path.pop()

def print_tree_with_labels(self):
    ''' Print tree with Labels.'''
    p = self.root()
    d = 0
    path = []
    self.preorder_label(p, d, path)

```

```

211 print("-----")
212
213 A.print_tree_with_labels()
214

```

```

-----
Paper
Title
Abstract
Section1
Section2
Conclusion
Sec1.1
Sec1.2
Sec1.3
Sec2.1
Sec2.2
Sec2.3

```

```

-----
Paper
1 Title
2 Abstract
3 Section1
  3.1 Sec1.1
  3.2 Sec1.2
  3.3 Sec1.3
4 Section2
  4.1 Sec2.1
  4.2 Sec2.2
  4.3 Sec2.3
5 Conclusion

```

# Parenthetic Representations of a Tree

- It is not possible to reconstruct a general tree, given only the preorder sequence of elements.
- Some additional context is necessary for the structure of the tree to be well defined.
- The use of indentation or numbered labels provides such context, with a very human-friendly presentation.
- There is a need for more concise string representations of trees that are computer-friendly.
- Parenthetic representation is one such representation.

- The ***parenthetic string representation***  $P(T)$  of tree  $T$  is recursively defined as follows:

- If  $T$  consists of a single position  $p$ , then  $P(T) = \text{str}(p.\text{element}())$
- Otherwise, it is defined recursively as,

$$P(T) = \text{str}(p.\text{element}()) + '(' + P(T_1) + ', ' + \dots + ', ' + P(T_k) + ')'$$

where  $p$  is the root of  $T$  and  $T_1, T_2, \dots, T_k$  are the subtrees rooted at the children of  $p$ , which are given in order if  $T$  is an ordered tree.

Electronics R'Us

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

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

Parenthetic representation

Labeled representation

```

167 def parenthesize(self, p):
168     '''Print parenthesized representation of subtree of T rooted at p.'''
169     print(p.element(), end='') # use of end avoids trailing newline
170     if not self.is_leaf(p):
171         first_time = True
172         for c in self.children(p):
173             sep = ' (' if first_time else ', ' # determine proper separator
174             print(sep, end='')
175             first_time = False # any future passes will not be the first
176             self.parenthesize(c) # recur on child
177         print(')', end='') # include closing parenthesis
178
179 def print_tree_with_parenthesis(self):
180     '''Generate parenthetic representation of the tree.'''
181     self.parenthesize(self.root())
182
183

```

```

230 print("-----")
231 A.print_tree_with_parenthesis()
232

```

-----

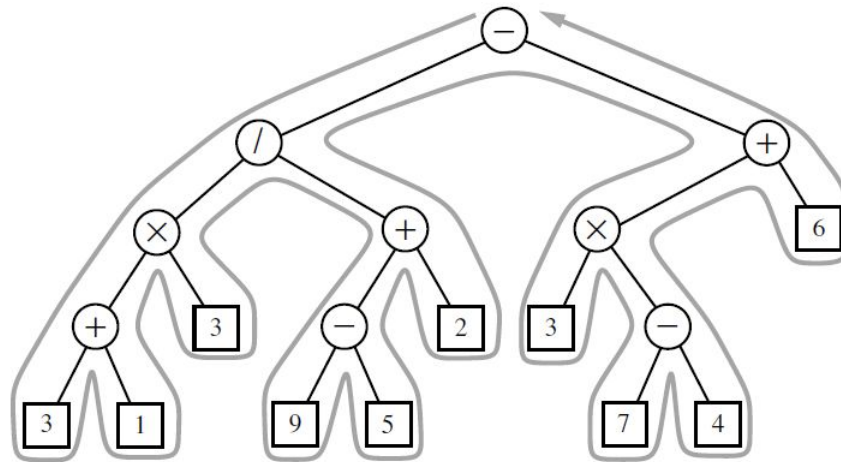
Paper (Title, Abstract, Section1 (Sec1.1, Sec1.2, Sec1.3), Section2 (Sec2.1, Sec2.2, Sec2.3), Conclusion)

---

# Euler Tour Traversal

- A more general tree traversal algorithm.
- “Walk” around the tree  $T$  where we start by going from the root toward its leftmost child, viewing the edges of  $T$  as being “walls” that we always keep to our left.
- The **complexity of the walk is  $O(n)$** 
  - because it progresses exactly two times along each of the  $n-1$  edges of the tree—once going downward along the edge, and later going upward along the edge.
- Combines preorder and postorder traversals.
- There are two notable “visits” to each position  $p$ :
  - A “**pre visit**” occurs when first reaching the position, that is, when the walk passes immediately left of the node in our visualization.
  - A “**post visit**” occurs when the walk later proceeds upward from that position, that is, when the walk passes to the right of the node in our visualization.





Euler tour traversal of a tree.

**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  $p$

```

1 class EulerTour:
2     '''
3     Abstract Base Class for performing Euler Tour of a tree.
4     _hook_previsit and _hook_postvisit may be overridden by subclasses.
5
6     '''
7     def __init__(self, tree):
8         '''Prepare an Euler Tour template for given tree.'''
9         self._tree = tree
10
11     def tree(self):
12         '''Return reference to the tree being traversed.'''
13         return self._tree
14
15     def execute(self):
16         '''Perform the tour and return any result from post visit of root.'''
17         if len(self._tree) > 0:
18             return self._tour(self._tree.root(), 0, []) # start recursion
19
20     def _tour(self, p, d, path):
21         '''Perform tour of subtree rooted at Position P.
22
23         p: Position of current node being visited
24         d: depth of p in the tree
25         path: list of indices of children on path from root to p.
26
27         '''
28         self._hook_previsit(p, d, path) # "pre visit" p
29         results = []
30         path.append(0) # add new index to end of path before recursion
31         for c in self._tree.children(p):
32             results.append(self._tour(c, d+1, path)) # recur on child's subtree
33             path[-1] += 1 # increment index
34         path.pop()
35         answer = self._hook_postvisit(p, d, path, results) # "post visit" p
36         return answer
37
38     def _hook_previsit(self, p, d, path): # can be overridden
39         pass
40
41     def _hook_postvisit(self, p, d, path, results): # can be overridden
42         pass

```

```

📄 Paper
  Title
  Abstract
  Section1
    Sec1.1
    Sec1.2
    Sec1.3
  Section2
    Sec2.1
    Sec2.2
    Sec2.3
  Conclusion
-----
Paper
  1 Title
  2 Abstract
  3 Section1
    3.1 Sec1.1
    3.2 Sec1.2
    3.3 Sec1.3
  4 Section2
    4.1 Sec2.1
    4.2 Sec2.2
    4.3 Sec2.3
  5 Conclusion

```

-----  
 Paper (Title, Abstract, Section1 (Sec1.1, Sec1.2, Sec1.3), Section2 (Sec2.1, Sec2.2, Sec2.3), Conclusion)

```

44 class PreorderPrintIndentedTour(EulerTour):
45     def _hook_previsit(self, p, d, path):
46         print(2*d*' ' + str(p.element()))
47
48     class PreorderPrintIndentedTour2(EulerTour):
49         def _hook_previsit(self, p, d, path):
50             label = '.'.join(str(j+1) for j in path) # labels are one-indexed
51             print(2*d*' ' + label, str(p.element()))
52
53     class ParenthesizeTour(EulerTour):
54         def _hook_previsit(self, p, d, path):
55             if path and path[-1] > 0: # p follows a sibling
56                 print(', ', end='') # so preface with comma
57                 print(p.element(), end='') # then print element
58                 if not self._tree().is_leaf(p): # if p has children
59                     print(' (', end='') # print open parenthesis
60
61         def _hook_postvisit(self, p, d, path, results):
62             if not self._tree().is_leaf(p): # if p has children
63                 print(')', end='')
64
65     # Test
66     tour = PreorderPrintIndentedTour(A)
67     tour.execute()
68     print("-----")
69     tour2 = PreorderPrintIndentedTour2(A)
70     tour2.execute()
71     print("-----")
72     tour3 = ParenthesizeTour(A)
73     tour3.execute()
74

```



# Summary

We study the following:

- General Tree ADT - their properties
- Binary Tree ADT - properties
- Python implementation of trees using Linked Lists and Arrays
- Tree Traversal Algorithms