Trees

Data Structures & Algorithms with Python

Lecture 8

Overview

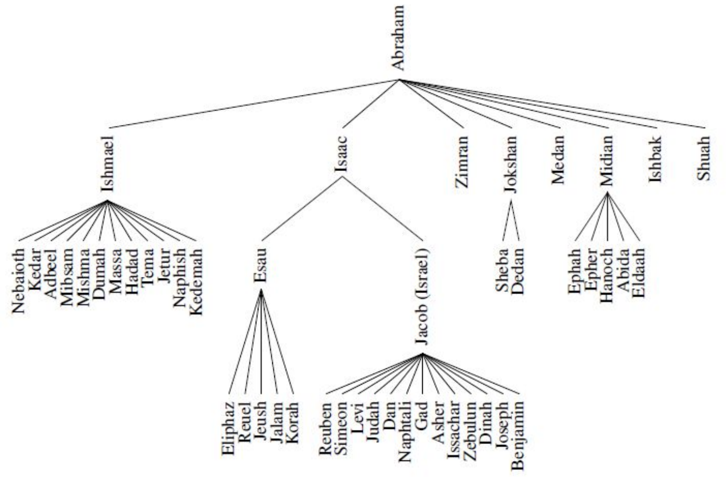
- Introduction to Trees
  - Definitions, Properties
  - Tree ADT - Computing Depth & Height
- Binary Trees
- Implementing Trees
  - Using Linked Lists
  - Using Arrays
- Tree Traversal Algorithms
- Summary

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.

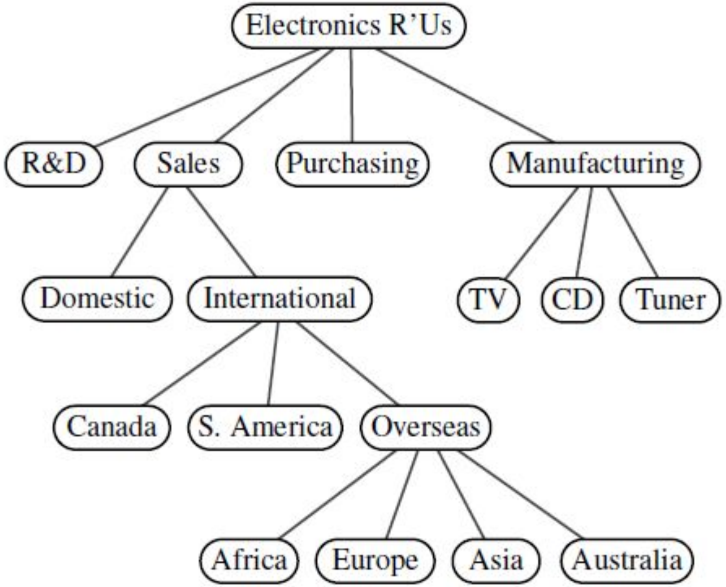
- Provide a natural organization of data which is inherently hierarchical.



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.***

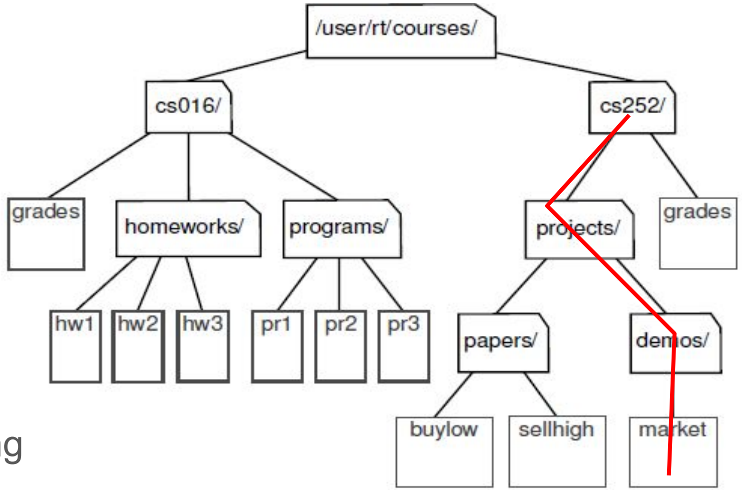
- A node u is an ***ancestor*** of a node v if u = v or u is an ancestor of the parent of v.

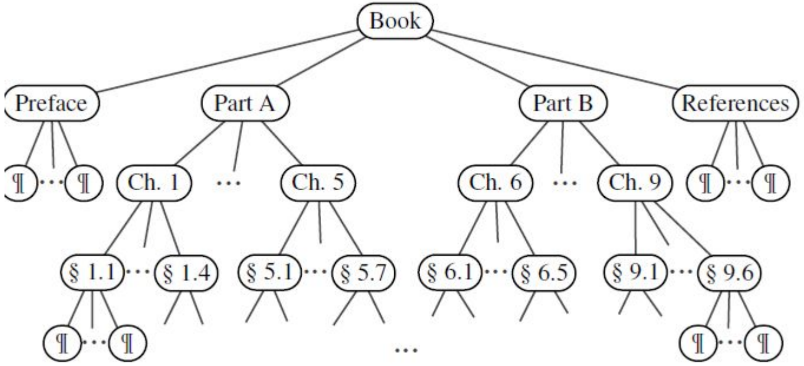
- Conversely, we say that a node v is a ***descendant*** of a node u if u is an ancestor of v.

- An ***edge*** of tree T is a pair of nodes (u,v) such that u is the parent of v, or vice versa.

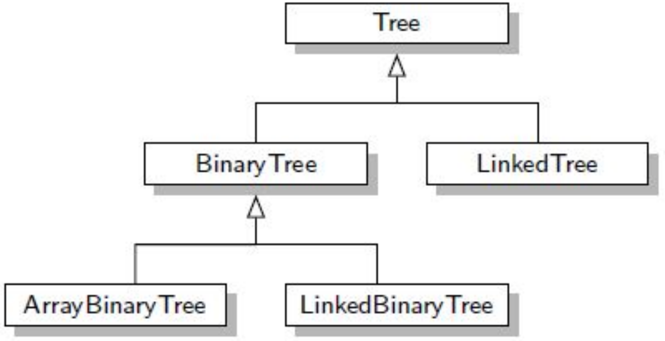
- A **path** of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge. Example: The adjacent tree contains the path (cs252/, projects/, demos/, market)

- A tree is ***ordered*** if there is a meaningful linear order among the children of each node.





An ordered tree associated with a book.



The inheritance hierarchy for modeling various abstractions and implementations of tree data structures.

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 importan*t. 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.

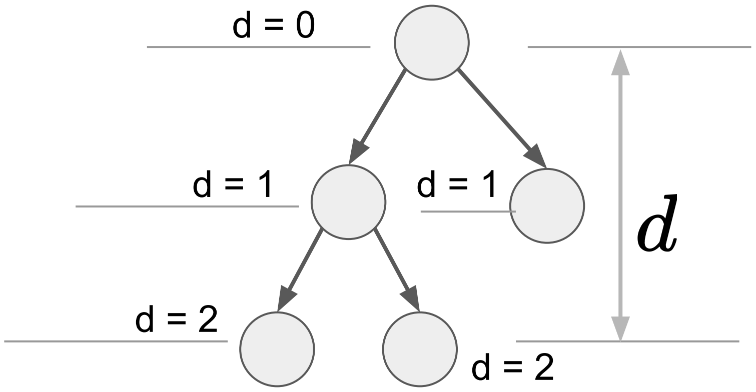
- The base class consists of
    - A nested position class with abstract methods.
    - Several abstract methods
    - Few concrete methods:
        - **`is_root()`**
        - **`is_leaf()`**
        - **`is_empty()`**

- No direct instance is created for the abstract base class

```python
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 the other does not represent the same location.'''
            return not(self == other)

    #------------ Abstract Methods that concrete subclass must support ----

    def root(self):
        ''' Returns Position representing the tree's root (or None if Empty)'''
        raise NotImplementedError('must be implemented by subclass')

    def parent(self, p):
        ''' Returns Position representing p's parent (or None if Empty)'''
        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):
        '''Return the number of children that Position p has.'''
        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' )

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

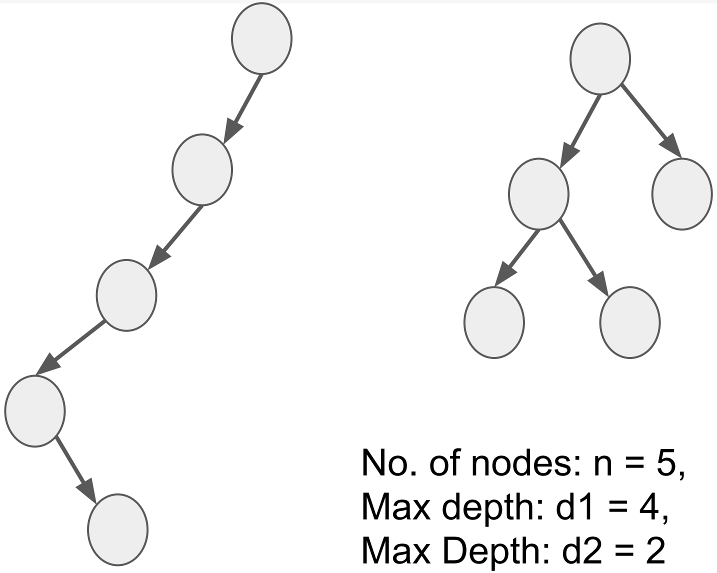
Computing Depth

- 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.

```python
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: d1 = 4,
Max Depth: d2 = 2

- The running time of `T.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.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.

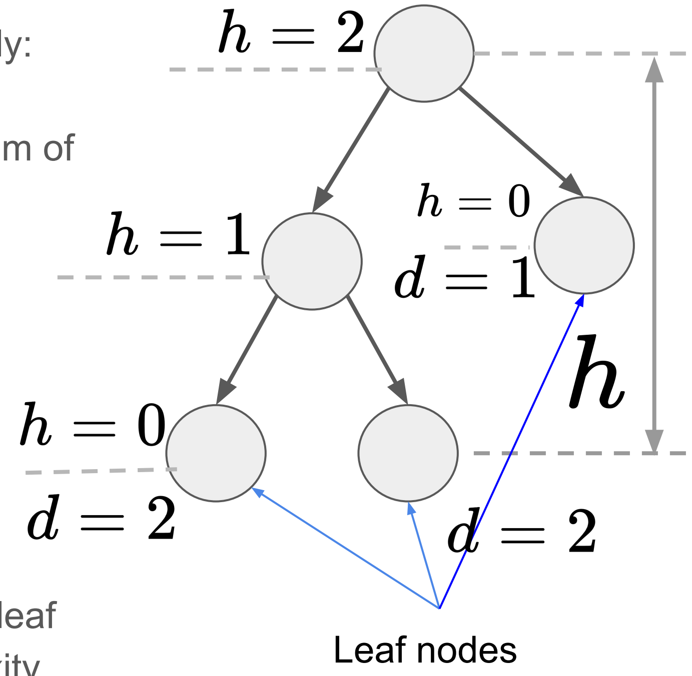
Computing Height

- 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:

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

```python
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)))
```

$\sim O(n^2)$

- 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

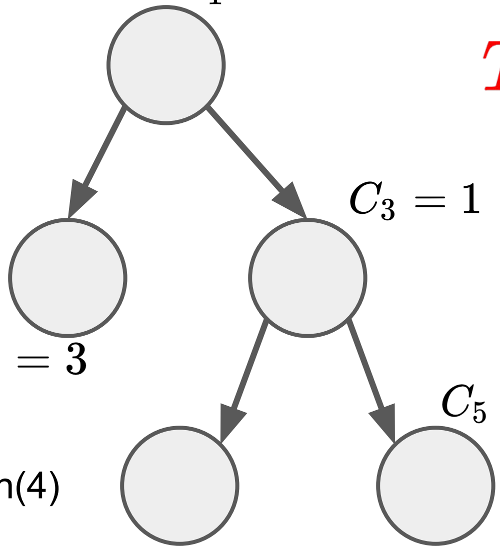
```
depths =[]
for p in  range (n):   (executed n times)
    if is_leaf(p):
        d = depth(p)   (executed L times)
        depths.append(d)
H = max(depths)
```

- So, $\sum_{p \in L}(d_p + 1) \propto n^2$

- So the worst-case time complexity of **`_height1()`** function is

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

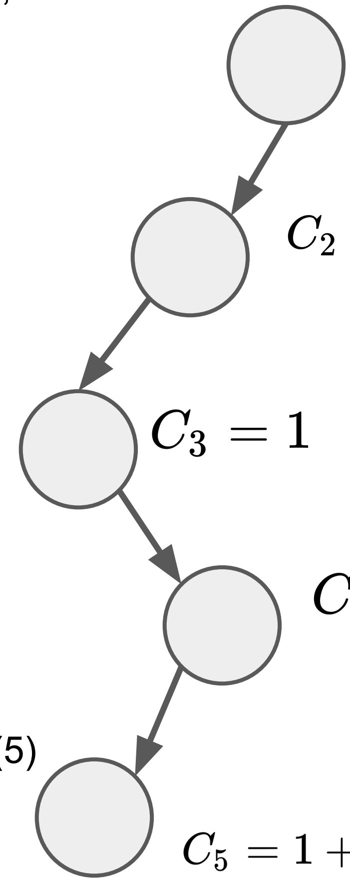
Case 1: n = 5, L = 3



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

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

Case II: n = 5, L = 1



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

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

- `_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

$$O(\sum_p (c_p + 1)) = O(n + \sum_p c_p)$$
$$= O(n + n - 1)$$
$$\sim O(n)$$

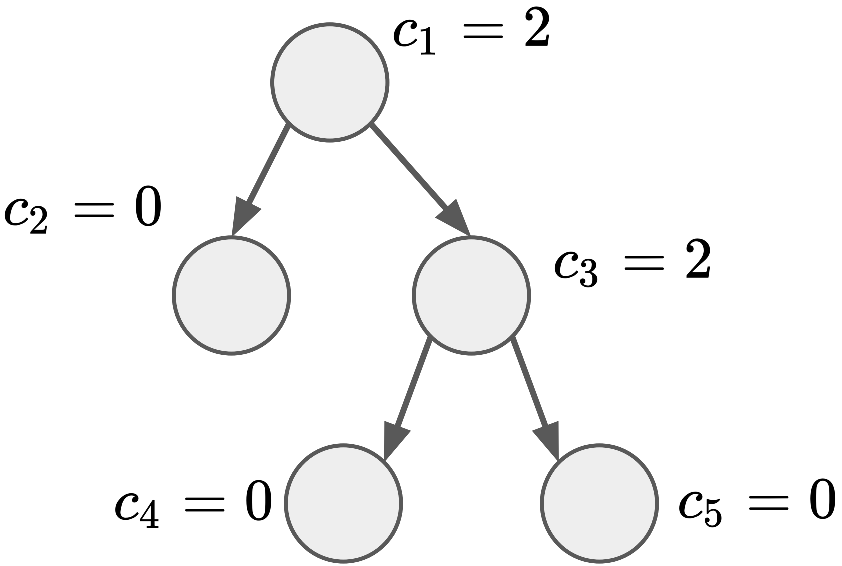
```python
def height(self, p = None):
  '''
  Return the height of subtree rooted at Position P
  if p is None, return the height of the entire tree.
  '''
  if p is None:
    p = self.root()
  return self._height2(p)   # start _height2 recursion
```

```python
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))
```

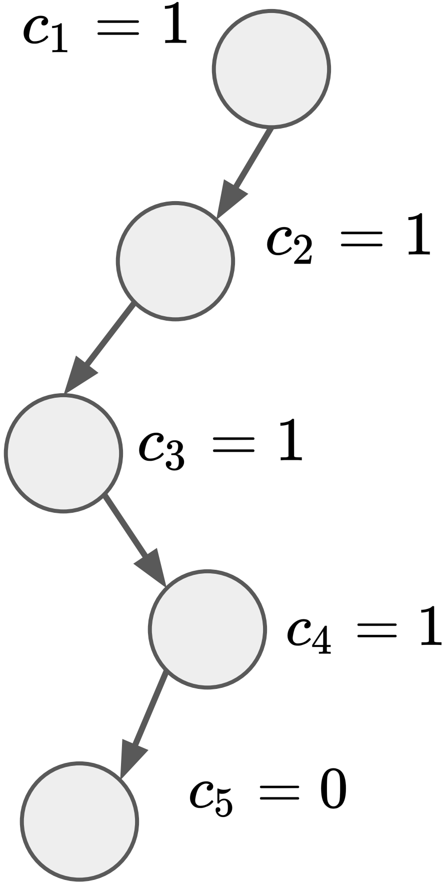
Proposition: *Let T be a tree with n positions, and let c_p denote the number of children of a position p of T. Then, summing over the positions of T,*

$$\sum_p c_p = n - 1$$

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.



$$n = 5$$

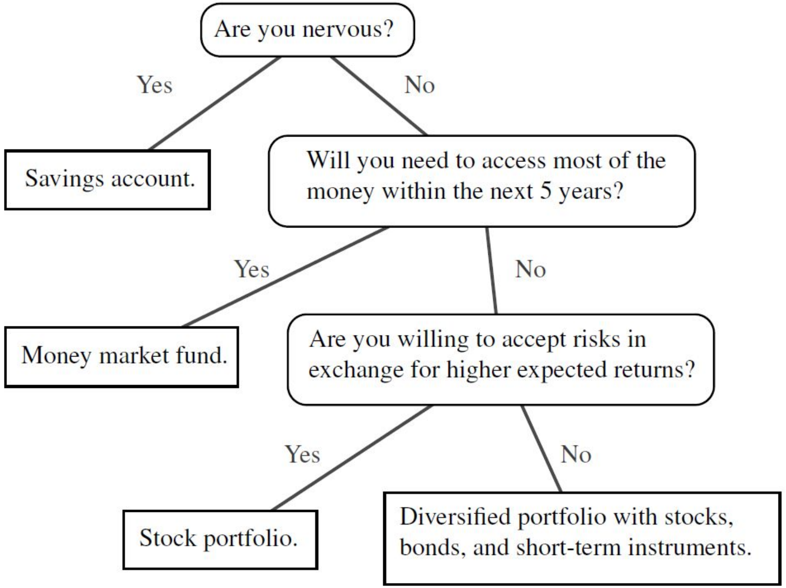


Total number of Children In the tree:

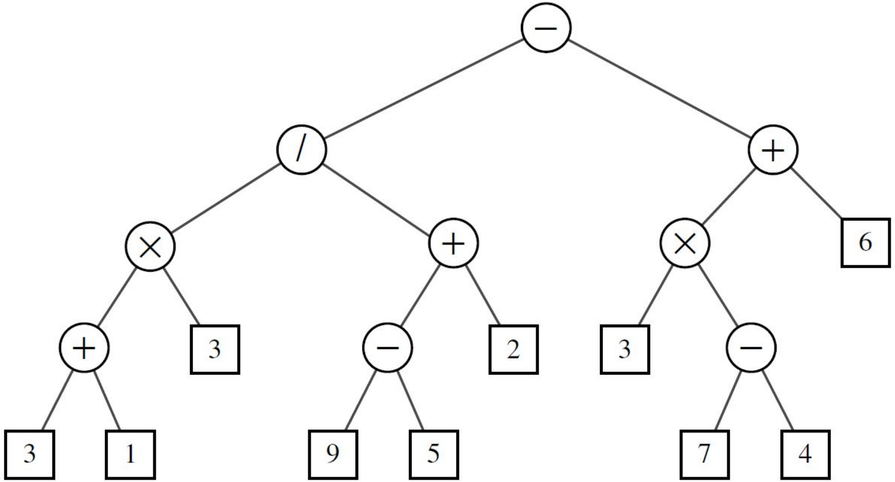
$$\sum_p c_p = 4$$

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.left(p)`**: Return the position that represents the left child of p, or None if p has no left child.
    - **`T.right(p)`**: Return the position that represents the right child of p, or None if p has no right child.
    - **`T.sibling(p)`**: Return the position that represents the sibling of p, or None if p has no sibling.

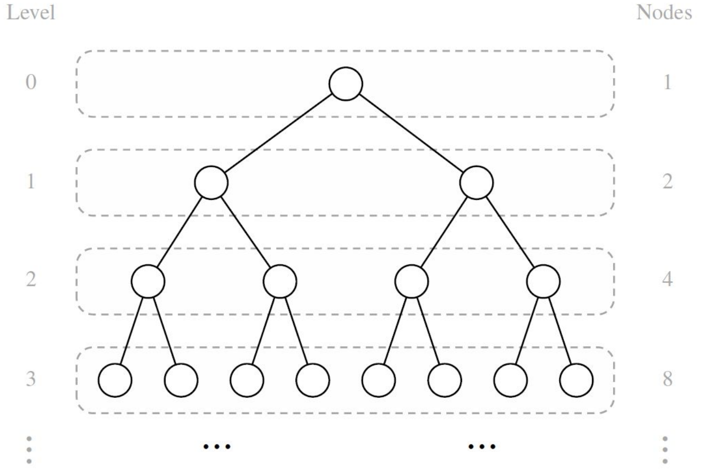
The Binary Tree Abstract Base Class Implementation

- **`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.

```
class BinaryTree(Tree):
  '''Abstract base class representing a binary tree structure.'''

  # -------- 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')

  # ---------- concrete methods implemented in this 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


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



Maximum number of nodes in the levels of a binary 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:*

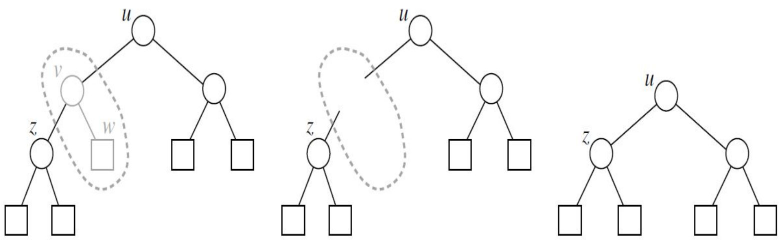
1. $h+1 \le n \le 2^{h+1}-1$
2. $1 \le n_E \le 2^h$
3. $h \le n_I \le 2^h - 1$
4. $\log(n+1)-1 \le h \le n-1$

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

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

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.



(a) (b) (c)

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

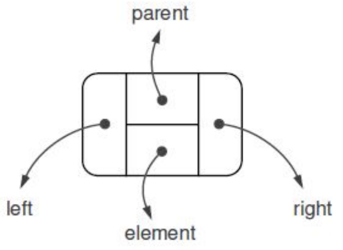
Linked Structure for Binary Trees

Each node maintains the references to the following:

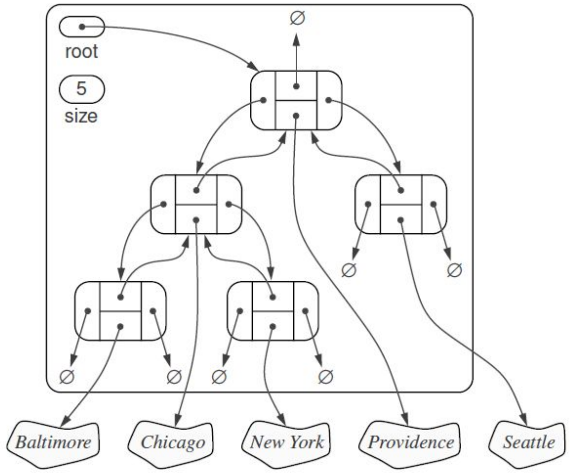
- The element stored at a position p.
- Left child node
- Right child node
- Parent node.

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.



(a)



(b)

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

```python
# Linked Binary tree
class LinkedBinaryTree(BinaryTree):
  '''Linked representation of a binary tree structure.'''

  class _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

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

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

    def element(self):
      '''Return the element stored at this position.'''
      return self._node._element

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

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

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

```python
#-------- 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:      # left child exists
    count += 1
  if node._right is not None:     # right child exists
    count += 1
  return count
```

```python
# ----------- 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
```

```python
    def _delete(self, p):
        '''
        Delete the node at Position p, and replace it with its child, if any.
        Return the element that had been stored at Position p.
        Raise ValueError if Position p is invalid or p has two children.
        '''
        node = self._validate(p)
        if self.num_children(p) == 2: raise ValueError('p has two children')
        child = node._left if node._left else node._right
        if child is not None:
          child._parent = node._parent   # child's grandparent becomes parent
        if node is self._root:
          self._root = child         # child becomes root if its parent is deleted
        else:
          parent = node._parent
          if node is parent._left:
            parent._left = child
          else:
            parent._right = child
        self._size -= 1
        node._parent = node              # convention for deprecated node
        return node._element

    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 tree must be same type
          raise TypeError('Tree types must match')
        self._size += len(t1) + len(t2)

        if not t1.is_empty():  # attach 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():   # attach 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
```

```python
    def _print_children(self, p, arr):
        if p is not None:
            arr += str(p.element())+': '
            if self.num_children(p) > 0:
                for c in self.children(p):
                    arr += str(c.element()) + '\t'
                arr += '\n'
                for c in self.children(p):
                    arr = self._print_children(c, arr)
            arr += '\n'
        return arr

    def __str__(self):
        ''' Provide a string representation of the tree'''

        start = self.root()
        arr = ''
        arr = self._print_children(start, arr)
        return arr
```

Above function prints a binary tree.

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

```python
############

P = LinkedBinaryTree()
P._add_root('Providence')
P._add_left(P.root(), 'Chicago')
P._add_right(P.root(), 'Seattle')
P._add_left(P.left(P.root()), 'Baltimore')
P._add_right(P.left(P.root()), 'New York')
print(P)
print ('\n-----------------------------------\n')

Q = LinkedBinaryTree()
Q._add_root('-')
Q._add_left(Q.root(),'/')
Q._add_right(Q.root(),'+')
Q._add_left(Q.left(Q.root()), 'X')
Q._add_right(Q.left(Q.root()), '+')

Q._add_left(Q.right(Q.root()), 'X')
Q._add_right(Q.right(Q.root()),6)

Q._add_left(Q.left(Q.left(Q.root())), '+')
Q._add_right(Q.left(Q.left(Q.root())), '3')
Q._add_left(Q.left(Q.left(Q.left(Q.root()))), '3')
Q._add_right(Q.left(Q.left(Q.left(Q.root()))), '1')

Q._add_left(Q.right(Q.left(Q.root())), '-')
Q._add_right(Q.right(Q.left(Q.root())), '2')
Q._add_left(Q.left(Q.right(Q.left(Q.root()))), '9')
Q._add_right(Q.left(Q.right(Q.left(Q.root()))), '5')

Q._add_left(Q.left(Q.right(Q.root())), '3')
Q._add_right(Q.left(Q.right(Q.root())), '-')

Q._add_left(Q.right(Q.left(Q.right(Q.root()))), '7')
Q._add_right(Q.right(Q.left(Q.right(Q.root()))), '4')

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	$O(1)$

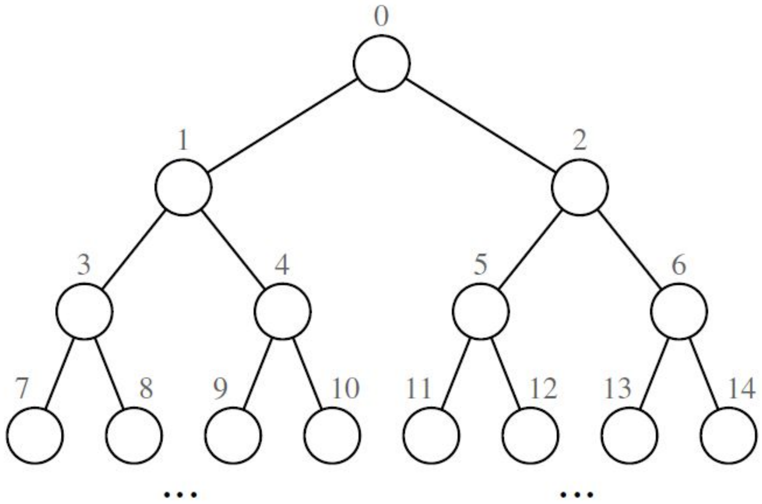
Array-Based Representation of a Binary Tree

- An alternative representation of a binary tree T is based on a way of numbering the positions of T.

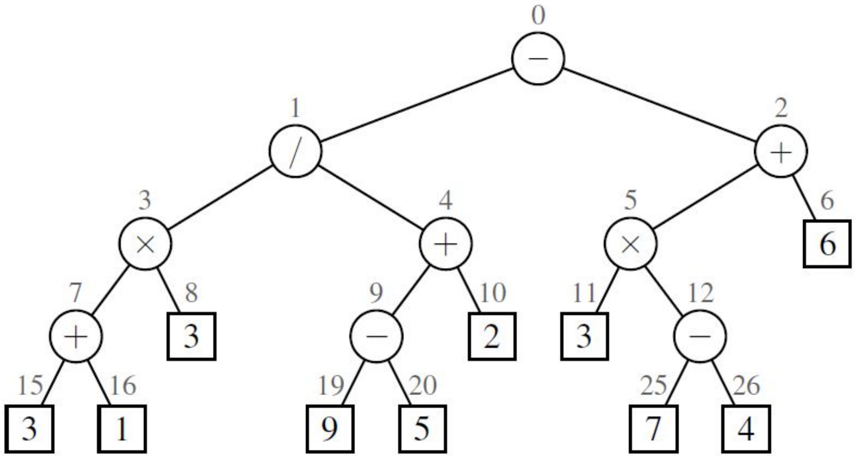
- For every position p of T, let f(p) be the integer defined as follows:
  - If p is the root of T, then f (p) = 0.
  - If p is the left child of position q, then f(p) = 2 f(q)+1.
  - If p is the right child of position q, then f(p) = 2 f(q)+2.

- 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.

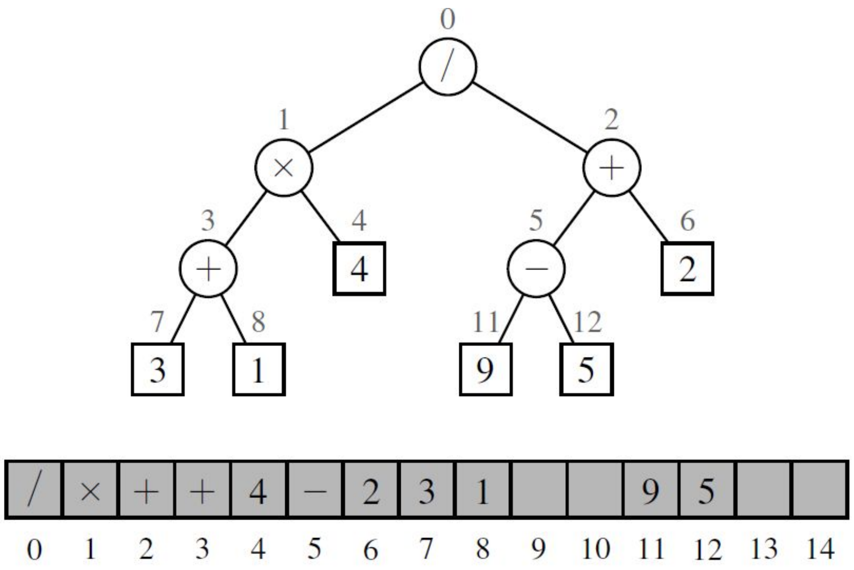


Binary Tree level numbering scheme



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 <u>*exponential worst-case space requirement*</u> of this representation is prohibitive.

Disadvantages of Array-based Tree implementation

- Array-based trees have ***exponential worst-case space requirement*** making them less desirable.

- <u>*Some update operations for trees cannot be efficiently supported*</u> 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.

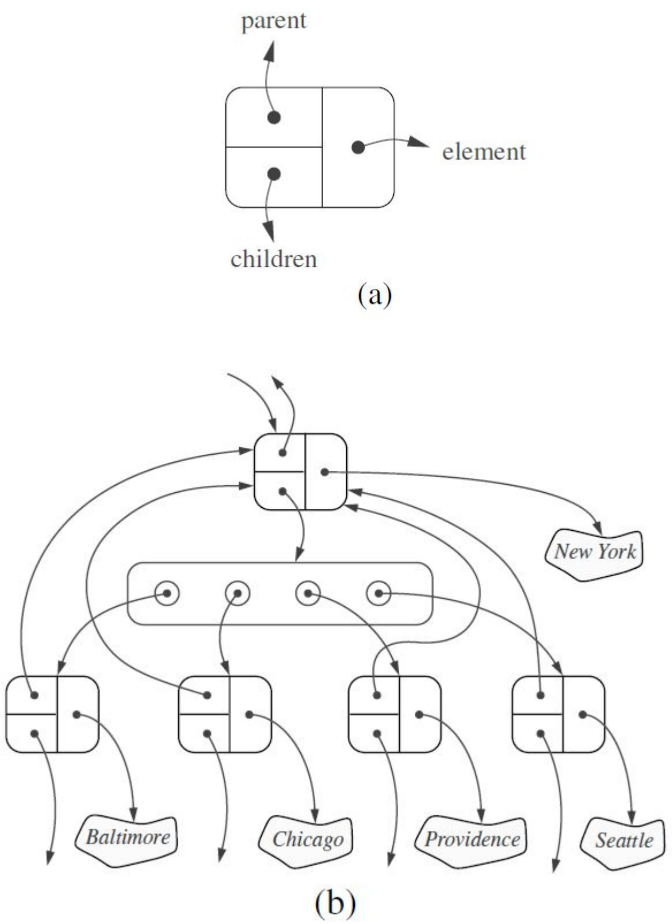
Linked Structure for General Trees

- 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	$O(n)$



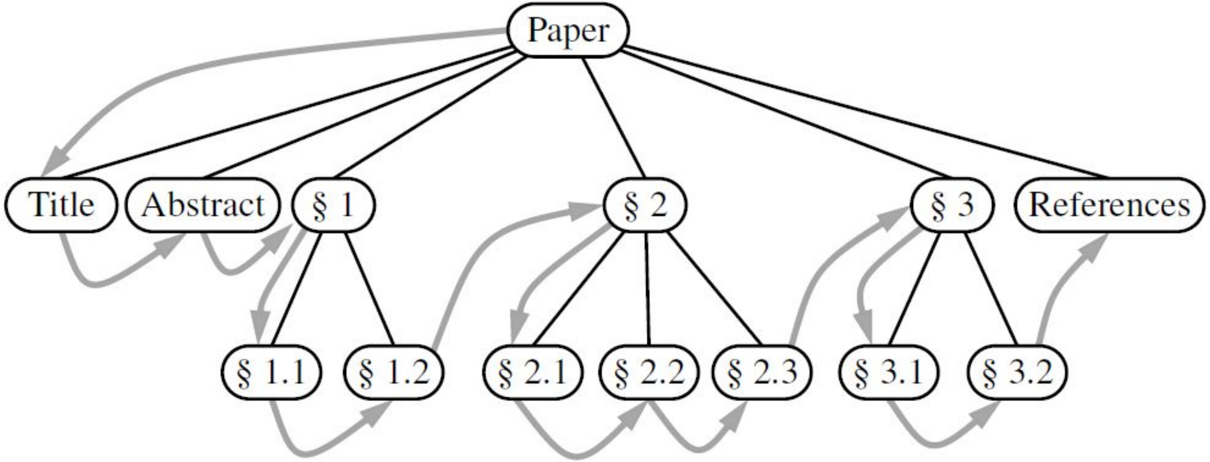
Tree Traversal Algorithms

- 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.

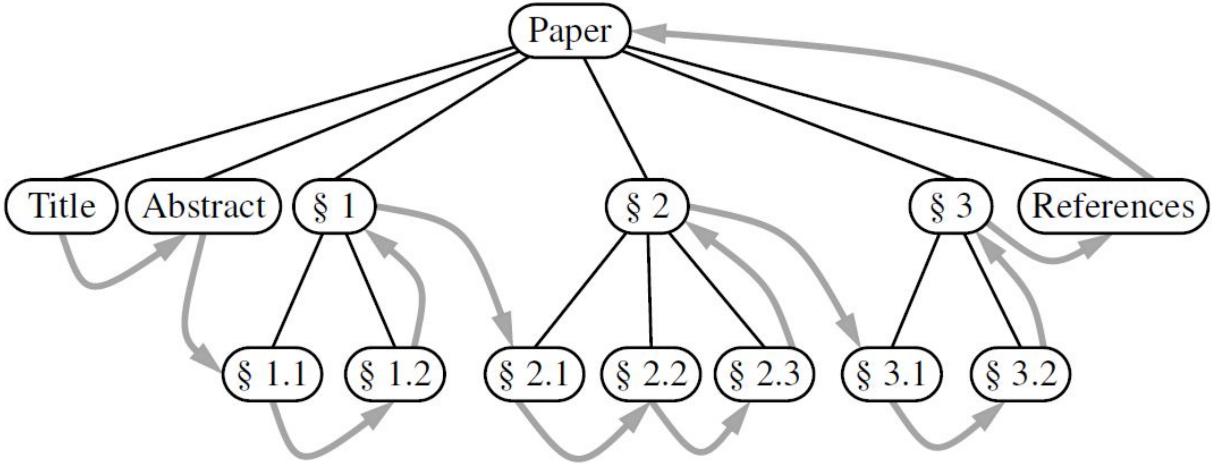
- If the tree is ordered, then the subtrees are traversed according to the order of the children.

- 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

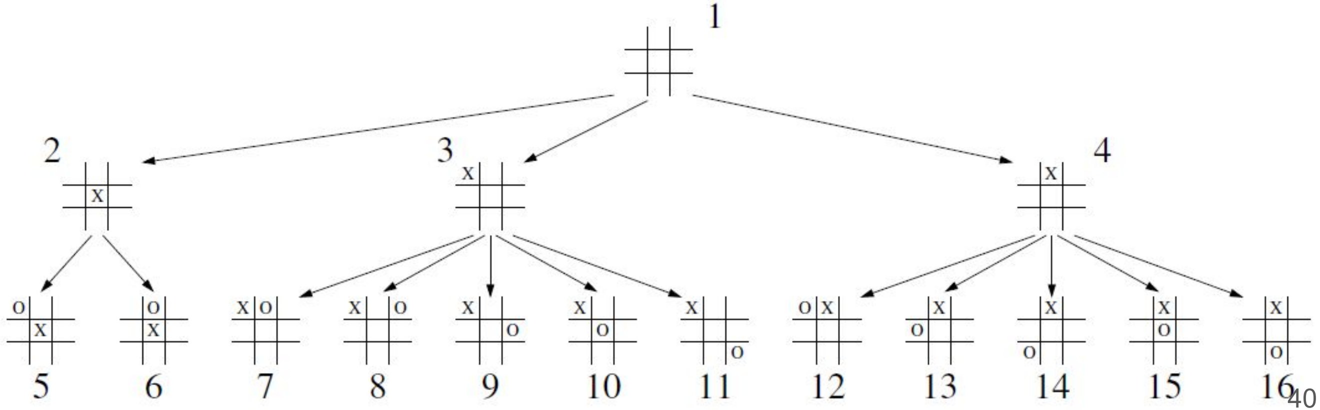
Breadth-First Tree Traversal

- ***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

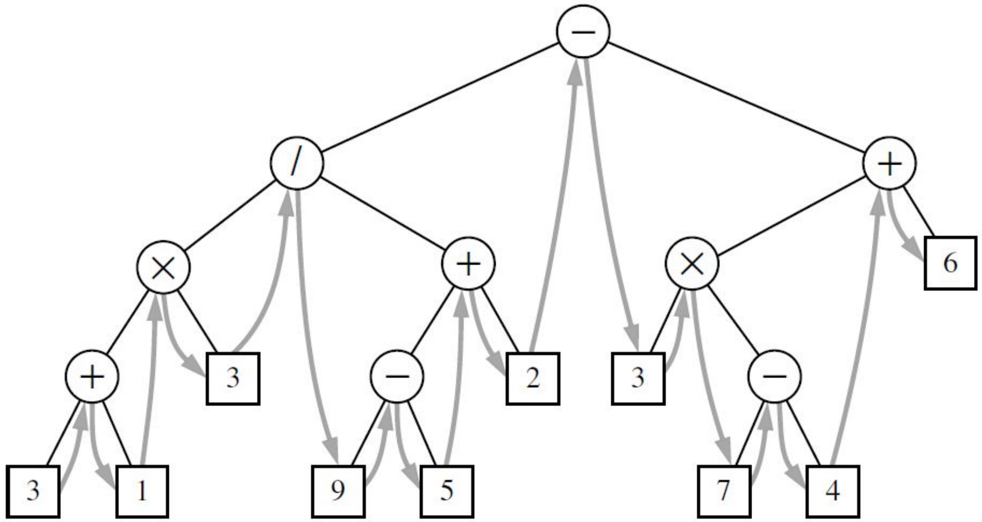
- The standard preorder, postorder, and breadth-first traversals described for general trees, can be directly applied to binary trees.

- ***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}

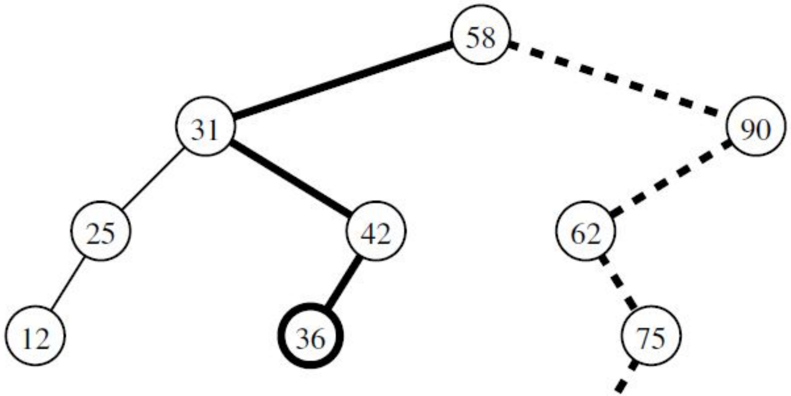
Binary Search Trees

- 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())
```

```python
class Tree:
  '''Abstract Base Class representing a tree structure'''

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

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

  def _subtree_preorder(self, p):
    '''Generate a preorder iteration of positions in subtree rooted at p.'''
    yield p                          # visit p before its subtrees
    for c in self.children(p):      # for each child c
      for other in self._subtree_preorder(c): # do preorder of c's subtree
        yield other                 # yield each element

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

  def postorder(self):
    '''Generate post order iteration of positions in the tree.'''
    if not self.is_empty():
      for p in self._subtree_postorder(self.root()):  # start recursion
        yield p

  def _subtree_postorder(self, p):
    '''Generate a postorder iteration of positions in subtree rooted at p.'''
    for c in self.children(p):          # for each child c
      for other in self._subtree_postorder(c):  # do postorder of c's subtree
        yield other                            # yield each element
    yield p              # then visit p after visiting sub-trees.

  def breadthfirst(self):
    '''Generate a breadth-first iteration of the positions of the tree.'''
    if not self.is_empty():
      fringe = LinkedQueue()          # known positions not yet yielded
      fringe.enqueue(self.root())     # store in queue starting with root
      while not fringe.is_empty():
        p = fringe.dequeue()          # remove from front of the queue
        yield p                       # report this position
        for c in self.children(p):
          fringe.enqueue(c)           # add children to back of queue
```

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)

```python
class LinkedTree(Tree):
  '''Linked representation of General Tree Structure.'''

  class _Node:
    __slots__='_element', '_parent','_children', '_noc'
    def __init__(self, element, parent=None):
      self._element = element
      self._parent = parent
      self._children = []
      self._noc = 0 # number of children

  class Position(Tree.Position):
    '''An abstraction representing the location of a single element.'''

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

    def element(self):
      '''Return the element stored at this position.'''
      return self._node._element

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

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

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

```python
  # ----------- 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 children(self, p):
    ''' Generate an iteration of Positions representing p's children'''
    node = self._validate(p)
    for i in range(node._noc):
      if node._children[i] is not None:
        yield self._make_position(node._children[i])

  def num_children(self, p):
    '''Return the number of children of Position P.'''
    node = self._validate(p)
    return node._noc
```

```python
    # ----- 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_child(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)
        child = self._Node(e, node)   # create a new child with node as its parent
        node._children.append(child) # add child to the list with
        self._size += 1
        node._noc += 1
        return self._make_position(child) # return current child position

    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
```

```python
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 childen 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 a 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)
```

```python
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
------------------
```

```python
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)
```

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

```python
print("------------------")

A.print_tree_with_labels()
```

```
------------------
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,

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

  where p is the root of T and T1,T2, . . . ,Tk 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
```

Labeled representation

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

Parenthetic representation

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

    def print_tree_with_parenthesis(self):
        '''Generate parenthetic representation of the tree.'''
        self.parenthesize(self.root())
```

```python
print("------------------")
A.print_tree_with_parenthesis()
```

```
-------------------
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.

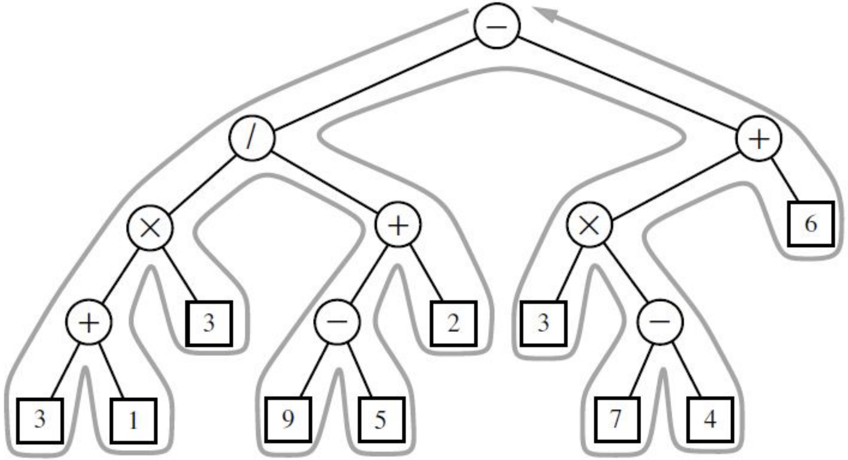
- “Walk” around the tree T where we start by going from the root toward its leftmost child, viewing the edges of T as being “walls” that we always keep to our left.

- 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

```python
class EulerTour:
  '''
  Abstract Base Class for performing Euler Tour of a tree.
  _hook_previsit and _hook_postvisit may be overriden by subclasses.

  '''
  def __init__(self, tree):
    '''Prepare an Euler Tour template for given tree.'''
    self._tree = tree

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

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

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

    p: Position of current node being visited
    d: depth of p in the tree
    path: list of indices of children on path from root to p.

    '''
    self._hook_previsit(p, d, path)            #"pre visit" p
    results = []
    path.append(0)           # add new index to end of path before recursion
    for c in self._tree.children(p):
      results.append(self._tour(c, d+1, path))  # recur on child's subtree
      path[-1] += 1        # increment index
    path.pop()
    answer = self._hook_postvisit(p, d, path, results) # "post visit" p
    return answer

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

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

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

class PreorderPrintIndentedTour2(EulerTour):
  def _hook_previsit(self, p, d, path):
    label = '.'.join(str(j+1)for j in path)  # labels are one-indexed
    print(2*d*' '+label, str(p.element()))

class ParenthesizeTour(EulerTour):
  def _hook_previsit(self, p, d, path):
    if path and path[-1] > 0:          # p follows a sibling
      print(', ', end='')              # so preface with comma
    print(p.element(), end='')         # then print element
    if not self.tree().is_leaf(p):     # if p has children
      print(' (', end='')              # print open parenthesis

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

# Test
tour = PreorderPrintIndentedTour(A)
tour.execute()
print("------------")
tour2 = PreorderPrintIndentedTour2(A)
tour2.execute()
print("-------------")
tour3 = ParenthesizeTour(A)
tour3.execute()
```

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