Section B4 Trees

Trees adopt a few other commonly used definitions:

- **Parent:** A node in the tree with other connecting nodes is the parent of those nodes. Node B is the parent of nodes D, E, and F.
- **Child:** This is a node connected to its parent. Nodes B and C are children of node A, the parent and root node.
- **Sibling:** All nodes with the same parent are siblings. This makes the nodes B and C siblings.
- **Level:** The level of a node is the number of connections from the root node. The root node is at level 0. Nodes B and C are at level 1.
- **Height of a tree:** This is the number of levels in a tree. Our tree has a height of 4.
- **Depth:** The depth of a node is the number of edges from the root of the tree to that node. The depth of node H is 2.

Traversing a tree (also called "walking the tree") is the process of visiting each vertex precisely once, without repeating any vertex. At each vertex visited, we may retrieve, update or delete the vertex data.
Tree traversal always begins at the tree root and proceeds along connected edges to child vertices. That is, we cannot randomly access a vertex in a tree. There are three methods for traversing a tree:

1. Pre-order Traversal
2. In-order Traversal
3. Post-order Traversal

Pre-order Traversal
Pre-order traversal reads data from the vertex first, then from the left subtree, and finally from the right subtree.

```
Until all vertices in the current tree are traversed:

    - traverse recursively over the left subtree,
    - visit the current vertex, and then
    - traverse recursively over the right subtree.
```

Post-order Traversal

```
Until all vertices have been visited:

    - recursively traverse the current vertex's left subtree,
    - recursively traverse the current vertex's right subtree, and then
    - visit the current vertex.
```

```
class TreeNode:
    def __init__(self, value=0, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right
def pre_order_traversal(node):
    """Pre-order traversal order: Root -> Left -> Right"""
    if node is None:
        return  # Return if the node is empty/null
    print(node.value, end=' ')
    pre_order_traversal(node.left)
    pre_order_traversal(node.right)
def in_order_traversal(node):
    """In-order traversal order: Left -> Root -> Right """
    if node is None:
        return  # Return if the node is empty/null
    in_order_traversal(node.left)
    print(node.value, end=' ')
    in_order_traversal(node.right)
def post_order_traversal(node):
    """Post-order traversal order: Left -> Right -> Root"""
    if node is None:
        return  # Return if the node is empty/null
    post_order_traversal(node.left)
    post_order_traversal(node.right)
    print(node.value, end=' ')
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
pre_order_traversal(root)
in_order_traversal(root)
post_order_traversal(root)
```

Iterable - Looping over Objects in a Container

The following function adds a useful dunder method to facilitate in-order printing of the contents of a tree.

One helpful addition we can make to this class is to make it *iterable*. An iterable type allows Python to loop over the object in much the same way as lists, tuples, sets, dictionaries and strings can be looped over. In simple terms, a container is iterable when it has data or values and we perform an iteration over it to get elements one by one. To make our TreeVertex class iterable, we must add a *dunder method* method ***__iter__***.

When considering iteration of a tree, we have options on whether we should traverse the tree pre-order, in-order or post-order. In this case we provide in-order traversal using the yield statement

```
def __iter__(self):
    '''
    Create an iterator for this class. When repeatedly called
    the function will return the values of the graph starting from
    the left-most leaf through to the rightmost lead traversing all
    of the vertices according to their parent child relationship.
    '''
    if self._left:
        for leaf in self._left:
            yield leaf

    yield self._value

    if self._right:
        for leaf in self._right:
            yield leaf
```

This function uses the yield statement to return from execution of the function to the caller while preserving sufficient state that when the function is called again, execution commences from the next statement after the yield. It's a popular means to implement iteration in Python

Binary Search Trees

class TreeVertex

```
class TreeVertex:
    def __init__ (self, value = None):
        # User domain payload of the Vertex
        self._value = value
        # Left and right sided children
        self._left = None
        self._right = None
    def get_value (self):
        return self._value
    def set_value (self, value):
        self._value = value
    def get_left (self):
        return self._left
    def set_left (self, new_left):
        self._left = new_left
    def get_right (self):
        return self._right
    def set_right (self, new_right):
        self._right = new_right
```

```
def insert(self, value):
    '''Recursive insert of a new value as a leaf vertex.'''
    if value == self._value:
        # Nothing to insert
        return self
    elif value < self._value:
        if self._left is None:
            self.set_left (TreeVertex(value))
        else:
            self._left.insert(value)
    else:
        if self._right is None:
            self.set_right (TreeVertex(value))
        else:                 self._right.insert(value)
```

```
def num_children(self):
    '''
    Convenience function that returns the number of children. A more general function
    would return the degree of the vertex but, since we don't hold the parent vertex,
    it's not possible to calculate how many edges this vertex has.
    '''
    kids = 0
    if self._left is not None:
        kids += 1
    if self._right is not None:
        kids += 1
    return kids
def only_child(self):
    '''
    For vertices with one child, this function carelessly returns the only child.
    '''
    if self._left is None:
        return self._right
    else:
        return self._left
def min_value(self):
    ''' Returns the minimum value of the tree and its left subtree.'''
    if self._left is None:
        return self._value
    else:
        return self._left.min_value()
def height (self):
    '''Returns the height of the left and right subtrees.'''
    left_height, right_height = 0, 0
    if self._left is not None:
        left_height = 1 + max(self._left.height())
    if self._right is not None:
        right_height = 1 + max(self._right.height())
    return left_height, right_height
def __str__(self):
    return self._value

def __iter__(self):
    '''
    Create an iterator for this class. When repeatedly called
    the function will return the values of the graph starting from
    the left-most leaf through to the rightmost lead traversing all
    of the vertices according to their parent child relationship.
    '''
    if self._left:
        for leaf in self._left:
            yield leaf

    yield self._value

    if self._right:
        for leaf in self._right:
            yield leaf
```

class BinarySearchTree

```
class BinarySearchTree:
    def __init__(self):
        self._root = None

    def print_values(self):
        '''
        Print the values of the tree vertices "in-order". That is,
        from the smallest value to the largest by traversing the tree.
        '''
        if self._root:
            for v in self._root:
                print (v, end = " ")
            print()
        else:
            print("empty root")

    def insert_value (self, value):
        '''
        Insert a value at its correct location in the tree. Recall
        that values are inserted as leaf nodes. Make sure to handle
        the case where the root vertex has not been set.
        '''
        if self._root is None:
            self._root = TreeVertex(value)
        else:
            self._root.insert(value)
```

```
def search_value (self, value):
    '''
    Return True when the value is found in the tree. This is a convenience
    function that wraps search_vertex.
    '''
    found, vertex, parent = self.search_vertex (value)
    return found

def search_vertex (self, value):
    '''
    Return True and the Vertex of the tree if found otherwise False.
    As a convenience for deleting vertices, we also return the parent
    of the vertex.
    '''
    found = False
    previous_vertex = None
    current_vertex = self._root

    while not found and current_vertex is not None:
        if current_vertex.get_value() == value:
            found = True
        elif current_vertex.get_value() > value:
            previous_vertex = current_vertex
            current_vertex = previous_vertex.get_left()
        else:
            previous_vertex = current_vertex
            current_vertex = previous_vertex.get_right()

    return found, current_vertex, previous_vertex
```

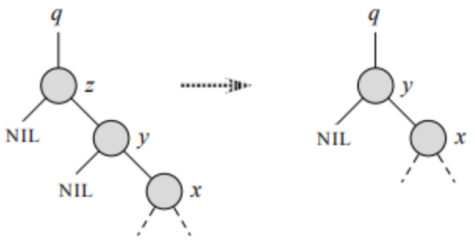
```
def delete_value (self, value):
    '''
    Delete the value from the tree handling each of the three cases discussed:
        - the vertex is a leaf,
        - the vertex has one child, and
        - the vertex has two children.
    Return True if the value was found and deleted otherwise False.
    '''
    found, vertex, parent = self.search_vertex (value)

    if found:

        children = vertex.num_children()
        if children == 0:
            if vertex.get_value() > parent.get_value():
                parent.set_right(None)
            else:
                parent.set_left(None)
        elif children == 1:
            if parent.get_left() == vertex:
                parent.set_left(vertex.only_child())
            else:
                parent.set_right(vertex.only_child())
        else:
            # Two children - in this case, the in-order successor of the right
            # sub tree replaces position with this vertex value and the in-order
            # successor is deleted.
            in_order_successor = vertex.get_right().min_value()
            self.delete_value(in_order_successor)
            vertex.set_value(in_order_successor)

    return found
```

Deleting Vertices

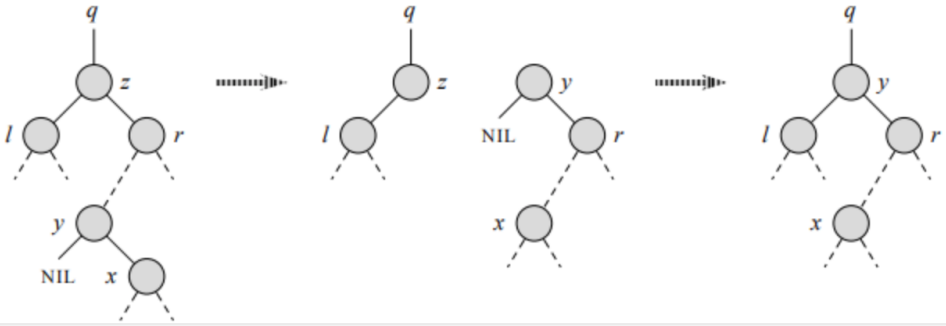
There are three cases to consider in deleting a node 'z':

(a) If 'z' is a leaf node, deletion occurs simply by setting the parent's (left or right) child relationship to `None`.

(b) If 'z' has a single child, that child gets elevated to become the child of 'z's parent, and



(c) If 'z' has two children we have more complex situation best illustrated with a picture:



In case (c), we are forced to consider which of the vertices will replace 'z' by looking for the "in-order successor" of 'z' - that is, the smallest value in the

right tree, in order to make this the replacement for 'z'.
We must note:

- because we are not storing a reference in each vertex to its parent, we must carefully consider how to implement vertex deletion, and
- as vertices are inserted and deleted, the tree becomes unbalanced so that some traversal operations (for search, insert or delete) will descend to a greater depth from the root and we start to lose O(log n) performance for these operations.

Unbalanced Trees
We can define a *balance factor* "bf" as:

```
bf = height of left subtree - height of right subtree
```

Thus a tree is height balanced if the balance factor at every node of the tree is -1, 0 or 1.
Let's extend the Vertex class to calculate the *height* of its subtrees:

```
def height (self):
      '''
      Returns the height of the left and right subtrees.
      '''
      left_height, right_height = 0, 0
      if self.left is not None:
          left_height = 1 + max(self.left.height())
      if self.right is not None:
          right_height = 1 + max(self.right.height())
      return left_height, right_height
```

Rebalancing
When faced with an unbalanced tree and deteriorating performance, there are essentially three choices available:

1. extract the data in-order and rebuild the tree in a balanced fashion,
2. monitor the balance of the tree and intervene to rebalance by rotating vertices at some imbalance threshold, or
3. implement a tree that self-balances during insertion and deletion.

Rotating vertices in the tree is a technique that changes the structure of the tree without changing the ordering of the data. Its effect is to decrease the height by moving smaller subtrees down and larger subtrees up.

Self-Balancing Trees
Self-balancing trees have been an area of active research in computer science with many algorithms offering advantages and disadvantages in search of performance that approaches average-case time complexity of O(log n) for the three operations on the BST - search, insert and delete.
For self-balancing binary search trees, the most commonly taught tree data structures are:

- AVL Trees (named after the inventors Adelson-Velsky and Landis),

- Red-Black Trees (an imperfectly balanced tree so named because each vertex stores extra data on the "coloured" balance state of its vertices), and
- B-Tree (a self-balancing tree that allows vertices with more than two children and is well suited for reading and writing large blocks of data).