Recitation 6

Binary Trees

A **binary tree** is a tree (a connected graph with no cycles) of **binary nodes**: a linked node container, similar to a linked list node, having a constant number of fields:

- a pointer to an item stored at the node,
- a pointer to a **parent node** (possibly `None`),
- a pointer to a **left child** node (possibly `None`), and
- a pointer to a **right child** node (possibly `None`).

```python
class Binary_Node:
    def __init__(A, x):                     # O(1)
        A.item   = x
        A.left   = None
        A.right  = None
        A.parent = None
        # A.subtree_update()                # wait for R07!
```

Why is a binary node called "binary"? In actuality, a binary node can be connected to **three** other nodes (its parent, left child, and right child), not just two. However, we will differentiate a node's parent from it's children, and so we call the node "binary" based on the number of children the node has.

A binary tree has one node that that is the **root** of the tree: the only node in the tree lacking a parent. All other nodes in the tree can reach the root of the tree containing them by traversing parent pointers. The set of nodes passed when traversing parent pointers from node `<X>` back to the root are called the **ancestors** for `<X>` in the tree. The **depth** of a node `<X>` in the subtree rooted at `<R>` is the length of the path from `<X>` back to `<R>`. The **height** of node `<X>` is the maximum depth of any node in the subtree rooted at `<X>`. If a node has no children, it is called a **leaf**.

Why would we want to store items in a binary tree? The difficulty with a linked list is that many linked list nodes can be $O(n)$ pointer hops away from the the head of the list, so it may take $O(n)$ time to reach them. By contrast, as we've seen in earlier recitations, it is possible to construct a binary tree on n nodes such that no node is more than $O(\log n)$ pointer hops away from the root, i.e., there exist binary trees with logarithmic height. The power of a binary tree structure is if we can keep the height h of the tree low, i.e., $O(\log n)$, and only perform operations on the tree that run in time on the order of the height of the tree, then these operations will run in $O(h) = O(\log n)$ time (which is much closer to $O(1)$ than to $O(n)$).

Traversal Order

The nodes in a binary tree have a natural order based on the fact that we distinguish one child to be left and one child to be right. We define a binary tree's **traversal order** based on the following implicit characterization:

- every node in the left subtree of node `<X>` comes **before** `<X>` in the traversal order; and
- every node in the right subtree of node `<X>` comes **after** `<X>` in the traversal order.

Given a binary node `<A>`, we can list the nodes in `<A>`'s subtree by recursively listing the nodes in `<A>`'s left subtree, listing `<A>` itself, and then recursively listing the nodes in `<A>`'s right subtree. This algorithm runs in $O(n)$ time because every node is recursed on once doing constant work.

```
def subtree_iter(A):                        # O(n)
    if A.left:   yield from A.left.subtree_iter()
    yield A
    if A.right:  yield from A.right.subtree_iter()
```

Right now, there is no semantic connection between the items being stored and the traversal order of the tree. Next time, we will provide two different semantic meanings to the traversal order (one of which will lead to an efficient implementation of the Sequence interface, and the other will lead to an efficient implementation of the Set interface), but for now, we will just want to preserve the traversal order as we manipulate the tree.

Tree Navigation

Given a binary tree, it will be useful to be able to navigate the nodes in their traversal order efficiently. Probably the most straight forward operation is to find the node in a given node's subtree that appears first (or last) in traversal order. To find the first node, simply walk left if a left child exists. This operation takes $O(h)$ time because each step of the recursion moves down the tree. Find the last node in a subtree is symmetric.

```
def subtree_first(A):                       # O(h)
    if A.left:  return A.left.subtree_first()
    else:       return A

def subtree_last(A):                        # O(h)
    if A.right: return A.right.subtree_last()
    else:       return A
```

Given a node in a binary tree, it would also be useful too find the next node in the traversal order, i.e., the node's **successor**, or the previous node in the traversal order, i.e., the node's **predecessor**. To find the successor of a node `<A>`, if `<A>` has a right child, then `<A>`'s successor will be the first node in the right child's subtree. Otherwise, `<A>`'s successor cannot exist in `<A>`'s subtree, so we walk up the tree to find the lowest ancestor of `<A>` such that `<A>` is in the ancestor's left subtree.

In the first case, the algorithm only walks down the tree to find the successor, so it runs in $O(h)$ time. Alternatively in the second case, the algorithm only walks up the tree to find the successor, so it also runs in $O(h)$ time. The predecessor algorithm is symmetric.

```
def successor(A):                      # O(h)
    if A.right: return A.right.subtree_first()
    while A.parent and (A is A.parent.right):
        A = A.parent
    return A.parent

def predecessor(A):                    # O(h)
    if A.left:  return A.left.subtree_last()
    while A.parent and (A is A.parent.left):
        A = A.parent
    return A.parent
```

Dynamic Operations

If we want to add or remove items in a binary tree, we must take care to preserve the traversal order of the other items in the tree. To insert a node `<B>` before a given node `<A>` in the traversal order, either node `<A>` has a left child or not. If `<A>` does not have a left child, than we can simply add `<B>` as the left child of `<A>`. Otherwise, if `<A>` has a left child, we can add `<B>` as the right child of the last node in `<A>`'s left subtree (which cannot have a right child). In either case, the algorithm walks down the tree at each step, so the algorithm runs in $O(h)$ time. Inserting after is symmetric.

```
def subtree_insert_before(A, B):       # O(h)
    if A.left:
        A = A.left.subtree_last()
        A.right, B.parent = B, A
    else:
        A.left,  B.parent = B, A
    # A.maintain()                     # wait for R07!

def subtree_insert_after(A, B):        # O(h)
    if A.right:
        A = A.right.subtree_first()
        A.left,  B.parent = B, A
    else:
        A.right, B.parent = B, A
    # A.maintain()                     # wait for R07!
```

To extract the item contained in a given node from its binary tree, there are two cases based on whether the node storing the item is a leaf. If the node is a leaf, then we can simply clear the child pointer from the node's parent and return the node. Alternatively, if the node is not a leaf, we can swap the node's item with the item in the node's successor or predecessor down the tree until the item is in a leaf which can be removed. Since swapping only occurs down the tree, again this operation runs in $O(h)$ time.

```
def subtree_extract(A):                 # O(h)
    if A.left or A.right:               # A is not a leaf
        if A.left:  B = A.predecessor()
        else:       B = A.successor()
        A.item, B.item = B.item, A.item
        return B.subtree_extract()
    if A.parent:                        # A is a leaf
        if A.parent.left is A:  A.parent.left  = None
        else:                   A.parent.right = None
        # A.parent.maintain()          # wait for R07!
    return A
```

Rotations

As we add or remove nodes to our tree, it is possible that our tree will become imbalanced. We will want to change the structure of the tree without changing its traversal order, in the hopes that we can make the tree's structure more balanced. We can change the structure of a tree using a local operation called a **rotation**. A rotation takes a subtree that locally looks like one the following two configurations and modifies the connections between nodes in $O(1)$ time to transform it into the other configuration.

```
              ______<D>__      rotate_right(<D>)     __<B>______
           __<B>__     <E>            =>            <A>     __<D>__
          <A>   <C>   / \                           / \    <C>   <E>
          / \   / \  /___\            <=           /___\   / \   / \
         /___\ /___\           rotate_left(<B>)          /___\ /___\
```

This operation preserves the traversal order of the tree while changing the depth of the nodes in subtrees <A> and <E>. Next time, we will use rotations to enforce that a balanced tree stays balanced after inserting or deleting a node.

```
def subtree_rotate_right(D):         def subtree_rotate_left(B):  # O(1)
    assert D.left                        assert B.right
    B, E = D.left, D.right               A, D = B.left, B.right
    A, C = B.left, B.right               C, E = D.left, D.right
    D, B = B, D                          B, D = D, B
    D.item, B.item = B.item, D.item      B.item, D.item = D.item, B.item
    B.left, B.right = A, D               D.left, D.right = B, E
    D.left, D.right = C, E               B.left, B.right = A, C
    if A: A.parent = B                   if A: A.parent = B
    if E: E.parent = D                   if E: E.parent = D
    # B.subtree_update()                 # B.subtree_update()     # wait for R07!
    # D.subtree_update()                 # D.subtree_update()     # wait for R07!
```

Exercise: Given an array of items $A = (a_0, \ldots, a_{n-1})$, describe a $O(n)$-time algorithm to construct a binary tree T containing the items in A such that (1) the item stored in the i^{th} node of T's traversal order is item a_i, and (2) T has height $O(\log n)$.

Solution: Build T by storing the middle item in a root node, and then recursively building the remaining left and right halves in left and right subtrees. This algorithm satisfies property (1) by definition of traversal order, and property (2) because the height roughly follows the recurrence $H(n) = 1 + 2T(n/2)$. The algorithm runs in $O(n)$ time because every node is recursed on once doing constant work.

```
def build_subtree(A, i, j):
    c = (i + j) // 2
    root = Binary_Node(A[c])
    if i < c:                   # needs to store more items in left subtree
        root.left = build_subtree(A, i, c - 1)
        root.left.parent = root
    if c < j:                   # needs to store more items in right subtree
        root.right = build_subtree(A, c + 1, j)
        root.right.parent = root
    return root
```

Exercise: Argue that the following iterative procedure to return the the nodes of a tree in traversal order takes $O(n)$ time.

```
def tree_iter(R):    # R is the root of a binary tree
    node = R.subtree_first()
    while node:
        yield node
        node = node.successor()
```

Solution: This procedure walks around the tree traversing each edge of the tree twice: once going down the tree, and once going back up. Then because the number of edges in a tree is one fewer than the number of nodes, the traversal takes $O(n)$ time.