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`).

```
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 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.

```
1    def subtree_iter(A):                    # O(n)
2        if A.left:   yield from A.left.subtree_iter()
3        yield A
4        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.

```
1    def subtree_first(A):                   # O(h)
2        if A.left:  return A.left.subtree_first()
3        else:       return A
4
5    def subtree_last(A):                    # O(h)
6        if A.right: return A.right.subtree_last()
7        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 delete 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_delete(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_delete()
        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
```

Binary Node Full Implementation

```
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!

    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()

    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

    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

```

```
34      def subtree_insert_before(A, B):        # O(h)
35          if A.left:
36              A = A.left.subtree_last()
37              A.right, B.parent = B, A
38          else:
39              A.left,  B.parent = B, A
40          # A.maintain()                      # wait for R07!
41
42      def subtree_insert_after(A, B):         # O(h)
43          if A.right:
44              A = A.right.subtree_first()
45              A.left,  B.parent = B, A
46          else:
47              A.right, B.parent = B, A
48          # A.maintain()                      # wait for R07!
49
50      def subtree_delete(A):                  # O(h)
51          if A.left or A.right:
52              if A.left:  B = A.predecessor()
53              else:       B = A.successor()
54              A.item, B.item = B.item, A.item
55              return B.subtree_delete()
56          if A.parent:
57              if A.parent.left is A:  A.parent.left  = None
58              else:                   A.parent.right = None
59               # A.parent.maintain()          # wait for R07!
60          return A
```

Top-Level Data Structure

All of the operations we have defined so far have been within the `Binary_Tree` class, so that they apply to any subtree. Now we can finally define a general Binary Tree data structure that stores a pointer to its root, and the number of items it stores. We can implement the same operations with a little extra work to keep track of the root and size.

```
1   class Binary_Tree:
2       def __init__(T, Node_Type = Binary_Node):
3           T.root = None
4           T.size = 0
5           T.Node_Type = Node_Type
6
7       def __len__(T): return T.size
8       def __iter__(T):
9           if T.root:
10              for A in T.root.subtree_iter():
11                  yield A.item
```

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 $H(n) = 1 + H(n/2)$. The algorithm runs in $O(n)$ time because every node is recursed on once doing constant work.

```
def build(X):
    A = [x for x in X]
    def build_subtree(A, i, j):
        c = (i + j) // 2
        root = self.Node_Type(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
    self.root = build_subtree(A, 0, len(A)-1)
```

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

```
def tree_iter(T):
    node = T.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.

Application: Set

To use a Binary Tree to implement a Set interface, we use the traversal order of the tree to store the items sorted in increasing key order. This property is often called the **Binary Search Tree Property**, where keys in a node's left subtree are less than the key stored at the node, and keys in the node's right subtree are greater than the key stored at the node. Then finding the node containing a query key (or determining that no node contains the key) can be done by walking down the tree, recursing on the appropriate side.

Exercise: Make a Set Binary Tree (Binary Search Tree) by inserting student-chosen items one by one, then searching and/or deleting student-chosen keys one by one.

```
class BST_Node(Binary_Node):
    def subtree_find(A, k):                 # O(h)
        if k < A.item.key:
            if A.left:  return A.left.subtree_find(k)
        elif k > A.item.key:
            if A.right: return A.right.subtree_find(k)
        else:           return A
        return None

    def subtree_find_next(A, k):            # O(h)
        if A.item.key <= k:
            if A.right: return A.right.subtree_find_next(k)
            else:       return None
        elif A.left:
            B = A.left.subtree_find_next(k)
            if B:       return B
        return A

    def subtree_find_prev(A, k):             # O(h)
        if A.item.key >= k:
            if A.left:  return A.left.subtree_find_prev(k)
            else:       return None
        elif A.right:
            B = A.right.subtree_find_prev(k)
            if B:       return B
        return A

    def subtree_insert(A, B):                # O(h)
        if B.item.key < A.item.key:
            if A.left:  A.left.subtree_insert(B)
            else:       A.subtree_insert_before(B)
        elif B.item.key > A.item.key:
            if A.right: A.right.subtree_insert(B)
            else:       A.subtree_insert_after(B)
        else:    A.item = B.item
```

```
class Set_Binary_Tree(Binary_Tree):  # Binary Search Tree
    def __init__(self): super().__init__(BST_Node)

    def iter_order(self): yield from self

    def build(self, X):
        for x in X: self.insert(x)

    def find_min(self):
        if self.root:   return self.root.subtree_first().item

    def find_max(self):
        if self.root:   return self.root.subtree_last().item

    def find(self, k):
        if self.root:
            node = self.root.subtree_find(k)
            if node:    return node.item

    def find_next(self, k):
        if self.root:
            node = self.root.subtree_find_next(k)
            if node:    return node.item

    def find_prev(self, k):
        if self.root:
            node = self.root.subtree_find_prev(k)
            if node:    return node.item

    def insert(self, x):
        new_node = self.Node_Type(x)
        if self.root:
            self.root.subtree_insert(new_node)
            if new_node.parent is None: return False
        else:
            self.root = new_node
        self.size += 1
        return True

    def delete(self, k):
        assert self.root
        node = self.root.subtree_find(k)
        assert node
        ext = node.subtree_delete()
        if ext.parent is None:  self.root = None
        self.size -= 1
        return ext.item
```

6.006 Introduction to Algorithms

Spring 2020