R6 Binary Trees☆

3.23 https://github.com/GUMI-21/MIT6.006_note
• 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).

ancestors
The *depth* of a node in the subtree rooted at x is the length of the path from x back to R
If a node has no children, it is called a *leaf*.

Traversal Order

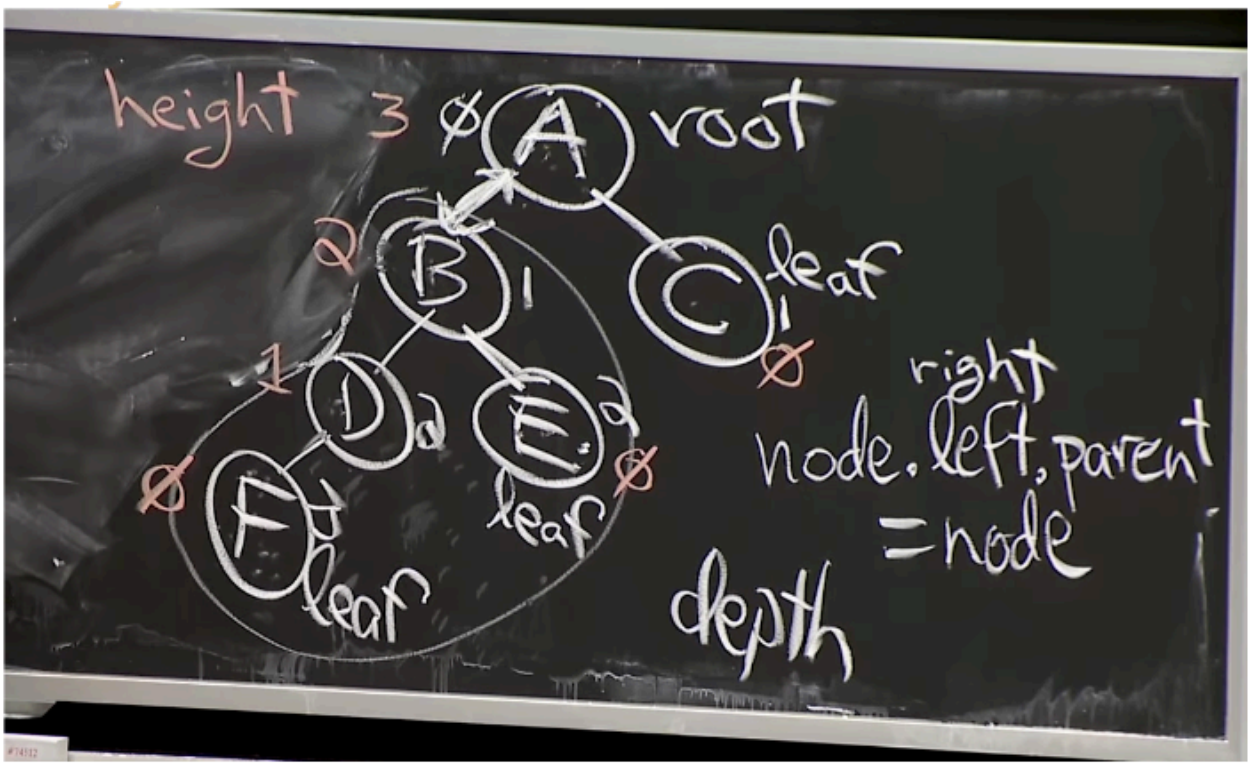
- every node in the left subtree of node x comes before x in the traversal order; and
- every node in the right subtree of node x comes after x in the traversal order.
  1.list the nodes in A's subtree by recursively listing the nodes in A's left subtree.
  2.listing A itself.
  3.list the nodes in A's right subtree.

Tree Navigation

```
def subtree_first(A):
    if A.left: return A.left.subtree_first()
    else: return A
def subtree_last(A):
    if A.right: return A.right.subtree_last()
    else: return A
```

- *successor*
  the next node in the traversal order
  -if node has right child -> return node.right's subtree's traversal order first
  -else go up until find nod is left subtree & return node (the root of left subtree)
- *predecessor*
  the previous node in the traversal
  -if A has left subtree -> A is after all left subtree, reutrn A.left.subtree_last()
  -else A is the last of the subtree of A.parent. go up until A is A.parent.right -> return A.parent.
  if A is A.parent.left means A is prev of A.parent.

```
def successor(A):
    if A.right: return A.right.subtree_first()  # navigation of traversal
order
    while A.parent and (A is A.parent.right):  # until A is
A.parent.left,means A is before parent so return A.parent
        A = A.parent
    return A.parent
def predecessor(A):
    if A.left: return A.left.subtree_last()
    while A.parent and (A is A.parent.left): # until A is A.parent.right,means
A is after parent so reuturn A.parent
        A = A.parent
    return A.parent
```



F-D-B-E-*A*-C

Dynamic Operations

preserve the traversal order of the tree.

- *insert node B before a given node A*
  -if A does not have a left child, then we can simply add B as the left child of A
  -else if A has left child, add B as the right child of the last node in A's left subtree.
  runtime O(h)
  insert after is symmetric

```
def subtree_insert_before(A, B):
    if A.left:
        A = A.left.subtree_last()   # find the last of A's left subtree, and
the node must have not right child,  -> node set on right
        A.right, B.parent = B, A
    else:
        A.left, B.parent = B, A  # is A doesn't have left child -> set on

def subtree_insert_after(A, B):
    if A.right
       A = A.right.subtree_firt() # find the first node of A's right subtree,
and the node must have not left child -> node set on left
       A.left, B.parent = B,A
    else:
       A.right, B.parent = B,A # direct set on right
```

- *delete the item contained in a given node*
  -if the ndoe is the leaf, then directly erase and return the node
  -else 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 directly.
  O(h) time

```
def subtree_delete(A):
    if A.left or A.right:
        if A.left: B = A.predecessor()   # go down to leaf, swap with
successor or predecessor.
        else B = A.successor()
        A.item, B.item = B.item, A.item
        return B.subtree_delete()
    if A.parent:  # delete directly when ndoe is leaf
        if A.parent.left is A: A.parent.left = None
        else: A.parent.right = None
        A.parent.maintain()
    return A
```

Exercise

Given an array of items A = (a0, . . . , an−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 ai, and (2) T has height O(log n).

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.