CS 315 - Day 24, Binary Search Trees

Last class we covered almost all the algorithms associated with BSTs, except Deleting a node. As the text states, "it [deletion] can be a bit tricky."

Conceptually, deleting node z from binary search tree T has three cases:

1. If z has no children, just remove it by changing z's parent to point to NIL instead of to z.
2. If z has just one child, then make that child take z's position in the tree by changing z's parent to point to z's child instead of to z, dragging the child's subtree along.
3. If z has two children, then find z's successor y and replace z by y in the tree. y must be in z's right subtree and have no left child. The rest of z's original right subtree becomes y's new right subtree, and z's left subtree becomes y's new left subtree.

   This case is a little tricky because the exact sequence of steps taken depends on whether y is z's right child.

Explored in the HW, if y did have a left child, that left child would be z's Successor!

Because this third case is "tricky," we introduce a helper routine, `Transplant`:

TRANSPLANT(T, u, v) replaces the subtree rooted at u by the subtree rooted at v:

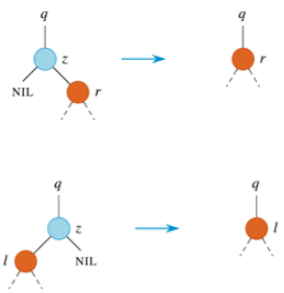
- Makes u's parent become v's parent (unless u is the root, in which case it makes v the root).
- u's parent gets v as either its left or right child, depending on whether u was a left or right child.
- Doesn't update $v.left$ or $v.right$, leaving that up to TRANSPLANT's caller.

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2      T.root = v
3  elseif u == u.p.left
4      u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7      v.p = u.p
```

And the implementation for Tree-Delete actually has four cases, the third is split, depending on whether or note y is z's right child:

TREE − DELETE(T, z) has four cases when deleting node z from binary search tree T :

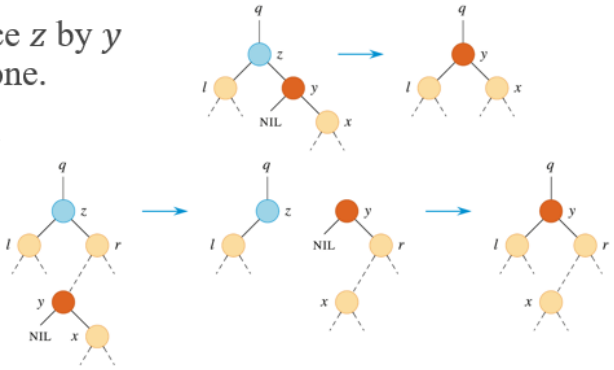
- If z has no left child, replace z by its right child. The right child may or may not be NIL. (If z's right child is NIL, then this case handles the situation in which z has no children.)
- If z has just one child, and that child is its left child, then replace z by its left child.



Otherwise, z has two children. Find z's successor y, y must lie in z's right subtree and have no left child (the solution to Exercise 12.2-5 on page 12-13 of this manual shows why).

Goal is to replace z by y, splicing y out of its current location.

- If y is z's right child, replace z by y and leave y's right child alone.
- Otherwise, y lies within z's right subtree but is not the root of this subtree. Replace y by its own right child. Then replace z by y.



TREE-DELETE(T, z)

```
1  if z.left == NIL
2      TRANSPLANT(T, z, z.right)          // replace z by its right child
3  elseif z.right == NIL
4      TRANSPLANT(T, z, z.left)           // replace z by its left child
5  else y = TREE-MINIMUM(z.right)         // y is z's successor
6      if y ≠ z.right                     // is y farther down the tree?
7          TRANSPLANT(T, y, y.right)      // replace y by its right child
8          y.right = z.right              // z's right child becomes
9          y.right.p = y                  //       y's right child
10     TRANSPLANT(T, z, y)                // replace z by its successor y
11     y.left = z.left                    // and give z's left child to y,
12     y.left.p = y                       //       which had no left child
```

Note that the last three lines execute when z has two children, regardless of whether y y is z's right child.

In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. On the given BST, work through the four examples (starting over after each delete):

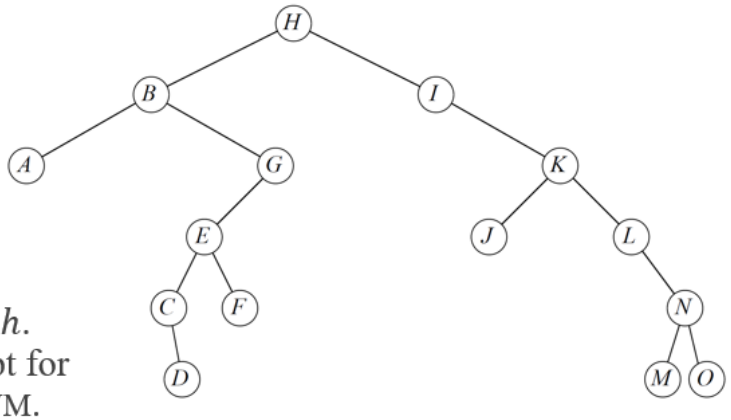
On this binary search tree T, run the following 4 examples:

TREE $-$ DELETE(T, I)

TREE $-$ DELETE(T, G)

TREE $-$ DELETE(T, K)

TREE $-$ DELETE(T, B)



Time

$O(h)$, on a tree of height h. Everything is $O(1)$ except for the call to TREE-MINIMUM.

Foreshadowing for Chapter 13 and Red-Black Trees:

[We've been analyzing running time in terms of h (the height of the binary search tree), instead of n (the number of nodes in the tree).

- *Problem: Worst case for binary search tree is $\Theta(n)$—no better than linked list.*
- *Solution: Guarantee small height (balanced tree)—$h = O(\lg n)$.*

In later chapters, by varying the properties of binary search trees, we will be able to analyze running time in terms of n.

- *Method: Restructure the tree if necessary. Nothing special is required for querying, but there may be extra work when changing the structure of the tree (inserting or deleting).*

Red-black trees are a special class of binary trees that avoids the worst-case behavior of $O(n)$ that we can see in "plain" binary search trees. Red-black trees are covered in detail in Chapter 13.]

May return to seats.

So, how would we use a BST to sort data?

Build the BST and then output an Inorder walk.

What is the asymptotic runtime for this? Building and then traversing? Think worst case.

What sort, which we have discussed in class, with $O(N \cdot \lg N)$ average case asymptotic run-time and $O(N^2)$ worst-case runtime, is reminiscent of using a BST to sort?

Hint: think of the "helper" routines.

The key value, left/right, smaller/larger → Quicksort!

Next, we will explore `binary_search_tree.ipynb`. Download it from the LMS and then open it in Google Colab.

Start by running, noting the examples given, and confirming that the output is correct.

Update your code with a new execution group.

In this execution group you will time both the building of the BST and then the Inorder traversal.

You should try for a variety of orders of magnitude of size N, starting with randomly distributed data inserted into the tree. Start small to debug and output your trees and traversals, to confirm correctness, including checking that the tree created satisfies the BST property.

The header for the function `inorder_tree_walk` is given here:

```
def inorder_tree_walk(self, x, func=print):
"""Run a function on all the nodes of the subtree rooted at node x in an
inorder tree walk.
Arguments:
x -- root of the subtree
func -- function to run on each node in the subtree.  If omitted, print.
"""
# Run func on the key of the root in between visiting the left and right
subtrees.
```

Note that if an input lambda function isn't provided, all the nodes will be output. If we want to time the traversals for large N, the output will be problematic.

You can suppress output by using a "no-op" function like: `func=lambda x: None`

Time, and calculate ratios, for the building and the traversals, both on random and sorted order.

Are you able to demonstrate a worst-case?

```
# @title Timing BST Sorting
if __name__ == "__main__":
  import numpy as np
  import time
  import math
  for i in range(1,5):
  # Define BST
    binary_tree_timing = BinarySearchTree()
    N = int(math.pow(10,i))
    array1 = np.arange(0,N,1)
    #np.random.shuffle(array1)
    start = time.time()
    for value in array1:
      binary_tree_timing.tree_insert(value)
    end = time.time()
    ratioLG=(end-start)/(N*math.log(N,2))
    ratioN2=ratio=(end-start)/(N*N)
    print(binary_tree_timing.is_BST())
    print("build: ",N,end-start,ratioLG,ratioN2)
    #print(binary_tree_timing)
    start = time.time()
    binary_tree_timing.inorder_tree_walk(binary_tree_timing.get_root(),func=l
ambda x: None)
    #binary_tree_timing.inorder_tree_walk(binary_tree_timing.get_root())
    end = time.time()
    ratio=(end-start)/(N*math.log(N,2))
    #ratio=ratio=(end-start)/(N*N)
    print("walk: ",N,end-start,ratio)
```