Exercise Session n. 9

Algorithms and Data Structures

We practice a bit with binary-search trees. All algorithms are based on trees defined by the following elementary structure:

```
class Node:
    def __init__(self,k):
        self.key = k
        self.parent = None
        self.left = None
        self.right = None
```

For the purpose of testing, you may use the following insertion algorithm.

```
def bst_insert(t, k):
    if t == None:
        return Node(k)
    x = t
    while True:
        if k <= x.key:
            if x.left == None:
                x.left = Node(k)
                x.left.parent = x
                return t
            x = x.left
        else:
            if x.right == None:
                x.right = Node(k)
                x.right.parent = x
                return t
            x = x.right
```

So, for example, if you wanted to create a BST containing 20 random integers between 1 and 100, you can write:

```
import random
t = None
for _ in range(20):
    t = bst_insert(t, random.randint(1,100))
```

Also for the purpose of testing, you might want to use the `print_binary_tree` function defined below:

```
class Canvas:
    def __init__(self,width):
        self.line_width = width
        self.canvas = []

    def put_char(self,x,y,c):
        if x < self.line_width:
            pos = y*self.line_width + x
            l = len(self.canvas)
            if pos < l:
                self.canvas[pos] = c
            else:
                self.canvas[l:] = [' ']*(pos - l)
                self.canvas.append(c)

    def print_out(self):
        i = 0
        sp = 0
        for c in self.canvas:
            if c != ' ':
                print(' '*sp, end='')
                print(c, end='')
                sp = 0
            else:
                sp += 1
            i = i + 1
            if i % self.line_width == 0:
                print('\n', end='')
                sp = 0
        if i % self.line_width != 0:
            print('\n', end='')

def print_binary_tree_r(t,x,y,canvas):
    max_y = y
    if t.left != None:
        x, max_y, lx, rx = print_binary_tree_r(t.left,x,y+2,canva
        x = x + 1
        for i in range(rx,x):
            canvas.put_char(i, y+1, '/')

    middle_l = x
    for c in str(t.key):
        canvas.put_char(x, y, c)
```

```
        x = x + 1
    middle_r = x

    if t.right != None:
        canvas.put_char(x, y+1, '\\')
        x = x + 1
        x0, max_y2, lx, rx = print_binary_tree_r(t.right,x,y+2,ca
        if max_y2 > max_y:
            max_y = max_y2
        for i in range(x,lx):
            canvas.put_char(i, y+1, '\\')
        x = x0

    return (x,max_y,middle_l,middle_r)

def print_tree_w(t,width):
    canvas = Canvas(width)
    print_binary_tree_r(t,0,0,canvas)
    canvas.print_out()

def print_tree(t):
    print_tree_w(t,20000)
```

The Height of a Binary Search Tree

Write an algorithm `bst_height(T)` that returns the height of a binary search tree T (i.e., rooted at T). The height of a BST is the maximal number of nodes on the path from the root to a leaf node.

Count In Range

Write an algorithm `bst_count_in_range(T,a,b)` that, given the root T of a binary search tree and two keys a and b, with $a \leq b$, returns the number of keys in T that are greater than a and less than b. Also, analyze the complexity of `bst_count_in_range(T,a,b)`.

Expected Height of a “Random” BST

Write a function `bst_expected_height(n,M)` computes and returns the average height of M binary search trees obtained by inserting n distinct keys in random order starting from an empty tree. *Hint:* you may use `random.shuffle` to randomize the order of a sequence. Plot a chart of the results you obtain from `bst_expected_height(n,M)` to then characterize the expected height of a random

BST as a function of n.