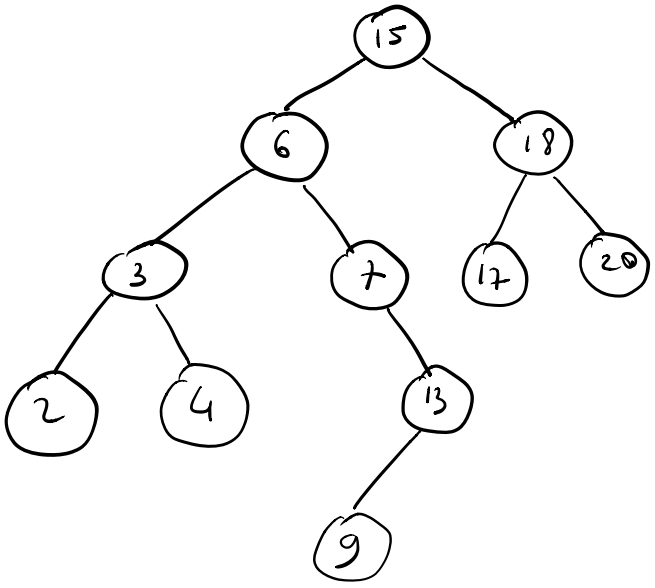
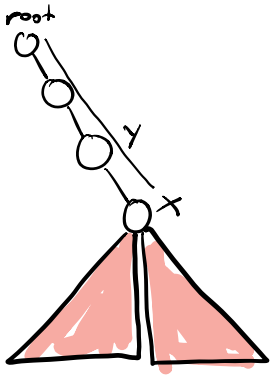
Exercises about trees

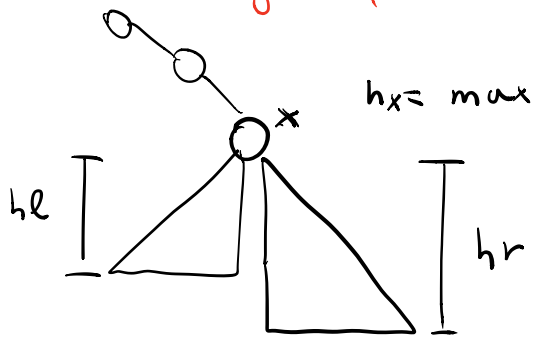


Visits

```
Visit(x)
    if x =/= NIL
        Visit(x.left)
        Process(x)
        Visit(x.right)
```

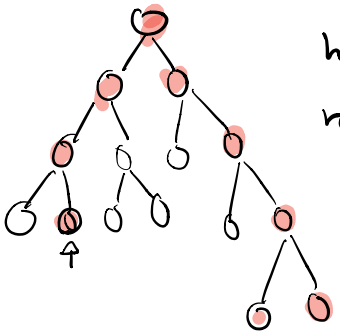


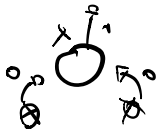
Compute height of a tree

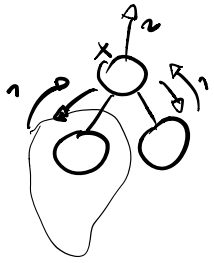


$h_x = \max(h_l, h_r) + 1$

```
height(x)
    if x == NIL
        return 0
    hl = height(x.left)
    hr = height(x.right)
    return max(hl, hr) + 1
```





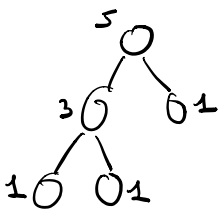


information go bottom up

Exercise

Given a tree, we would like to compute the subtree size of every node

Example,

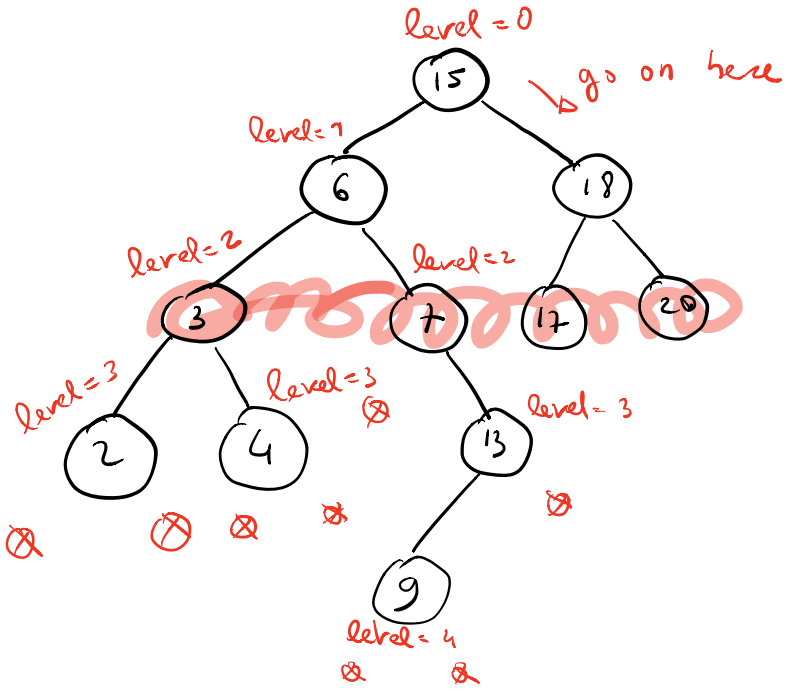


Solution

```
Subtree size (x)
    if x == NIL    return 0
    ssl = Subtree size (x.left)
    ssr = Subtree size (x.right)
    return   ssl + ssr + 1
```

Exercise

Given a tree, print the keys of every node at a given level k



level
0
1
2
3
4
:

```
PrintLevelk (x, k, level)        info that goes down to the leaves
    if x == NIL
        return
    if level == k
        print (x.key)
    PrintLevelk (x.left, k, level + 1)
    PrintLevelk (x.right, k, level + 1)
```

First call

PrintLevelk (root, 2, 0)

```
PrintLevelk(x, k)
    if x == NILL      return
    if k == 0         print(x.key)
    PrintLevelk(x.left, k-1)
    PrintLevelk(x.right, k-1)
```

Assume the tree is a BST, print keys at level k sorted

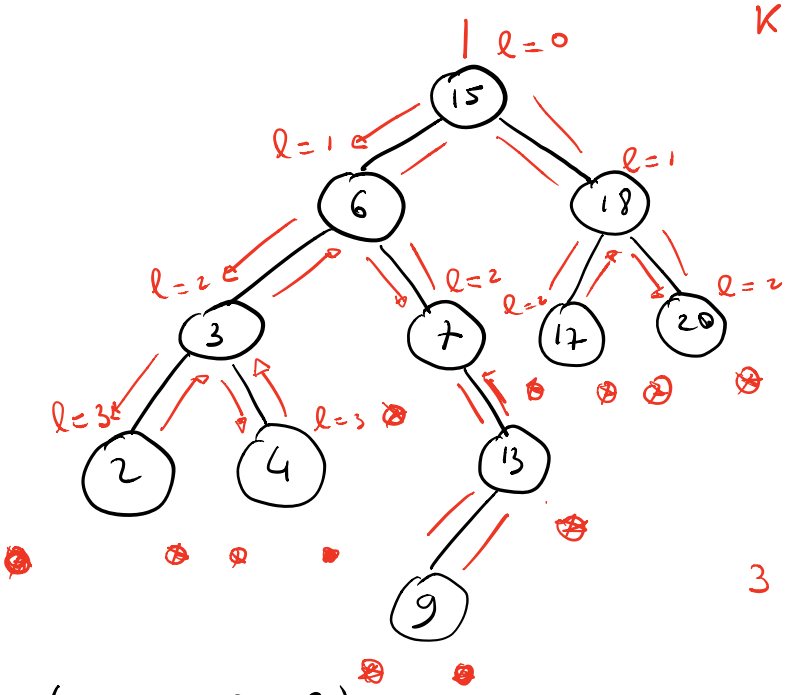
```
PrintLevelk(x, k)
    if x == NILL      return
    if k == 0         print(x.key)     // Process x
    PrintLevelk(x.left, k-1)           // visit left
    PrintLevelk(x.right, k-1)          // visit right
```

pre order visit but if we visit in inorder a BST we get the array sorted

```
PrintLevelk(x, k)
    if x == NILL      return
    PrintLevelk(x.left, k-1)           // visit left
    if k == 0         print(x.key)     // Process x
    PrintLevelk(x right  k-1)          // visit right
```

Some of your (incorrect) solution

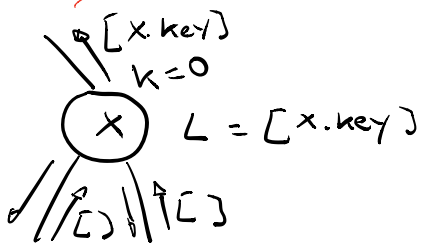
K = 2



3 3 7 7 17 17 20 20

```
p(x, k, level)
    if k == NIL          return
    p(x.left, k, level+1)
    if level == k
            print (x.key)
    p(x.right, k, level+1)
    if level == k
            print (x.key)
                        )
```

```
p(x, k, L)
    L = []
    if x == NIL  return
    if k == 0  L.append(x.key)
    p(x.left, k-1)
    p(x.right, k-1)
    return sorted(L)
```




```
p(x, k)
    if x == NIL  return []
    L = p(x.left, k-1)
    if k == 0  L.append(x.key)
    R = p(x.right, k-1)
    return  L.extend(R)
```

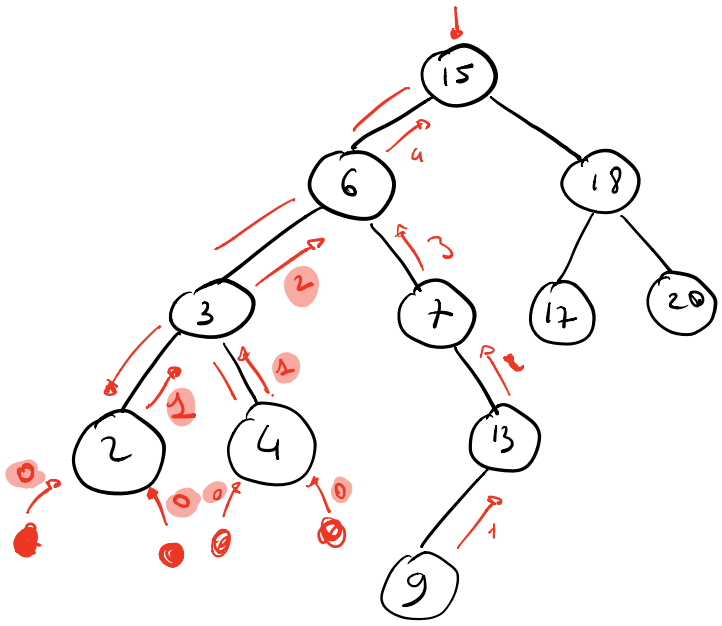
Exercise

Given a binary tree, print the key of each node x such that $x.key$ equals the subtree size of x.

```
f(x)
    if x == NIL  return 0
    sl = f(x.left)
    sr = f(x.right)
    sx = sl + sr + 1
    if x.key == sx  print(x.key)
    return sx
```

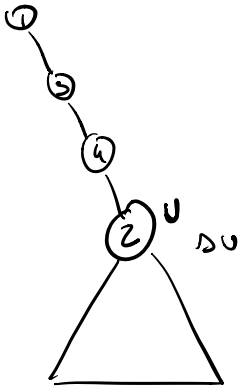
but print $u.key$ if its distance from leaf equals $u.key$

```
f(x)
    if x == NIL  return 0
    hl = f(x.left)
    hr = f(x.right)
    hx = max(hl, hr) + 1

    if hx == x.key
        print(x.key)

    return hx
```


15
6
18
3
7
17
20
2
4
13
9

Exercise

Given a tree, print any node u such that the sum of key on the path from root to u equals u's subtree size



$1 + 3 + 4 + 2 = 10$

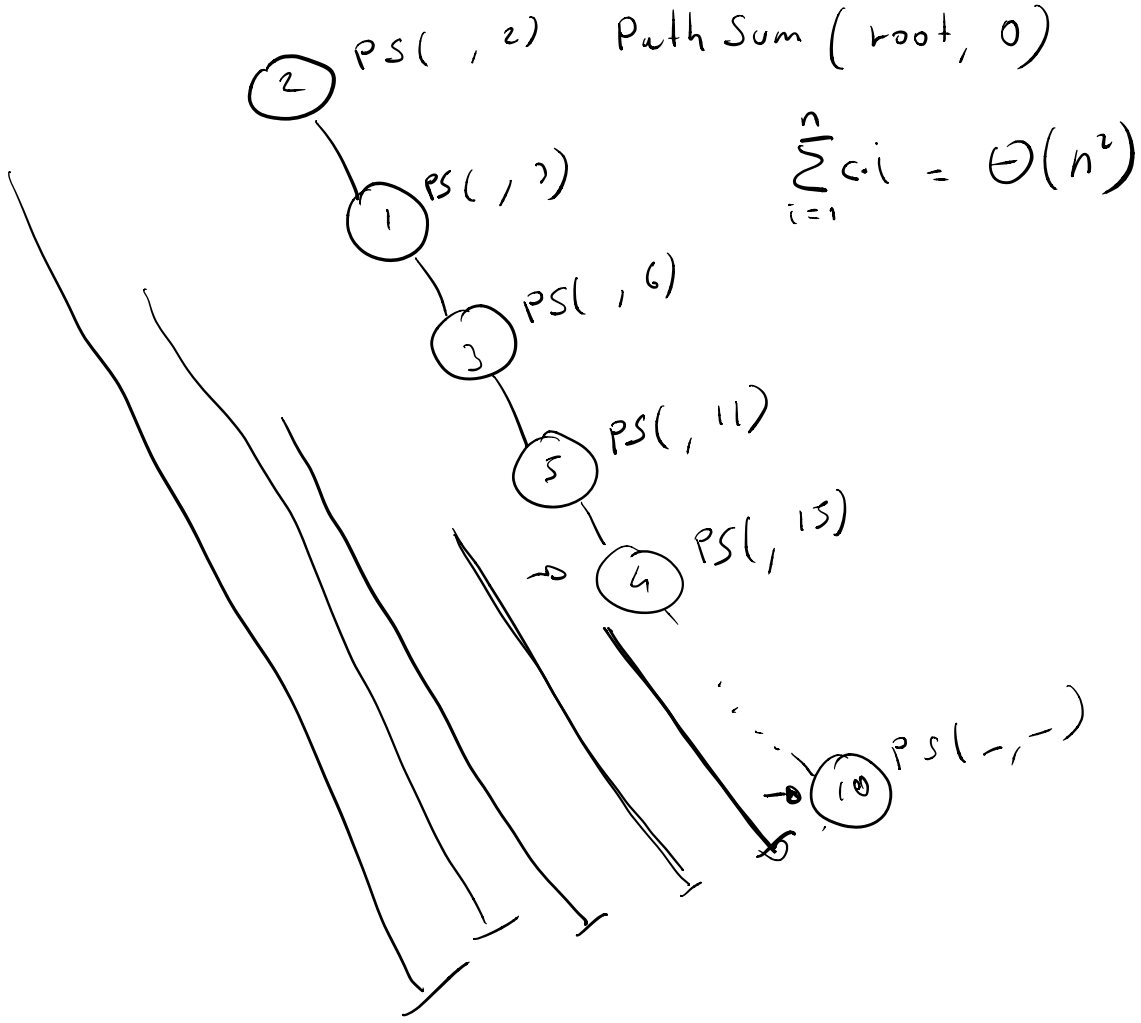
Very BAD solution

```
path sum (x, sum)
    if x == NIL return
    sum += u.key
    path sum (x.left, sum)
    path sum (x.right, sum)
    if subtree size (x) == sum
        print (x.key)
```

```
subtree size (x)
    if x == NIL return 0
    sl = subtree size (x.left)
    sr = subtree size (x.right)
    sx = sl + sr + 1
    return sx
```

it takes $\Theta(n^2)$ in wc on a tree of n nodes

Worst case scenario



$$\sum_{i=1}^{n} c \cdot i = \Theta(n^2)$$

```
path sum (x, sum)
    if x == NIL return 0
    sum += x.key
    sl = path sum (x.left, sum)
    sr = path sum (x.right, sum)
    sx = sl + sr + 1 // Θ(1)
    if sx == sum
        print (x.key)
    return sx
```

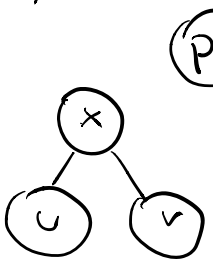
```
path sum (x, sum)
    if x == NIL return
    sum += x.key
    path sum (x.left, sum)
    path sum (x.right, sum)
    if subtree size (x) == sum
        print (x.key)
```

```
subtree size (x)
    if x == NIL return 0
    sl = subtree size (x.left)
    sr = subtree size (x.right)
    sx = sl + sr + 1
    return sx
```

Complexity = $\Theta(n)$ time

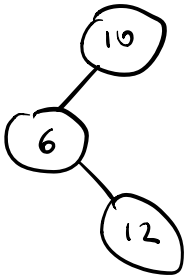
Exercise

Given a tree, check if it's a BST



u.key < x.key and
x.key < v.key

NOT CORRECT!!!



P_1 is satisfied for every node
BUT IT'S NOT A BST

- for every node x,
  max (v.left) ≤ x.key and
  min (v.right) ≥ x.key

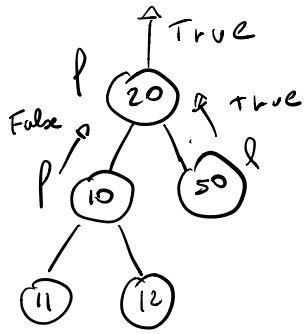
Assume P_1 is enough, write a correct program to check P_1 true for every node

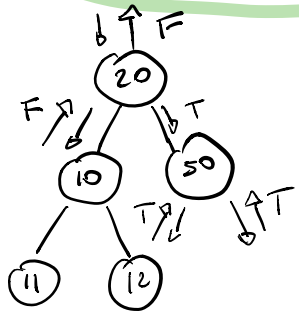
```
p(x)
    if x == NIL return True
    if x.left.key > x.key
        return False
    if x.right.key < x.key
        return False
    p(x.left)
    p(x.right)
```

```
bl = p(x.left)
br = p(x.right)
```

return True

return bl and br





max(x.left subtree) < x < min(x.right subtree) P2

```
check BST (x)                    // return - boolean False if some subtree below x was not a BST
    if x == NIL  return True, +∞, -∞
    bl, ml, Ml = check BST(x.left)    - min key on x's subtree
    br, mr, Mr = check BST(x.right)   - max key on x's subtree
    bx = bl and br
    if (Ml > x.key) or (mR < x.key)
                bx = False
    mx = min ( ml, mr, x.key) // O(1)  min of 3
    Mx = max(Ml, Mr, x.key) // O(1)
    return  bx, mx, Mx
```

compute minkey(x)

```
min key (x)
    if x == NIL  return +∞
    ml = min key (x.left)
    mr = min key (x.right)
    mx = min ( ml, mr, x.key)
    return mx
```

```
min BST (x)
    if x.left == NIL  return x.key
    return min BST(x.left)
```