ADS 2021: Week 10 Solutions

Solutions for week 10 of Algorithms and Data Structures.

3.2.9 - Green

```
N = 2

Insertion order: 1 2
 1
  2

Insertion order: 2 1
 2
1


N = 3

Insertion order: 1 2 3
 1
   2
     3

Insertion order: 1 3 2
 1
   3
  2

Insertion order: 2 1 3, 2 3 1
 2
1 3

Insertion order: 3 1 2
 3
1
 2

Insertion order: 3 2 1
   3
  2
 1


N = 4

Insertion order: 1 2 3 4
 1
  2
   3
```

```
    4

Insertion order: 1 2 4 3
 1
  2
   4
  3

Insertion order: 1 3 2 4, 1 3 4 2 (same shape)
 1
  3
 2 4

Insertion order: 1 4 2 3
 1
  4
 2
  3

Insertion order: 1 4 3 2
  1
   4
  3
 2

Insertion order: 2 1 3 4, 2 3 1 4, 2 3 4 1
 2
1 3
   4

Insertion order: 2 1 4 3, 2 4 1 3, 2 4 3 1
  2
1   4
   3

Insertion order: 3 2 1 4, 3 2 4 1, 3 4 2 1
    3
  2   4
 1

Insertion order: 3 1 2 4, 3 1 4 2, 3 4 1 2
    3
  1   4
   2

Insertion order: 4 3 2 1
    4
   3
  2
```

```
 1

Insertion order: 4 3 1 2
    4
   3
  1
   2

Insertion order: 4 2 1 3, 4 2 3 1
    4
   2
  1 3

Insertion order: 4 1 2 3
    4
   1
    2
     3

Insertion order: 4 1 3 2
    4
   1
    3
   2
```

3.3.2 - Green Hint: Keep this in mind when solving 3.3.11 so you can compare the two trees with the students.

```
Insert Y     Y

       L     LY

       P     LPY

       P     P
            L Y

       M       P
            LM Y

       X       P
            LM XY

       H         P
            HLM XY

       H       LP
            H M  XY

       C       LP
```

```
        CH M  XY

R         LP
        CH M  RXY

R         LPX
        CH M  R Y

R           P
          L     X
        CH M   R Y

A           P
            L     X
        ACH M   R Y

A           P
          CL      X
        A H M   R Y

E           P
          CL       X
        A EH M   R Y

S             P
          CL         X
        A EH M   RS Y
```

3.3.3 - Green

```
Insertion order: A X E M R C H S

Insert A     A

Insert X     AX

Insert E     AEX

              E
             A X

Insert M      E
             A MX

Insert R      E
             A MRX

              ER
             A M X
```

```
Insert C      ER
            AC M X

Insert H       ER
            AC HM X

Insert S       ER
            AC HM SX
```

3.3.11 - Green

```
Insert Y        (B)Y

       L        (B)Y
              (R)L

       P        (B)Y
              (R)L
                 (R)P

       P        (B)Y
              (R)P
            (R)L

       P      (B)P
            (R)L  (R)Y

       P      (R)P
            (B)L  (B)Y

       P      (B)P
            (B)L  (B)Y

       M        (B)P
            (B)L      (B)Y
              (R)M

       M       (B)P
            (B)M     (B)Y
          (R)L

       X        (B)P
            (B)M      (B)Y
          (R)L      (R)X

       H         (B)P
             (B)M      (B)Y
           (R)L      (R)X
          (R)H
```

```
H           (B)P
     (B)L         (B)Y
   (R)H (R)M    (R)X
```

```
H           (B)P
     (R)L         (B)Y
   (B)H (B)M    (R)X
```

```
C            (B)P
      (R)L         (B)Y
    (B)H (B)M    (R)X
  (R)C
```

```
R            (B)P
      (R)L         (B)Y
    (B)H (B)M    (R)X
  (R)C          (R)R
```

```
R            (B)P
      (R)L         (B)X
    (B)H (B)M    (R)R (R)Y
  (R)C
```

```
R            (B)P
      (R)L         (R)X
    (B)H (B)M    (B)R (B)Y
  (R)C
```

```
R            (R)P
      (B)L         (B)X
    (B)H (B)M    (B)R (B)Y
  (R)C
```

```
R            (B)P
      (B)L         (B)X
    (B)H (B)M    (B)R (B)Y
  (R)C
```

```
A              (B)P
        (B)L         (B)X
      (B)H (B)M    (B)R (B)Y
    (R)C
   (R)A
```

```
A                 (B)P
           (B)L          (B)X
      (B)C     (B)M    (B)R (B)Y
    (R)A (R)H
```

```
A                (B)P
            (B)L          (B)X
      (R)C     (B)M    (B)R (B)Y
    (B)A (B)H
```

```
E                (B)P
            (B)L          (B)X
      (R)C     (B)M    (B)R (B)Y
    (B)A  (B)H
         (R)E
```

```
S                (B)P
            (B)L             (B)X
      (R)C     (B)M    (B)R     (B)Y
    (B)A  (B)H             (R)S
         (R)E
```

```
S                (B)P
            (B)L             (B)X
      (R)C     (B)M     (B)S    (B)Y
    (B)A  (B)H         (R)R
         (R)E
```

3.2.11 - Yellow We can build different shapes of trees of height N-1 with all combinations of right and left links on nodes with children. Example with N = 4:

```
Insertion order: 1 2 3 4
1
 2
  3
   4
```

```
Insertion order: 1 2 4 3
1
 2
  4
 3
```

```
Insertion order: 1 4 3 2
 1
  4
 3
2
```

```
Insertion order: 1 4 2 3
 1
  4
 2
```

```
  3

Insertion order: 4 1 3 2
 4
1
 3
2

Insertion order: 4 1 2 3
 4
1
 2
  3

Insertion order: 4 3 1 2
  4
 3
1
 2

Insertion order: 4 3 2 1
   4
  3
 2
1
```

Therefore, there are $2^{(N-1)}$ binary tree shapes of N nodes with height $N-1$. And there are $2^{(N-1)}$ different ways to insert N distinct keys into an initially empty BST that result in a tree of height $N-1$.

3.2.20 - Yellow We present two possible solutions:

a) Consider the algorithm as making growing subtrees:

The first "drill down" creates a left boundary: We start at the root and recursively either keep going left or cut off the subtree rooted at x.left and go down x.right. This left boundary will at most examine nodes corresponding to the depth of the tree.

The algorithm then walks up the left boundary, adding the key if it is between lo and hi and recursively call itself on x.right. If the key is larger than hi, the algorithm returns, effectively creating a right boundary. If it is not larger then hi (note, these are larger than lo, since they are to the right of the left boundary), then the nodes in this subtree are added.

The nodes between the two boundaries corresponds to the amount of keys between lo and hi.

The total time complexity will therefore be the depth of the tree + the amount of keys returned.

b) Proposition: The running time of the two-argument keys() in a BST is at most proportional to the tree height plus the number of keys in the range.

Proof: The two-argument keys() method in a BST works in the following way: 1- It searches the tree until it finds the element which is equal or higher than the lower bound. 2- It adds the element to the queue and also adds all its right children that are inside the search range. 3- It adds all the other elements which are inside the search range by doing an in-order search in the tree (but cutting branches once it finds elements outside the range).

Step 1, searching the element which is equal or higher than the lower bound is a regular search in a BST and takes O(lg N) time, which is equal, in the worst case, to the tree height. Steps 2 and 3 combined take R operations, where R is the number of elements in the range searched. There may be a constant number of other compares, which are the cases where the method finds an element outside the range and stops the search on the current branch of the tree. Combining all the steps the two-argument keys() in a BST is at most proportional to the tree height plus the number of keys in the range.

Counting keys - Red As in the BST.java implementation, we add an int size attribute to the node class. Then we perform the same "drill down" operation as in the first proposed solution for exercise 3.2.20. When we "cut off" the left subtree, we subtract the size of the left subtree from (for simplicity) a global count that is instantiated as the size of the root. When we reach the bottom, we do NOT explore the subtrees to the right. At this point, the count is the amount of keys between $[a; \infty]$. At the root, we perform the same "drill down" operation, but defining a *right* boundary instead. Then we have the nodes between $[a; b]$.

Counting odd keys - Red The solution is the same as counting keys, but we modify the size attribute to describe the amount of odd keys in the subtree.