CS 430 - Introduction to Algorithms

Homework #13

Name : Madhushalini Murali

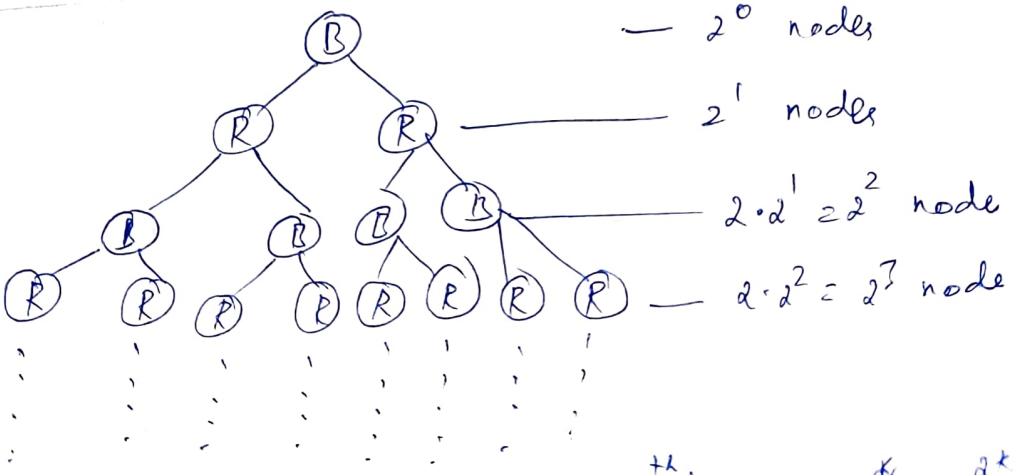
CWID : A20513784

1) The largest possible number of internal nodes in a red-black tree with height k is $2^{2k+1} - 1$.

This can be achieved by considering a complete binary tree. If the height is k, then level 0 will have 2^0 nodes, level 1 will have $2 \times 2^1 = 2^1$ nodes, level 2 will have $2^1 \cdot 2 = 2^2$ nodes, so the k^{th} level will have $2 \cdot 2^k = 2^{k+1}$ nodes. Since we are having both red & black nodes, the k^{th} level will have 2^{2k+1} nodes. Since it is the internal nodes, the height will $2^{2k-1} + 1$.

The smallest possible number is $2^{k-1} - 1$ which has only the black nodes in a complete binary tree.

The balanced binary tree for maximum possible number.

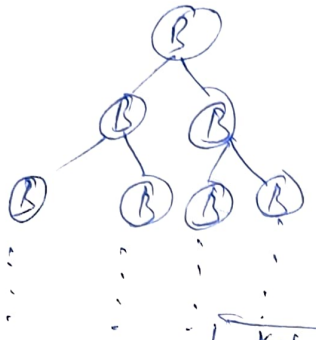


k^{th} level - 2^{2k-1} nodes.

because it has combination of red & black nodes

The balanced binary tree for the smallest possible number :

This has only black nodes



The height is $2^{K-1} - 1$.

2) Let range be a function that returns the number of nodes / key(x) in the specified range $a <= x <= b$.

```
def range(root, a, b):
    if not root:
        return none
    if root.data == a && root.data == b
        return true
if a <= root.data and b >= root.data
    check if the value is within the range.
    If in range, then add the value to the range
    and then check for its left and right subtrees.
    return true
else if a > root.data
    check and recur the values in right subtree
    return false
else (# this check for b < root.data)
    check and recur the values in left subtree
    return false
```

From the 'range' function, we can get the key values that satisfy the condition $a <= x <= b$.

Time Complexity : $O(\log n)$

h - height of the Tree

k - number of nodes in given range

Pseudocode :

```
def range (root, a, b) :
    if root :
        return false

    if root.data == a && root.data == b
        return true

    if a <= root.data && b >= root.data
        return true

    else if a > root.data
        return range (root.right, a, b)

    else if b < root.data
        return range (root.left, a, b)
```

Time : $O(\log n)$

3) function count (D, x):
    return large_count (root, x)

function large_count (root, x):
    if (root.left == None and root.right == None)
        if (root.data() > x):
            return 1
        else:
            return 0
    if (root.data() == x)
        if (root.right != None):
            return root.right.size()
        else:
            return 0
    else if (root.data() > x):
        if (root.left != None):
            if (root.right == None):
                return 1 + large_count (root.left, x)
            else
                return 1 + root.right.size() +
                    large_count (root.left, x)
        else
            return 1 + root.right.size()
    else:
        if (root.right == None):
            return 0
        else:
            return large_count (root.right, x)
    else:
        return 0

The time complexity is $O(\log n)$ => n : number of nodes

- The present node's child is a part of recursive calls of the function.

- The sub tree's size can be determined since it is stored in the node information

1) Let us define a function Prime() that checks the given condition in the question.

```
function Prime (x, T):
{
    if (x >= 2)      (# 0 & 1 are not prime):
    {
        if search (x, T) != None:
            return 1      {# for x is prime
                           and the key already exist
                           in BST};

        # to check if x is prime
        for i in range (2, int (sqrt(x)) + 1):
            if (x % i == 0)
                return 0      (# x is not prime)

        insert (x, 1)      (# x is prime & insert key in BST)
        return 1
    }
    else      (# if x is less than 2)
    {
        return 0
    }
}
```