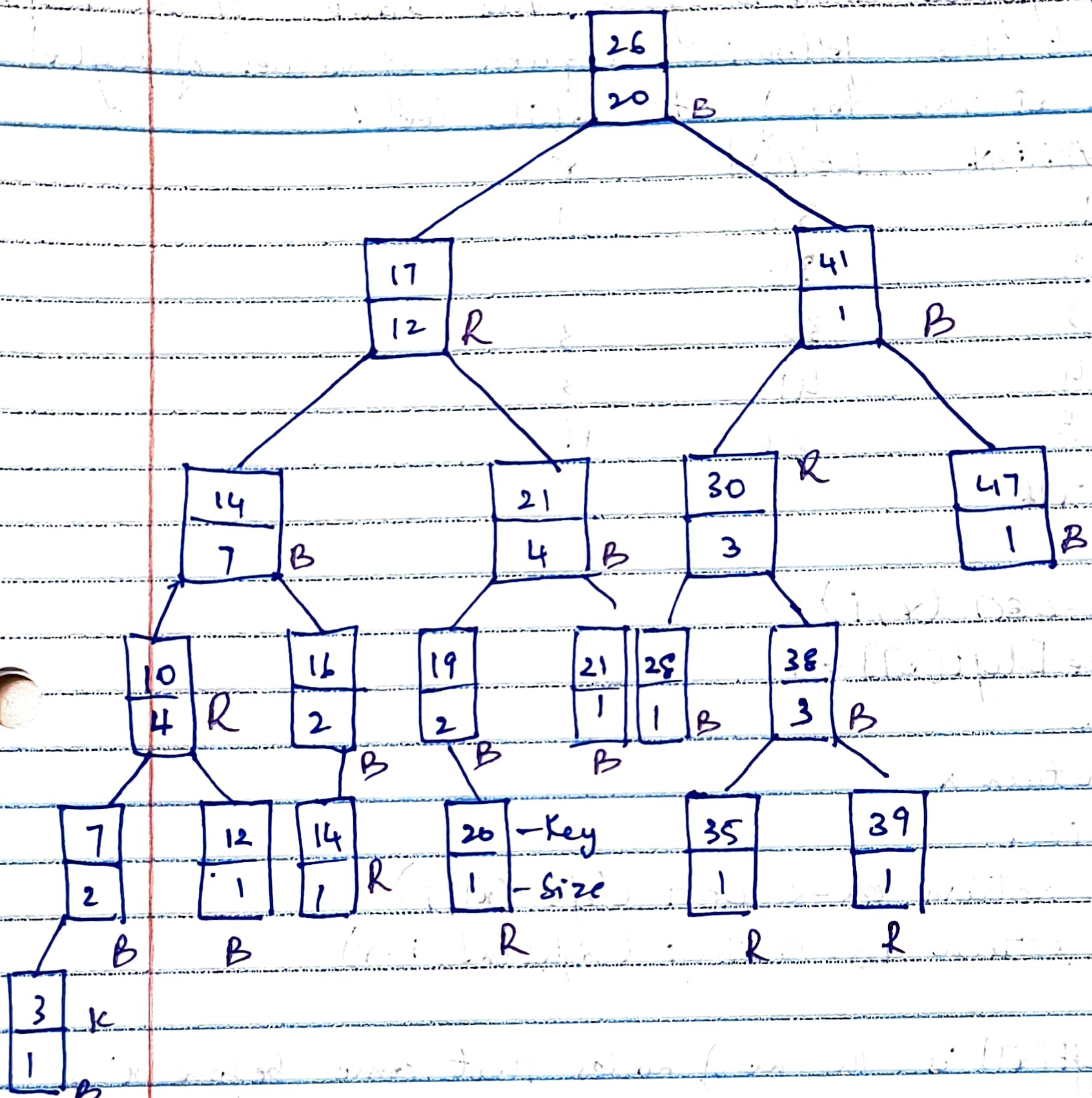
14.1-1. Step-1



Step-2

OS-RANK (T, x) operation with the node x with Key [x]=35
OS-RANK (T, x)
r ← size[left[x]]+1
y ← x
while y ≠ root (T)
    do if y = right [p [y]]
        then r ← r+size [left [p[y]]]+1
    y ← p [y]
return r

When we run OS-RANK on the order-statistic tree in the figure above to find the rank of the key 35, we get the following sequence of values of key[x] and r at the top of the while loop:

ITERATION	KEY[x]	r
1	35	1
2	38	1
3	30	3
4	41	3
5	26	16

Rank is 16

14.1-2 OS-SELECT (x, i)
r size [left[x]] + 1
if i = r
then return x
elseif i < r
then return OS-SELECT (left [x], i)
else return OS-SELECT (right [x], i - r)

size[left[x]] is the no of nodes that come before x in an inorder tree walk of the subtree rooted at x.
size [left[x]] + 1 is the rank x within the subtree rooted at x.

We compute r, the rank of node x within the subtree rooted at x. If i = r, then x is the i^{th} smallest. Return x in line 3. If i < r, then i^{th} smallest element is x's left subtree. If i > r, then i^{th} smallest element is in x's right subtree.

Example:- To know how OS-SELECT operates, consider the search for the 17^{th} element. We begin with x as the root, whose key is 26, and with i = 9. Since the size of 26's left subtree is 12, its rank is 13. Thus, we know the node with rank 8 is less than i = 9. 9 - 8 = 1 is 6 smallest element in subtree rooted at the node with key 21.
After the recursive call, x is the node with key 19. It's rank is 1 and i = 1.
A pointer to the node with key 19 is returned.

```
OS-RANK (T,x)
r size[left[x]]+1
y x
while y root[T]
do if y = right[p[y]]
then r r + size[left[p[y]]] + 1
y p[y]
return r.
```

The rank of x can be viewed as the no. of nodes preceding x in an inorder tree walk, plus 1 for itself. At the top of the while loop of lines 3-6, r is the rank of key[x] in the subtree rooted at node y. We consider the subtree rooted at p[y]. We have already counted the no. of nodes in the subtree rooted

Example Explanation:

When we run OS-RANK on the order statistic tree of algorithm to find the rank of the node with key 35, we get the sequence of values of key[y] and r at the top of the while loop

iteration	key[y]	r
1	35	1
2	38	1
3	30	3
4	41	3
5	26	11

Rank = 11.

14.1.5) Data Structure should have these 2 operations

1 Get(i) - which gives the key at i^{th} position of the total order of keys

2 Rank(x) - which gives the position of x in total order of keys

Get (Rank(x) + i)

In an order statistic tree, each and every node x keeps the record of the no. of nodes contained in the subtree rooted on x.

Using these 2 operations will keep the track of no. of nodes lie to the left of our path

14.3.3 Step-1

As it goes down the tree, INTERVAL-SEARCH first checks whether current node x overlaps the query interval i and if it does not, goes down to either the left or right child. If node x overlaps i, node in the right subtree overlaps i, but no node in the left subtree overlaps i, because the keys are the low endpoints. If there is an interval that overlaps i in the left subtree of x, then checking x before the left subtree might cause the procedure to return an interval whose low endpoint is not minimum of those that overlap i.

Step-2

If there is a probability that the left subtree might contain an interval that overlaps i, we need to check the left subtree first. If there is no overlap in the left subtree but node x overlaps i, then we return x. Check the right subtree under the same conditions as in INTERVAL-SEARCH: the left subtree cannot contain an interval that overlaps i. It is easier to write the pseudocode to use a recursive procedure MIN-INTERVAL-SEARCH-FROM (T, x, i), which returns the node overlapping i with the minimum low endpoint in the subtree rooted at x, or nil[T] if there is no such node

```
MIN-INTERVAL-SEARCH(T, i):
    x = T.root
    while x != T.nil:
        if x.left != T.nil and i.low <= x.left.max:
            x = x.left
        elif x.int overlaps i:
            break
        else
            x = x.right
return x
```

The call MIN-INTERVAL-SEARCH(T, i) takes $O(\lg n)$ time, since each recursive call of MIN-INTERVAL-SEARCH-FROM goes one node lower in the tree, and the height of the tree is $O(\lg n)$