The given implementation of `insert(key)` and `contains(key)` is not linearizable.

Consider two processes P and Q concurrently adding the same key `l` and a process W adding the key `r`, `r > l`. Prior to start of these processes the tree has such structure that If `insert(l)` and `insert(r)` were performed sequentially, `Node(l)` would become the right child of some node `n1` and `Node(r)` would become the right child of `Node(l)`.

P finds the insertion point and suspends before the invocation of `n1.lock.lock()` until Q reaches the same point. P wakes, locks `n2`, assigns `n1.right = Node(l)` and releases the lock. Denote `Node(l)` created by P `n2`. Then wakes W, finds an insertion point for `r`, which is `n2.right`, locks `n2`, assigns `n2.right = Node(r)`, unlocks `n2` and completes the operation. Then wakes Q, locks `n1` and rewrites `n2` with a new node `n3`.

P : `insert(l)`

```
if node.right != null:
  return insert(node.right, key)
// node is n1
// n1.right is insertion point
node.lock.lock()
node.right = Node(key) // n2
node.lock.unlock()
```

Q : `insert(l)`

```
if node.right != null:
  return insert(node.right, key)
// node = n1
// n1.right is insertion point
// sleep...










// ...wake
node.lock.lock()
// rewrites n2 with a new node
node.right = Node(key)
node.lock.unlock()
```

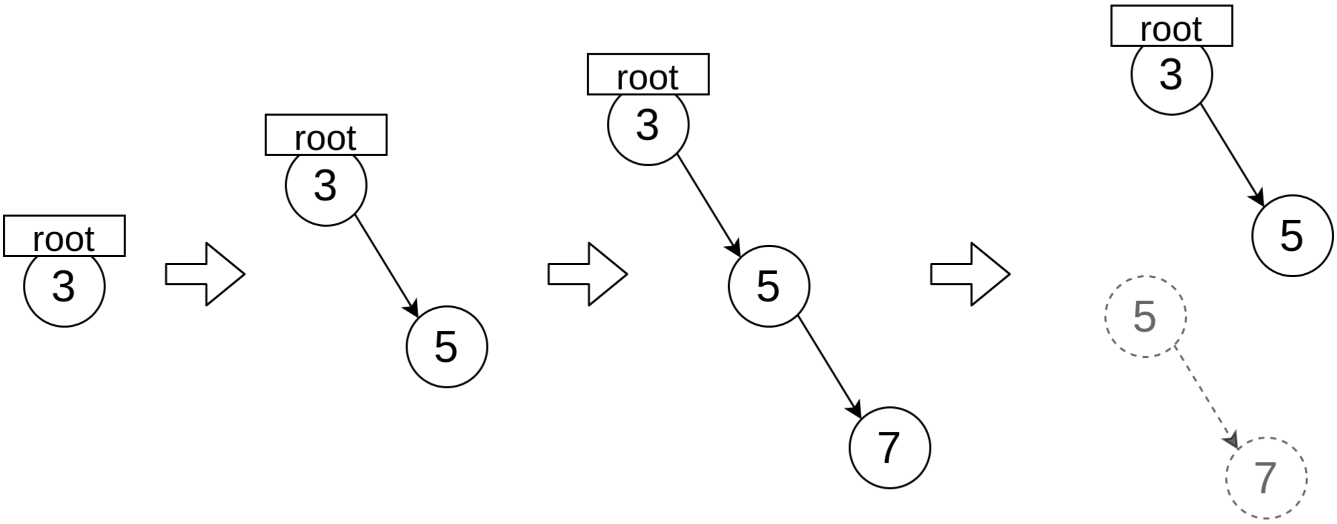
W : `insert(r)`

```
// sleep...






// ...wake
if node.right != null:
  return insert(node.right, key)
// node is n2
// n2.right is insertion point
node.lock.lock()
node.right = Node(key)
node.lock.unlock()
```



So the added key `r` gets lost.

Imagine another process Z invoking `contains(r)` twice sequentially. As contains does not lock anything, the first invocation may be performed right after the process W added `r` and return `true`, while the second invocation may start after the process Q rewritten `n2` and return `false`.

As keys are never removed, the history where a process observes vanishing of a key is not linearizable.

We may use double-checked locking pattern to fix the tree implementation.

```
class Node(val key: Int) {
  var left: Node? = null
  var right: Node? = null
  val lock = Lock()
}

fun contains(node: Node, key: Int): Boolean {
  if node == null: return false
  if node.key == key: return true
  nextNode = node.key < key ? node.right : node.left
  return contains(nextNode, key)
}

fun insert(node: Node, key: Int): Boolean {
  if node.key == key: return false
  if node.key < key {
    if node.right != null:
      return insert(node.right, key)
    node.lock.lock()
    if node.right == null {
      node.right = Node(key)
      node.lock.unlock()
      return true
    } else {
      node.lock.unlock()
      return insert(node.right, key)
    }
  } else {
    // similar code for insertion into root.left
    ...
  }
}
```