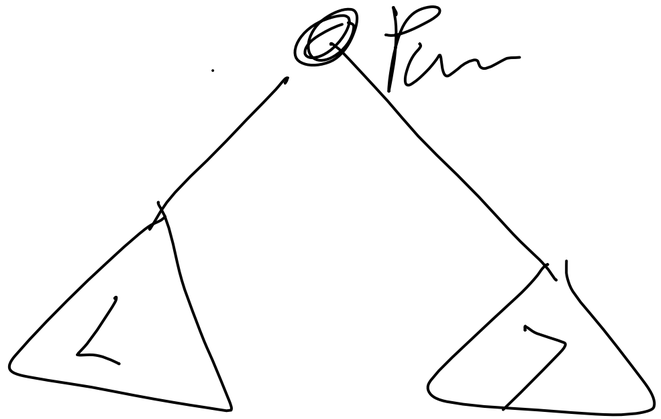
11.09.2024

Binary Search Tree

- Binary Tree
- key(left tree) < key(parent) < key(right tree)



Procedure Insert (Node n, Node T)

// n is the new node to be inserted

// into the tree T { n.left = n.right = NULL }

if (T = NULL) return n.

Temp ← T

while (Temp ≠ NULL)

{ oldTemp ← Temp.

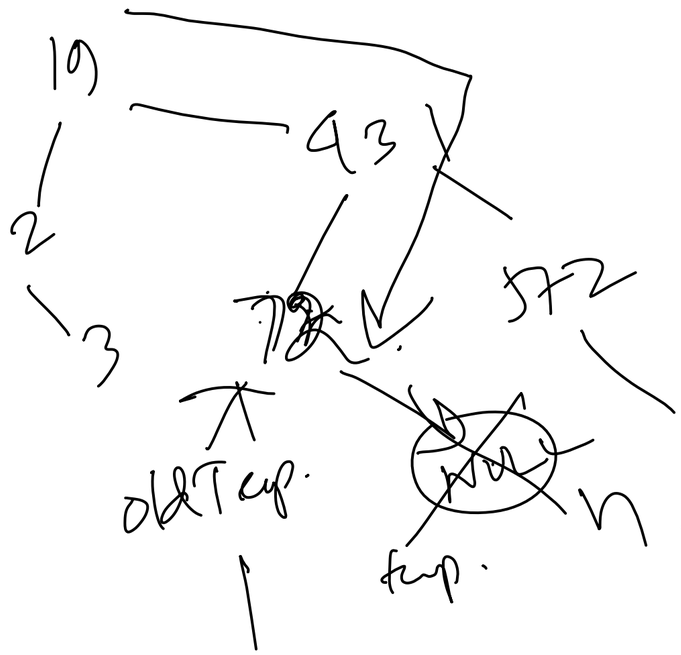
if (Temp.key = n.key)

return error

// key already exists.

else if (Temp.key > n.key)

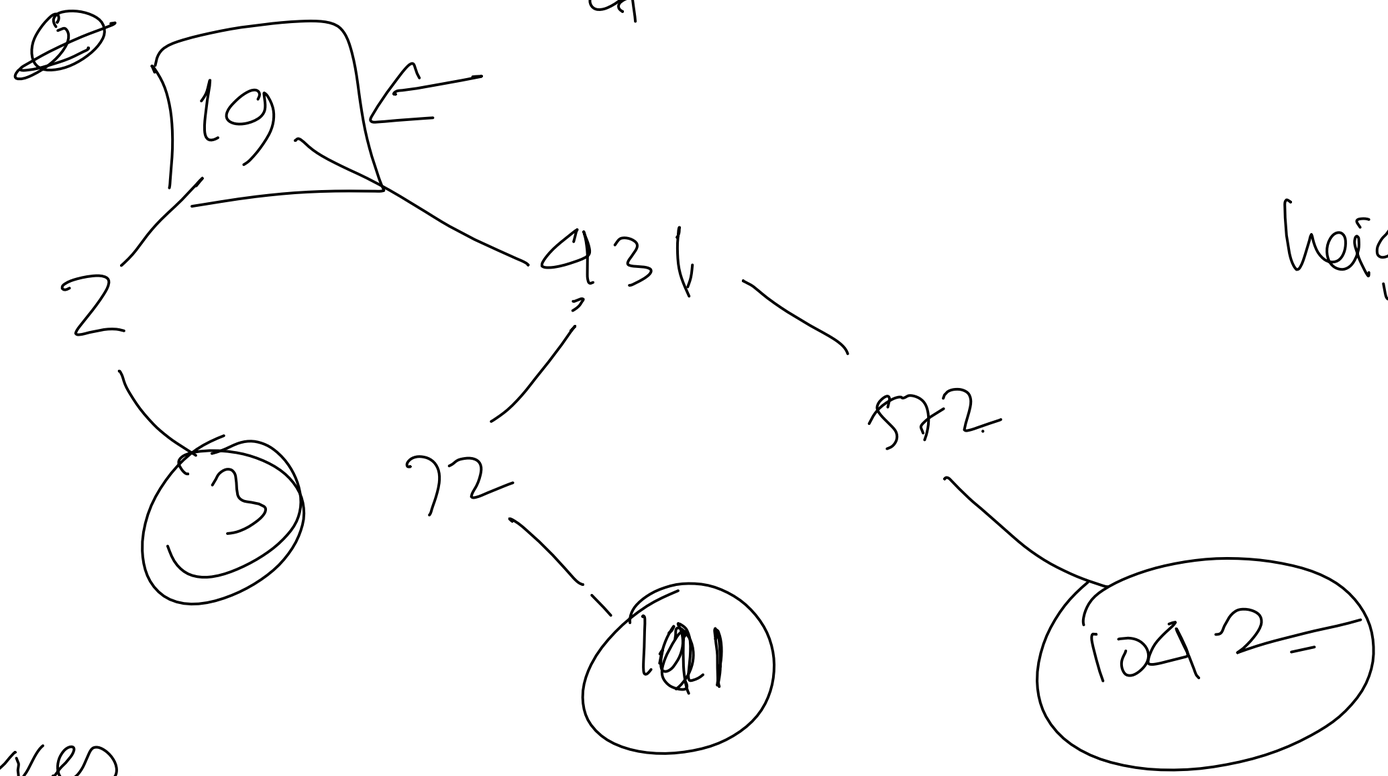
Temp ← temp.left



```
                else
                    temp ← temp.right.
            endif
        endwhile
    if (oldTemp.key < n.key)
        oldTemp.right ← n.
        n.parent ← oldTemp.
        n.left ← NULL
        n.right ← NULL
else if (oldTemp.key > n.key).
        oldTemp.left ← n.
        n.parent ← oldTemp
        n.left ← NULL
        n.right ← NULL.
endif
```

Time Complexity ?

height of tree: maximum # hops of needed to go from root to any leaf of tree.



height = 3

○ → leaves.

□ → root

Search

Procedure Search (NodeT, Key).

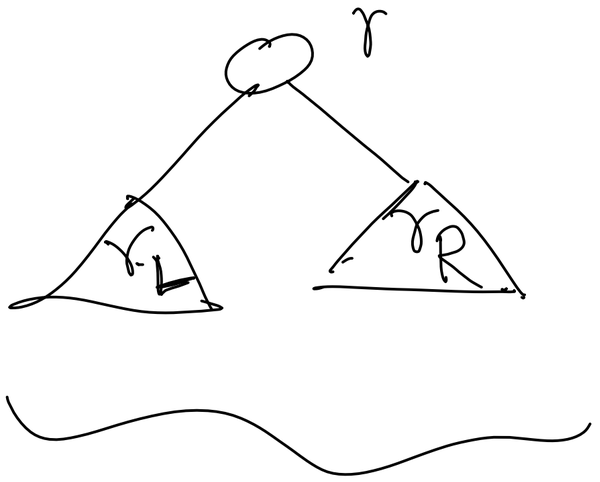
// return either node corresponding to Key

// or NULL.

{

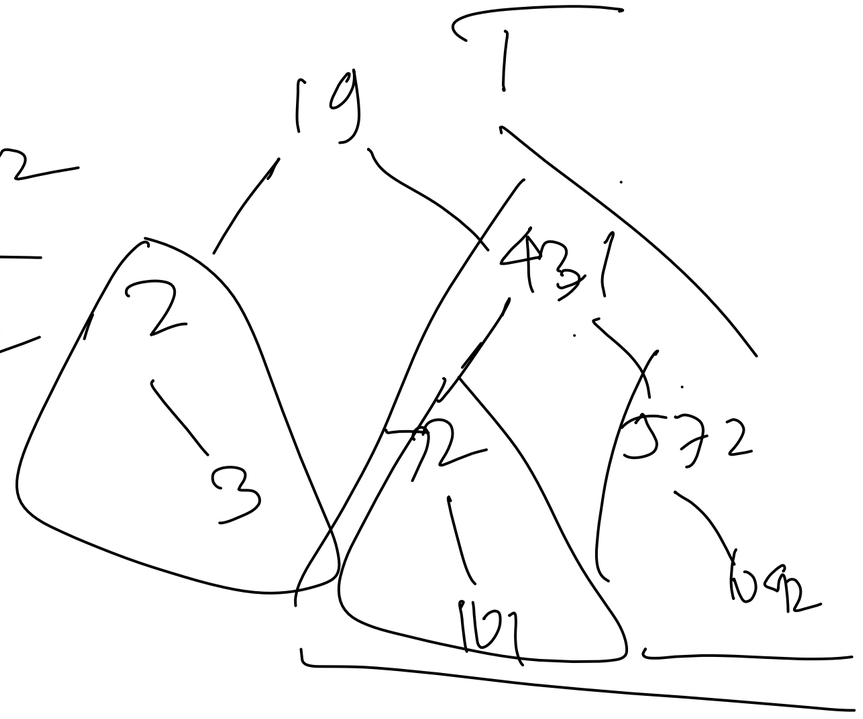
Tree ~~Key~~ Traversal

- inOrder → (r_L, r, r_R)
- preOrder → r, r_L, r_R
- postOrder → r_L, r_R, r



2, 3, 19, 72, 101, 431, 572, 1042

19, 2, 3, 431, 72, 101, 572, 1042



```
Procedure InOrder(Tree T)
  if (T = NULL) return
  InOrder(T.left)
  //visit root
  Print(T.key)
  InOrder(T.right)
```

Correctness: 1, 2, ..., n-1

n.

Obvious: Inductive argument. n = # nodes.

O(n)

Procedure Smallest (Node T).

// returns leftmost node.

Procedure Largest (Node T)

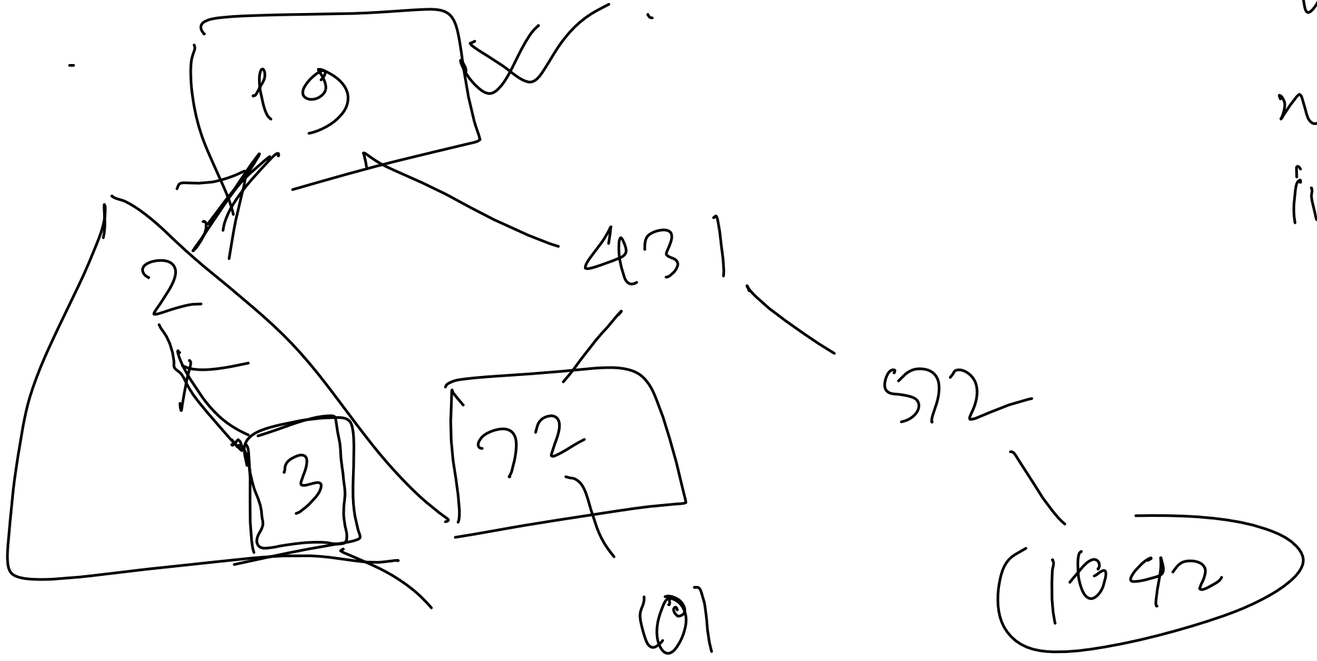
// returns largest node.

running tm: $O(h)$

h: height of tree.

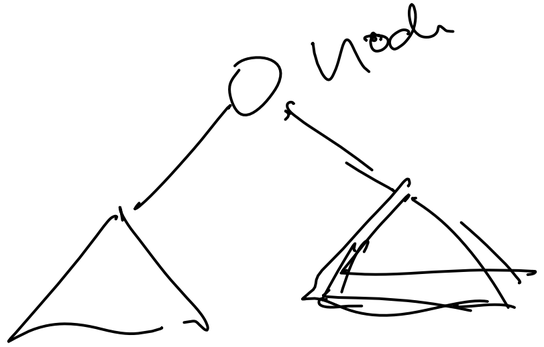
Successor & Predecessor (key)

Successor ("key") : returns the node whose key is immediately next to "key" in the inorder traversal / sorted order.



Subtree would be 2.

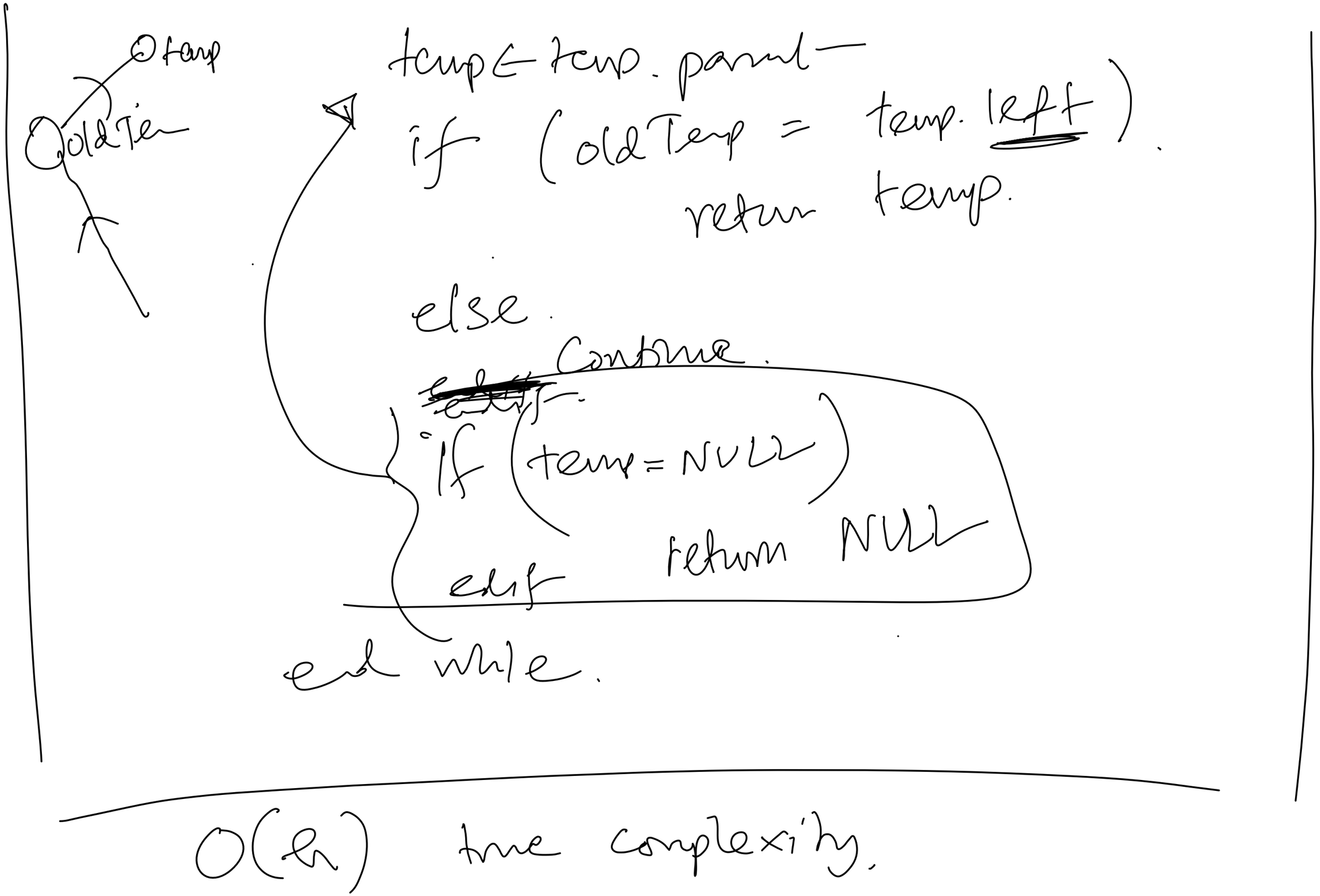
Successor (19) = 72 ; Success (3) = 19.



successor(node) → smallest(node.right)

```
Procedure Successor(Node n).
// returns Successor node of n.
if (n = NULL)  return error.
elseif (n.right ≠ NULL).
        return Smallest(n.right).

else
        temp ← n.
        while(1)
                oldTemp ← temp.
```

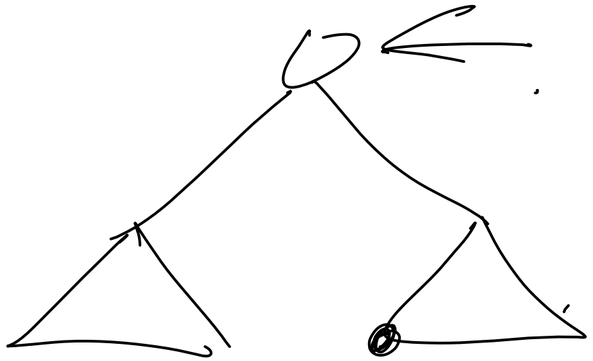


O(h) time complexity.

```
Procedure Predecessor (node n).
// returns node immediately before n
// in the inorder traversal.
```

Delete

replace with successor.



Procedure Delete (Node n).

if (n = null) return error.

if (n.right = null & n.left = null)

if (n = parent. = null)

return null.

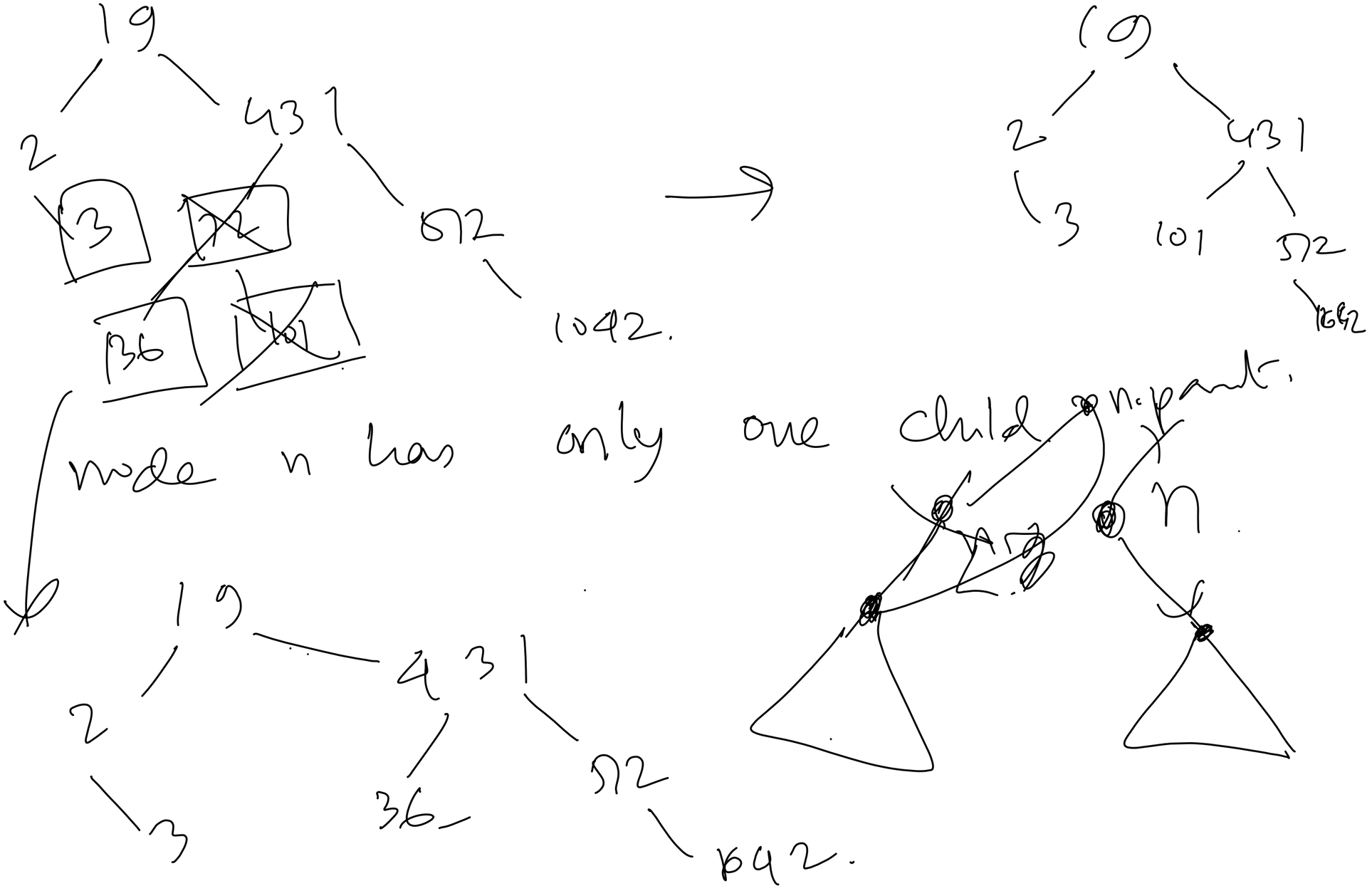
n is a leaf node.

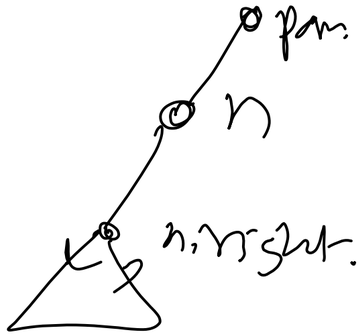
else. temp ← n.parent.

if (n = temp.left)

temp.left ← NULL.

else (temp.right ← NULL)


19
2
431
3
572
1042.
36
101
Lazy node n has only one child
n: parent.
n
19
2
431
3
36
572
642.



else if (n.left = NULL)

// replace n. with n.left.

~~else~~ n_parent ← n.parent.

n_right ← n.right.

n.right.parent ← n.parent.

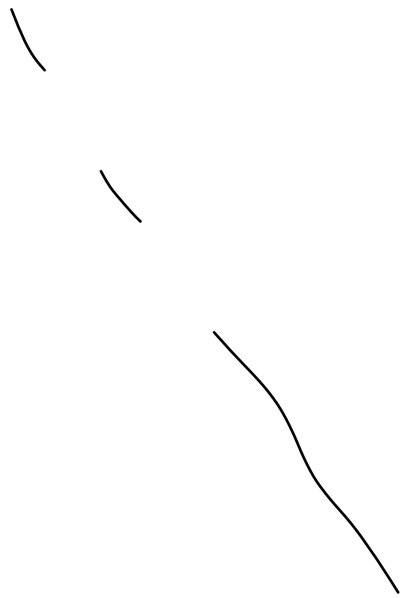
if (n = n_parent.left)

n_parent.left ← n.right.

else

n_parent.right ← n.right

elif (n.right == NULL)
// replace n with n.right.



```
else
{
    // n has got both left & right
       child.
    // successor exists in right subtree
    // return that.
```