Algorithm 23_tree_insert(root, key)

Here we assume that each node (either 2-node or 3-node) in a 2-3-tree is a structure with 5 fields:

left_child, first_key, mid_child, second_key, right_child

For a 2-node, the "second_key" field is set to a special value that is bigger than any legal key value (for example, if keys are integers, we can set the second_key filed to an integer INT-MAX which is bigger than any possible key values)

Input: root - root node of the 23-tree
key - the key value to be inserted

Output: NULL, if key is already in the tree
else root is returned after the insertion
root may be a new node (if the tree was previously empty, or if the root node is split)

Processing: If the key is not in the tree, find the suitable leaf node (along with the path from the leaf to root) and insert the key in the leaf. If the insertion causes a 3-node to overflow (split), then move the median key to the parent of the split node, going up the path to root, until no overflow after the insertion, or the root node is being split - in that case, build a new root node

```
Step1  [empty tree?]
      If (root = NULL)
          root <-- new_root(NULL, key, NULL)   /* build a root node */
          return root

Step2   [key already in tree?]
       stack <-- search1(root, key)
                   /* stack = NULL if key already in the tree */
       If (stack = NULL)
          return NULL
Step3  [insert]
                      /* stack has the path from the root to leaf,
                        top of the stack is the leaf */
        taller <-- true
        newNode  <-- NULL

        while (taller = true and stack not empty)

           node <-- pop(stack)
                    /* first time, node is the leaf  to insert key into */
                    /* subsequently node is the parent of a split node */
           if (node is a 2-node)
                putin(node, key, newNode)
                taller = false    /* insert key into node, no overflow  */
           else               /* node is a 3-node */
               (newNode, median_key) <-- split(node, key, newNode)
               key <-- median_key
                        /* the "split" function splits "node" into
                         a modified "node" and newNode, and median_key
                         is the one "promoted" to parent of "node" */
                If (root = node)   /* the root is being split already */
                    root <-- new_root(node, key, newNode)
             /* end of the while loop */

           return root
```

****

new_root(left_tree, key, right_tree)

Input: root of left subtree, key to be inserted into new root node,
and root of the right subtree

output: new root node (a 2 node), containing key as its first key,
left_tree, right_tree as its left_child and mid_child

****

search1 (root, key)

Input: root node of a 2-3-tree
    and a search key
output: NULL if key is already in tree,
    else, a stack containing the nodes in the path from
    the root to a leaf (to which the key should be inserted),
    with stack top being the leaf

****

pop(stack)

Input: a stack
Output: top of the stack, if stack is not empty
    else -1

****

putin(node, key, newNode)

Input: node - a 2-node
    key - to be inserted into node
    newNode - the root node of the subtree immediately to the right of the key

Output: node (updated - key is inserted into node)

Processing: key is inserted into node
    newNode becomes the mid_child or right_child of node
    dependent on whether key is the first key or second key
    in node.

****

split(node, key, newNode)

Input: node - a 3-node to be split
    key - a key value to be inserted
    newNode - either NULL (when node is a leaf)
        or a 2-node - in this case, it is the
        root node of the subtree which should be
        immediately to the right of key

Output: newNode - a newly generated 2-node after splitting node
    median_key - the median key to be promoted to node’s parent

Processing:

The output newNode should contain the key which is the max value in {key, node-> first_key, node-> second_key}, and the median_key should be the middle value in the three keys above, and the updated node should be a 2-node with the smallest key from the three keys above.

The three original children pointers in node, plus the input newNode occupy the 4 children fields in the modified node and the output newNode.