```
def prune_min(t):
    """Prune the tree mutatively from the bottom up.

    >>> t1 = Tree(6)
    >>> prune_min(t1)
    >>> t1
    Tree(6)
    >>> t2 = Tree(6, [Tree(3), Tree(4)])
    >>> prune_min(t2)
    >>> t2
    Tree(6, [Tree(3)])
    >>> t3 = Tree(6, [Tree(3, [Tree(1), Tree(2)]), Tree(5, [Tree(3), Tree(4)])])
    >>> prune_min(t3)
    >>> t3
    Tree(6, [Tree(3, [Tree(1)])])
    """
    if t.branches == []:
        return
    prune_min(t.branches[0])
    prune_min(t.branches[1])
    if (t.branches[0].label > t.branches[1].label):
        t.branches.pop[0]
    else:
        t.branches.pop(1)



def align_skeleton(skeleton, code):
    """
    Aligns the given skeleton with the given code, minimizing the edit distance
between
    the two. Both skeleton and code are assumed to be valid one-line strings of
code.

    >>> align_skeleton(skeleton="", code="")
    ''
    >>> align_skeleton(skeleton="", code="i")
    '+[i]'
    >>> align_skeleton(skeleton="i", code="")
    '-[i]'
    >>> align_skeleton(skeleton="i", code="i")
    'i'
    >>> align_skeleton(skeleton="i", code="j")
    '+[j]-[i]'
    >>> align_skeleton(skeleton="x=5", code="x=6")
    'x=+[6]-[5]'
    >>> align_skeleton(skeleton="return x", code="return x+1")
    'returnx+[+]+[1]'
    >>> align_skeleton(skeleton="while x<y", code="for x<y")
    '+[f]+[o]+[r]-[w]-[h]-[i]-[l]-[e]x<y'
    >>> align_skeleton(skeleton="def f(x):", code="def g(x):")
    'def+[g]-[f](x):'
    """
    skeleton, code = skeleton.replace(" ", ""), code.replace(" ", "")

    def helper_align(skeleton_idx, code_idx):
        """
        Aligns the given skeletal segment with the code.
        Returns (match, cost)
            match: the sequence of corrections as a string
            cost: the cost of the corrections, in edits
        """
        if skeleton_idx == len(skeleton) and code_idx == len(code):
            return '', 0
        if skeleton_idx < len(skeleton) and code_idx == len(code):
            edits = "".join(["-[" + c + "]" for c in skeleton[skeleton_idx:]])
            return edits, len(skeleton) - skeleton_idx
        if skeleton_idx == len(skeleton) and code_idx < len(code):
            edits = "".join(["+[" + c + "]" for c in code[code_idx:]])
            return edits, len(code) - code_idx

        possibilities = []
        skel_char, code_char = skeleton[skeleton_idx], code[code_idx]
        # Match
        if skel_char == code_char:
            s, c = helper_align(skeleton_idx + 1, code_idx + 1)
            new_s = code_char + s
            possibilities.append((new_s, c))
        # Insert
        s, c = helper_align(skeleton_idx, code_idx + 1)
        new_s = "+[" + code_char + "]" + s
        possibilities.append((new_s, c + 1))
        # Delete
        s, c = helper_align(skeleton_idx + 1, code_idx)
        new_s = "-[" + skel_char + "]" + s
        possibilities.append((new_s, c + 1))
        return min(possibilities, key=lambda x: x[1])
    result, cost = helper_align(0, 0)
    return result


def num_splits(s, d):
    """Return the number of ways in which s can be partitioned into two
    sublists that have sums within d of each other.

    >>> num_splits([1, 5, 4], 0)  # splits to [1, 4] and [5]
    1
    >>> num_splits([6, 1, 3], 1)  # no split possible
    0
    >>> num_splits([-2, 1, 3], 2) # [-2, 3], [1] and [-2, 1, 3], []
    2
    >>> num_splits([1, 4, 6, 8, 2, 9, 5], 3)
    12
    """
    def difference_so_far(s, difference):
        if not s:
            if abs(difference) <= d:
                return 1
            else:
                return 0
        element = s[0]
        s = s[1:]
        return difference_so_far(s, difference + element) + difference_so_far(s,
difference - element)
    return difference_so_far(s, 0)//2


def insert(link, value, index):
    """Insert a value into a Link at the given index.

    >>> link = Link(1, Link(2, Link(3)))
    >>> print(link)
    <1 2 3>
    >>> insert(link, 9001, 0)
    >>> print(link)
    <9001 1 2 3>
    >>> insert(link, 100, 2)
    >>> print(link)
    <9001 1 100 2 3>
    >>> insert(link, 4, 5)
    IndexError
    """
    if index == 0:
        link.rest = Link(link.first, link.rest)
        link.first = value
        # line not needed
    elif link.rest is Link.empty:
        raise IndexError
    else:
        insert(link.rest, value, index - 1)



class Tree:
    """
    >>> t = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
    >>> t.label
    3
    >>> t.branches[0].label
    2
    >>> t.branches[1].is_leaf()
    True
    """
    def __init__(self, label, branches=[]):
        for b in branches:
            assert isinstance(b, Tree)
        self.label = label
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches

    def map(self, fn):
        """
        Apply a function `fn` to each node in the tree and mutate the tree.

        >>> t1 = Tree(1)
        >>> t1.map(lambda x: x + 2)
        >>> t1.map(lambda x : x * 4)
        >>> t1.label
        12
        >>> t2 = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
        >>> t2.map(lambda x: x * x)
        >>> t2
        Tree(9, [Tree(4, [Tree(25)]), Tree(16)])
        """
        self.label = fn(self.label)
        for b in self.branches:
            b.map(fn)

    def __contains__(self, e):
        """
        Determine whether an element exists in the tree.

        >>> t1 = Tree(1)
        >>> 1 in t1
        True
        >>> 8 in t1
        False
        >>> t2 = Tree(3, [Tree(2, [Tree(5)]), Tree(4)])
        >>> 6 in t2
        False
        >>> 5 in t2
        True
        """
        if self.label == e:
            return True
        for b in self.branches:
            if e in b:
                return True
        return False

    def __repr__(self):
        if self.branches:
            branch_str = ', ' + repr(self.branches)
        else:
            branch_str = ''
        return 'Tree({0}{1})'.format(self.label, branch_str)

    def __str__(self):
        def print_tree(t, indent=0):
            tree_str = '  ' * indent + str(t.label) + "\n"
            for b in t.branches:
                tree_str += print_tree(b, indent + 1)
            return tree_str
        return print_tree(self).rstrip()


class Link:
    """A linked list.

    >>> s = Link(1)
    >>> s.first
    1
    >>> s.rest is Link.empty
    True
    >>> s = Link(2, Link(3, Link(4)))
    >>> s.first = 5
    >>> s.rest.first = 6
    >>> s.rest.rest = Link.empty
    >>> s                                    # Displays the contents of repr(s)
    Link(5, Link(6))
    >>> s.rest = Link(7, Link(Link(8, Link(9))))
    >>> s
    Link(5, Link(7, Link(Link(8, Link(9)))))
    >>> print(s)                             # Prints str(s)
    <5 7 <8 9>>
    """
    empty = ()

    def __init__(self, first, rest=empty):
        assert rest is Link.empty or isinstance(rest, Link)
        self.first = first
        self.rest = rest

    def __repr__(self):
        if self.rest is not Link.empty:
            rest_repr = ', ' + repr(self.rest)
        else:
            rest_repr = ''
        return 'Link(' + repr(self.first) + rest_repr + ')'

    def __str__(self):
        string = '<'
        while self.rest is not Link.empty:
            string += str(self.first) + ' '
            self = self.rest
        return string + str(self.first) + '>'
```