```python

# HSLU / ICS/AIML : Modul ADS : Algorithmen & Datenstrukturen
# Path   : uebung07/al/aufgabe01
# Version: Tue Apr  1 10:38:49 CEST 2025

from uebung07.al.aufgabe01.binary_search_tree import BinarySearchTree
from uebung07.al.aufgabe01.binary_search_tree_adv import BinarySearchTreeADV
from uebung07.al.aufgabe01.tree_printer import TreePrinterAcc
from uebung07.al.aufgabe01.tree_printer import TreePrinter

if __name__ == '__main__':

  bst = BinarySearchTree()
  #bst =  BinarySearchTreeADV("Binary-Search-Tree")
  #bst =  BinarySearchTreeADV("Binary-Search-Tree", 3, 2)

  treePrinterAcc = TreePrinterAcc(bst)
  treePrinter = TreePrinter(treePrinterAcc)

  bst.insert(6, "Sechs")
  print("Inorder: " + bst.inorder())
  bst.insert(2, "Zwei")
  print("Inorder: " + bst.inorder())
  bst.insert(9, "Neun")
  print("Inorder: " + bst.inorder())
  bst.insert(1, "Eins")
  print("Inorder: " + bst.inorder())
  bst.insert(4, "Vier")
  print("Inorder: " + bst.inorder())
  bst.insert(8, "Acht")
  print("Inorder: " + bst.inorder())
  bst.insert(5, "Fuenf")
  print("Inorder: " + bst.inorder())
  treePrinter.print()
  print("remove(1): " + bst.remove(1))
  treePrinter.print()
  print("remove(4): " + bst.remove(4))
  treePrinter.print()
  print("remove(5): " + bst.remove(5))
  treePrinter.print()
  bst.insert(1, "Eins")
  bst.insert(4, "Vier")
  bst.insert(5, "Fuenf")
  treePrinter.print()
  print("remove(6): " + bst.remove(6))
  treePrinter.print()


```



```python

""" Session-Log:

Inorder: (6/Sechs)
Inorder: (2/Zwei) (6/Sechs)
Inorder: (2/Zwei) (6/Sechs) (9/Neun)
Inorder: (1/Eins) (2/Zwei) (6/Sechs) (9/Neun)
Inorder: (1/Eins) (2/Zwei) (4/Vier) (6/Sechs) (9/Neun)
Inorder: (1/Eins) (2/Zwei) (4/Vier) (6/Sechs) (8/Acht) (9/Neun)
Inorder: (1/Eins) (2/Zwei) (4/Vier) (5/Fuenf) (6/Sechs) (8/Acht) (9/Neun)
=== Tree: ========================
                    (6,Sechs)
      (2,Zwei)/                \(9,Neun)
(1,Eins)/ \(4,Vier)       (8,Acht)/
               \(5,Fuenf)
=== Tree. ========================
remove(1): Eins
=== Tree: ========================
               (6,Sechs)
(2,Zwei)/                \(9,Neun)
     \(4,Vier)       (8,Acht)/
          \(5,Fuenf)
=== Tree. ========================
remove(4): Vier
=== Tree: ========================
          (6,Sechs)
(2,Zwei)/           \(9,Neun)
     \(5,Fuenf)(8,Acht)/
=== Tree. ========================
remove(5): Fuenf
=== Tree: ========================
     (6,Sechs)
(2,Zwei)/      \(9,Neun)
          (8,Acht)/
=== Tree. ========================
=== Tree: ========================
                    (6,Sechs)
      (2,Zwei)/                \(9,Neun)
(1,Eins)/ \(4,Vier)       (8,Acht)/
               \(5,Fuenf)
=== Tree. ========================
remove(6): Sechs
=== Tree: ========================
               (5,Fuenf)
     (2,Zwei)/           \(9,Neun)
(1,Eins)/ \(4,Vier) (8,Acht)/
=== Tree. ========================

"""
```

```python

# HSLU / ICS/AIML : Modul ADS : Algorithmen & Datenstrukturen
# Path   : uebung07/al/aufgabe01
# Version: Tue Apr  1 10:38:49 CEST 2025


class BinarySearchTree:
  """
  A Binary-Search-Tree with internal nodes which store key/values and
  external nodes as 'Leave-Marker' (without key/values).
  """

  class _Node:

    def __init__(self, bst):
      self._bst = bst  # the Binary-Search-Tree
      self._key = None
      self._value = None
      self._left = None
      self._right = None

    def set_key(self, key):
      self._key = key

    def get_key(self):
      return self._key

    def set_value(self, value):
      self._value = value

    def get_value(self):
      return self._value

    def set_left(self, left):
      self._left = left

    def get_left(self):
      return self._left

    def set_right(self, right):
      self._right = right

    def get_right(self):
      return self._right

    def is_external(self):
      return self._left == None and self._right == None

    def convert_to_internal_node(self, key, value):
      self._key = key
      self._value = value
      self._left = self._bst._new_node()
      self._right = self._bst._new_node()

    # End of class _Node


  def __init__(self):
    self._root = self._new_node()

  def _new_node(self):
    """
    Factory-Method: Creates a new node.

    Returns a new created node.
    """
    return BinarySearchTree._Node(self)


```



```python

  def height(self):
    """
    Calculates the height of this tree.

    Returns the height. For an empty tree: -1
    """
    return self._height(self._root)

  def _height(self, node):
    """
    Calculates recursively the height of the subtree below node.

    node: The root of the subtree.
    Returns the height of this subtree. For an empty tree: -1
    """
    if node.is_external():
      return -1
    height_left  = self._height(node.get_left())
    height_right = self._height(node.get_right())
    return max(height_left, height_right) + 1

  def find(self, key):
    """
    Searches for key in the tree.

    key: The key to search for.
    Returns the associated value or None if key is not found.
    """
    node = self._search(key, self._root, None)
    if not node.is_external():
      value = node.get_value()
    else:
      value = None
    return value

  def _search(self, key, node, path_to_root):
    """
    Searches recursively for key in the subtree with node as root.

    key: The key to search for.
    node: The root of the subtree.
    path_to_root: The path from the returned node to root.
    Returns: If key found the corresponding internal node,
             else the corresponding external node.
    """
    if path_to_root != None:
      path_to_root.insert(0, node)
    if node.is_external():
      return node
    compare_result = key - node.get_key()
    if compare_result < 0:
      return self._search(key, node.get_left(), path_to_root)
    else:
      pass
      if compare_result > 0:
        return self._search(key, node.get_right(), path_to_root)
      else:
        return node  # Key found in this node.

  def insert(self, key, value):
    """
    Inserts a key and its associated value into the tree in a way, that a
    inorder-traverse will return the elements in sorted order.
    """
    node = self._search(key, self._root, None)
    if not node.is_external():
      node.set_value(value)
    else:
      node.convert_to_internal_node(key, value)

```

```
139
140   def inorder(self):
141     """
142     Performs an inorder-traverse and returns all key/values as a string.
143
144     Returns a string with all key/value-pairs of all nodes in inorder.
145     """
146     return self._inorder(self._root, "")
147
148   def _inorder(self, node, inorderString):
149
150     # TODO: Implement here...
151
152     return ""
153
154   def remove(self, key):
155
156     # TODO: Implement here...
157
158     return ""
159
160   def _removeExternal(self, external_node, path_to_root):
161     """
162     Removes an external.
163
164     externalNode: The external Node to delete together with its parent.
165     pathToRoot: The path from the parent of the external node to the root.
166     Returns the value of the parent.
167     """
168
169     # TODO: Implement here...
170
171     return None
172
173   def _max(self, subtree_root, path_to_root):
174     """
175     Searches for the node with the greatest key in this subtree.
176
177     subtree_root: The root of this subtree.
178     path_to_root: The path from subtree to the root.
179     Returns the node with the greatest key in this subtree.
180     """
181
182     # TODO: Implement here...
183
184     return None
185
```