Laporan

Struktur Data  
HashMap

short line



Dosen Pengampu:

[Muchammad Chandra Cahyo Utomo, M.Kom.](https://if.itk.ac.id/profile/dosen/detail/muchammad-chandra-cahyo-utomo-mkom) 199205202019031013

Disusun Oleh :

|  |  |
| --- | --- |
| Guntur Wisnu Saputra | 11211042 |
| Muhammad Insan Kamil | 11211058 |
| Muhammad Ricky Zakaria | 11211062 |
| Ramadhan Djibran Sanjaya | 11211070 |
| Rangga Hermawan | 11211071 |
| Rendy Pernanda | 11211074 |
|  |  |

21 November 2022

# *# Source Code*

|  |  |
| --- | --- |
| No. | HashMap.py |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341  342  343  344  345 | class Node:  def \_\_init\_\_(self, key, value):  self.\_\_parent = None  self.\_\_left = None  self.\_\_right = None  self.\_\_key = key  self.\_\_isRed = False  self.\_\_value = value  def setRed(self, boolean):  self.\_\_isRed = boolean  def setParent(self, parent):  self.\_\_parent = parent  def setLeft(self,left):  self.\_\_left = left  def setRight(self,right):  self.\_\_right = right  def setKey(self, key):  self.\_\_key = key  def setValue(self, value):  self.\_\_value = value  def getRed(self):  return self.\_\_isRed  def getParent(self):  return self.\_\_parent  def getLeft(self):  return self.\_\_left  def getRight(self):  return self.\_\_right  def getKey(self):  return self.\_\_key  def getValue(self):  return self.\_\_value  class HashMap:  def \_\_init\_\_(self,sensitive):  self.nil = Node(0,"nil")  self.nil.setRed(False)  self.nil.setLeft(None)  self.nil.setRight(None)  self.root = self.nil  self.sensitive = sensitive  def add(self, key, value):  key = key if self.sensitive == True else key.lower()  if self.sensitive == False and type(value) == str:  value = value.lower()  new\_node = Node(key, value)  new\_node.setParent(None)  new\_node.setLeft(self.nil)  new\_node.setRight(self.nil)  new\_node.setRed(True)  parent = None  current = self.root  while current != self.nil:  parent = current  if new\_node.getKey() < current.getKey():  current = current.getLeft()  elif new\_node.getKey() > current.getKey():  current = current.getRight()  else:  return  new\_node.setParent(parent)  if parent == None:  self.root = new\_node  elif new\_node.getKey() < parent.getKey():  parent.setLeft(new\_node)  else:  parent.setRight(new\_node)  self.fix\_add(new\_node)  def rotate\_left(self, node):  y = node.getRight()  node.setRight(y.getLeft())  if y.getLeft() != self.nil:  y.getLeft().setParent(node)  y.setParent(node.getParent())  if node.getParent() == None:  self.root = y  elif node == node.getParent().getLeft():  node.getParent().setLeft(y)  else:  node.getParent().setRight(y)  y.setLeft(node)  node.setParent(y)  def rotate\_right(self, node):  y = node.getLeft()  node.setLeft(y.getRight())  if y.getRight() != self.nil:  y.getRight().setParent(node)  y.setParent(node.getParent())  if node.getParent() == None:  self.root = y  elif node == node.getParent().getRight():  node.getParent().setRight(y)  else:  node.getParent().setLeft(y)  y.setRight(node)  node.setParent(y)  def fix\_add(self, new\_node):  while self.root != new\_node and True == new\_node.getParent().getRed():  if new\_node.getParent() == new\_node.getParent().getParent().getLeft():  if new\_node.getParent().getParent().getRight().getRed():  new\_node.getParent().getParent().getRight().setRed(False)  new\_node.getParent().getParent().setRed(True)  new\_node.getParent().setRed(False)  new\_node = new\_node.getParent().getParent()  else:  if new\_node == new\_node.getParent().getRight():  self.rotate\_left( new\_node.getParent() )  new\_node.getParent().setRed(False)  new\_node.getParent().getParent().setRed(True)  self.rotate\_right( new\_node.getParent().getParent() )  else:  if new\_node.getParent().getParent().getLeft().getRed():  new\_node.getParent().getParent().getLeft().setRed(False)  new\_node.getParent().getParent().setRed(True)  new\_node.getParent().setRed(False)  new\_node = new\_node.getParent().getParent()  else:  if new\_node == new\_node.getParent().getLeft():  self.rotate\_right( new\_node.getParent() )  new\_node.getParent().setRed(False)  new\_node.getParent().getParent().setRed(True)  self.rotate\_left( new\_node.getParent().getParent() )  self.root.setRed(False)  def maxKeyNode\_printHelper(self):  current = self.root  while(current.getRight() is not self.nil):  current = current.getRight()  return current  def minKeyNode\_printHelper(self):  current = self.root  while(current.getLeft() is not self.nil):  current = current.getLeft()  return current  def minKeyNode(self, node):  current = node  while(current.getLeft() is not self.nil):  current = current.getLeft()  return current    def transplant(self, deletedNode, replacer):  if deletedNode.getParent() == self.nil:  self.root = replacer  elif deletedNode == deletedNode.getParent().getLeft():  deletedNode.getParent().setLeft(replacer)  else:  deletedNode.getParent().setRight(replacer)  replacer.setParent(deletedNode.getParent())  def delete\_fixup(self, node):  while node != self.root and node.getRed() == False:  if node == node.getParent().getLeft():  siblings = node.getParent().getRight()  if siblings.getRed() == True:  siblings.setRed(False)  node.getParent().setRed(True)  self.rotate\_left(node.getParent())  siblings = node.getParent().getRight()  if siblings.getLeft().getRed() == False and siblings.getRight().getRed() == False:  siblings.setRed(True)  node = node.getParent()  else:  if siblings.getRight().getRed() == False:  siblings.getLeft().setRed(False)  siblings.setRed(True)  self.rotate\_right(siblings)  siblings = node.getParent().getRight()  siblings.setRed(node.getParent().getRed())  node.getParent().setRed(False)  siblings.getRight().setRed(False)  self.rotate\_left(node.getParent())  node = self.root  else:  siblings = node.getParent().getLeft()  if siblings.getRed() == True:  siblings.setRed(False)  node.getParent().setRed(True)  self.rotate\_right(node.getParent())  siblings = node.getParent().getLeft()  if siblings.getRight().getRed() == False and siblings.getLeft().getRed() == False:  siblings.setRed(True)  node = node.getParent()  else:  if siblings.getLeft().getRed() == False:  siblings.getRight().setRed(False)  siblings.setRed(True)  self.rotate\_left(siblings)  siblings = node.getParent().getLeft()  siblings.setRed(node.getParent().getRed())  node.getParent().setRed(False)  siblings.getLeft().setRed(False)  self.rotate\_right(node.getParent())  node = self.root  node.setRed(False)  def delete(self, key):  key = key if self.sensitive == True else key.lower()  if self.search(self.root,key):  deletedNode = self.search(self.root,key)  else:  print(f"Tidak bisa menghapus, key:'{key}' tidak ada")  return  x = None  replacer\_orignal\_color = deletedNode.getRed()  if deletedNode.getLeft() == self.nil:  x = deletedNode.getRight()  self.transplant(deletedNode, deletedNode.getRight())  elif deletedNode.getRight() == self.nil:  x = deletedNode.getLeft()  self.transplant(deletedNode, deletedNode.getLeft())  else:  replacer = self.minKeyNode(deletedNode.getRight())  replacer\_orignal\_color = replacer.getRed()  x = replacer.getRight()  if replacer.getParent()== deletedNode:  x.setParent(deletedNode)  else:  self.transplant(replacer, replacer.getRight())  replacer.setRight(deletedNode.getRight())  replacer.getRight().setParent(replacer)  self.transplant(deletedNode, replacer)  replacer.setLeft(deletedNode.getLeft())  replacer.getLeft().setParent(replacer)  replacer.setRed(deletedNode.getRed())  if replacer\_orignal\_color == False:  self.delete\_fixup(x)  def edit(self, key, value):  key = key if self.sensitive == True else key.lower()  if self.sensitive == False and type(value) == str:  value = value.lower()  self.search(self.root,key).setValue(value)  def get(self, key):  key = key if self.sensitive == True else key.lower()  if self.search(self.root,key):  return self.search(self.root,key).getValue()  else:  return f"Key:'{key}' tidak ada"  def search(self,node, key):  key = key if self.sensitive == True else key.lower()  if key < node.getKey():  if node.getLeft() is self.nil:  return False  return self.search(node.getLeft(),key)  elif key > node.getKey():  if node.getRight() is self.nil:  return False  return self.search(node.getRight(),key)  else:  return node  def printHashMap(self):  return self.printHelper(self.root)    def printHelper(self,node):  if node:  self.printHelper(node.getLeft())  if type(node.getKey()) == str and type(node.getValue()) == str:  if node == self.minKeyNode\_printHelper() and node.getRight() == self.nil and node.getLeft() == self.nil and node == self.root:  txt3 = "{}'{}': '{}'".format("{",node.getKey(),node.getValue())  print(txt3, end="}\n")  elif node == self.minKeyNode\_printHelper():  txt3 = "{}'{}': '{}'".format("{",node.getKey(),node.getValue())  print(txt3, end=", ")  elif node == self.maxKeyNode\_printHelper():  print(f"'{node.getKey()}': '{node.getValue()}'", end="}\n")  else:  print(f"'{node.getKey()}': '{node.getValue()}'", end=", ")  elif type(node.getKey()) == str and type(node.getValue()) != str:  if node == self.minKeyNode\_printHelper():  txt3 = "{}'{}': {}".format("{",node.getKey(),node.getValue())  print(txt3, end=", ")  elif node == self.maxKeyNode\_printHelper():  print(f"'{node.getKey()}': {node.getValue()}", end="}\n")  else:  print(f"'{node.getKey()}': {node.getValue()}", end=", ")  self.printHelper(node.getRight())  Device = HashMap(False)  print("add('Nama Perangkat', 'Redmi Note 8'):")  Device.add("Nama Perangkat", "Redmi Note 8")  Device.printHashMap()  print()  print("add('ROM(GB)',64):")  Device.add("ROM(GB)",64)  Device.printHashMap()  print()  print("add('Versi MIUI', 12):")  Device.add("Versi MIUI", 12)  Device.printHashMap()  print()  print("add('Versi Android', 11):")  Device.add("Versi Android", 11)  Device.printHashMap()  print()  print("add('RAM(GB)', 4):")  Device.add("RAM(GB)", 4)  Device.printHashMap()  print()  print("delete('Versi Android'):")  Device.delete("Versi Android")  Device.printHashMap()  print()  print("delete('CPU'):")  Device.delete("CPU")  print()  print("get('RAM(GB)'):")  print(Device.get("RAM(GB)"))  print()  print("get('Versi Android'):")  print(Device.get("Versi Android")) |

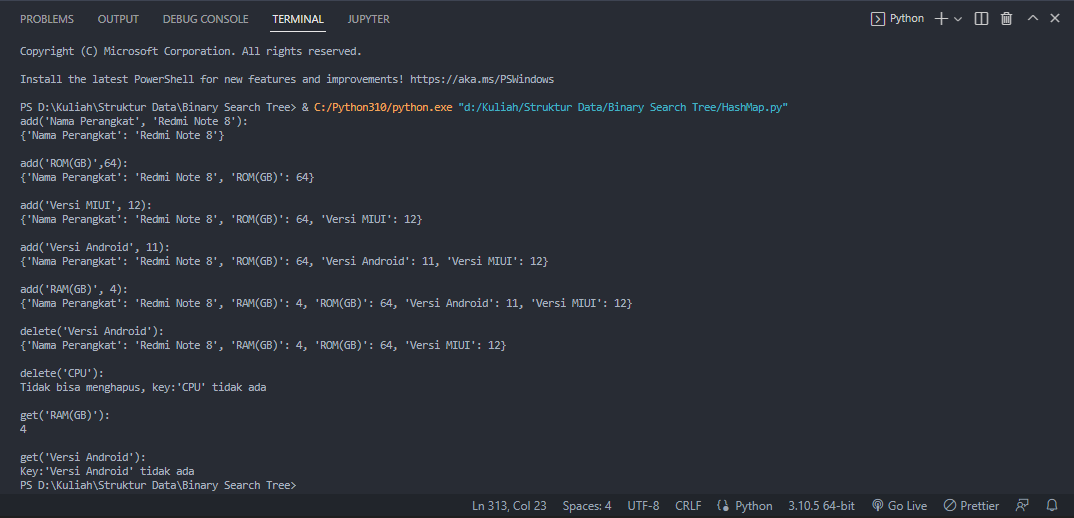
# *#Hasil Run*

# **HashMap.py**

**Case-insensitive**



**Case-sensitive**

****