云南大学数学与统计学院

上机实践报告

|  |  |  |
| --- | --- | --- |
| **课程名称**：数据结构与算法实验 | **年级**：2015级 | **上机实践成绩**： |
| **指导教师**：陆正福 | **姓名**：刘鹏 |  |
| **上机实践名称**：查找树实验 | **学号**：20151910042 | **上机实践日期**：2017-05-14 |
| **上机实践编号**：No.11 | **组号**： | **上机实践时间**：上午3、4节 |

# 一、实验目的

1. 熟悉与查找树有关的数据结构与算法

2. 熟悉主讲教材Chapter 11的代码片段

# 二、实验内容

1. 与查找树有关的数据结构设计与算法设计

2. 调试主讲教材Chapter 11的Python程序

# 三、实验平台

Windows 10 1703 Enterprise 中文版；

Python 3.6.0；

Wing IDE Professional 6.0.5-1集成开发环境。

# 四、实验记录与实验结果分析

|  |  |
| --- | --- |
| 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  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360  361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380  381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400  401  402  403  404  405  406  407  408  409  410  411  412  413  414  415  416  417  418  419  420  421  422  423  424  425  426  427  428  429  430  431  432  433  434  435  436  437  438  439  440  441  442  443  444  445  446  447  448  449  450  451  452  453  454  455  456  457  458  459  460  461  462  463  464  465  466  467  468  469  470  471  472  473  474  475  476  477  478  479  480  481  482  483  484  485  486  487  488  489  490  491  492  493  494  495  496  497  498  499  500  501  502  503  504  505  506  507  508  509  510  511  512  513  514  515  516  517  518  519  520  521  522  523  524  525  526  527  528  529  530  531  532  533  534  535  536  537  538  539  540  541  542  543  544  545  546  547  548  549  550  551  552  553  554 | # 11.6.2 Python Implementation  **class** **Tree:**  """Abstract base class representing a tree structure."""  #------------------- nested Position class -------------------  **class** **Position:**  """An abstraction representing the location of a single element."""  **def** element**(**self**):**  """Return the element stored at this Position."""  **raise** NotImplementedError**(**'must be implemented by subclass'**)**  **def** \_\_eq\_\_**(**self**,**other**):**  """Return True if other Positon represents the same location."""  **raise** NotImplementedError**(**'must be implemented by subclass'**)**  **def** \_\_ne\_\_**(**self**,**other**):**  """Return True if other does not represent the same location."""  **return** **not** **(**self **==** other**)**  #---------- abstract methods that concrete subclass must support ----------  **def** root**(**self**):**  """Return Position representing the tree's root (or None if empty)."""  **raise** NotImplementedError**(**'must be implemented by subclass'**)**  **def** parent**(**self**,**p**):**  """Return Position representing p's parent (or None if p is root)."""  **raise** NotImplementedError**(**'must be implemented by subclass'**)**  **def** num\_children**(**self**,**p**):**  """Return the number of children that Position p has."""  **raise** NotImplementedError**(**'must be implemented by subclass'**)**  **def** children**(**self**,**p**):**  """Generate an iteration of Positions representing p's children."""  **raise** NotImplementedError**(**'must be implemented by subclass'**)**  **def** \_\_len\_\_**(**self**):**  """Return the total number of elements in the tree."""  **raise** NotImplementedError**(**'must be implemented by subclass'**)**  #---------- concrete methods implemented in this class ----------  **def** is\_root**(**self**,**p**):**  """Return True if Position p represents the root of the tree."""  **return** self**.**root**()** **==** p  **def** is\_leaf**(**self**,**p**):**  """Return True if Positoin p does not have any children."""  **return** self**.**num\_children **==** 0  **def** is\_empty**(**self**):**  """Return True if the tree is empty."""  **return** len**(**self**)** **==** 0  #---------- new methods in this section ----------  **def** depth**(**self**,**p**):**  """Return the number of levels separating Positon p from the root."""  **if** self**.**is\_root**(**p**):**  **return** 0  **else:**  **return** 1 **+** self**.**depth**(**self**.**parent**(**p**))**  **def** \_height1**(**self**):** # works, but O(n^2) worst-case time  """Return the height of the tree."""  **return** max**(**self**.**depth**(**p**)** **for** p **in** self**.**positions**()** **if** self**.**is\_leaf**(**p**))**  **def** \_height2**(**self**,**p**):**  """Ruturn the height of the subtree rooted at Postion p."""  **if** self**.**is\_leaf**(**p**):**  **return** 0  **else:**  **return** 1 **+** max**(**self**.**\_height2**(**c**)** **for** c **in** self**.**children**(**p**))**  **def** height**(**self**,**p**=None):**  """Return the height of the subtree rooted ar Position p.  If p is None, return the height of the entire tree.  """  **if** p **is** **None:**  p **=** self**.**root**()**  **return** self**.**\_height2**(**p**)** # start \_height2 recursion  **class** **BinaryTree(**Tree**):**  """Abstract base class representint a bianry tree structure."""  #--------------------- additional abstract methods ---------------------  **def** left**(**self**,**p**):**  """Return a Position representing p's left child.  Return None if p does not have a left child.  """  **raise** NotImplementedError**(**'must be implemented by subclass'**)**  **def** right**(**self**,**p**):**  """Return a Position representing p's right child.  Return None if p does not have a right child.  """  **raise** NotImplementedError**(**'must be implemented by subclass'**)**  #---------- concrete methods implemented in this class ----------  **def** sibling**(**self**,**p**):**  """  Return a Position representing p's sibling  (or None if no sibling).  """  parent **=** self**.**parent**(**p**)**  **if** parent **is** **None:** # p must be the root  **return** **None** # root has no sibling  **else:**  **if** p **==** self**.**left**(**parent**):**  **return** self**.**right**(**parent**)** # possibly None  **else:**  **return** self**.**left**(**parent**)** # possibly None  **def** children**(**self**,**p**):**  """Generate an iteration of Positions representing p's children."""  **if** self**.**left**(**p**)** **is** **not** **None:**  **yield** self**.**left**(**p**)**  **if** self**.**right**(**p**)** **is** **not** **None:**  **yield** self**.**right**(**p**)**  **class** **LinkedBinaryTree(**BinaryTree**):**  """Linked representation of a binary tree structure."""  **class** **\_Node:** # Lightweight, nonpublic class for storing a node.  \_\_slots\_\_ **=** '\_element'**,**'\_parent'**,**'\_left'**,**'\_right'  **def** \_\_init\_\_**(**self**,**element**,**parent**=None,**left**=None,**right**=None):**  self**.**\_element **=** element  self**.**\_parent **=** parent  self**.**\_left **=** left  self**.**\_right **=** right  **class** **Position(**BinaryTree**.**Position**):**  """An abstraction representing the location of a single element."""  **def** \_\_init\_\_**(**self**,**container**,**node**):**  """Constructor should not be invoked by user."""  self**.**\_container **=** container  self**.**\_node **=** node  **def** element**(**self**):**  """Return the element stored at this Position."""  **return** self**.**\_node**.**\_element  **def** \_\_eq\_\_**(**self**,**other**):**  """Return True if other is a Position representing  the same location."""  **return** type**(**other**)** **is** type**(**self**)** **and** other **.**\_node **is** self**.**\_node  **def** \_validate**(**self**,**p**):**  """Return associated node, if position is valid."""  **if** **not** isinstance**(**p**,**self**.**Position**):**  **raise** TypeError**(**'p must be proper Positon type'**)**  **if** p**.**\_container **is** **not** self**:**  **raise** ValueError**(**'p does not belong to this type'**)**  **if** p**.**\_node**.**\_parent **is** p**.**\_node**:** # convention for deprecated nodes  **raise** ValueError**(**'p is not longer valid'**)**  **return** p**.**\_node  **def** \_make\_position**(**self**,**node**):**  """Return Position instance for given node (or None if no node)."""  **return** self**.**Position**(**self**,**node**)if** node **is** **not** **None** **else** **None**  #---------------------- binary tree constructor ----------------------  **def** \_\_init\_\_**(**self**):**  """Create an initially empty binary tree."""  self**.**\_root **=** **None**  self**.**\_size **=** 0  #-------------------------- public accessors --------------------------  **def** \_\_len\_\_**(**self**):**  """Return the total number of elements in the tree."""  **return** self**.**\_size  **def** root**(**self**):**  """Return the root Position of the tree (or None if tree is empty)."""  **return** self**.**\_make\_position**(**self**.**\_root**)**  **def** parent**(**self**,**p**):**  """Return the Position of p's parent (or None if p is root)."""  node **=** self**.**\_validate**(**p**)**  **return** self**.**\_make\_position**(**node**.**\_parent**)**  **def** left**(**self**,**p**):**  """Return the Position of p's left child (or None if no left child)."""  node **=** self**.**\_validate**(**p**)**  **return** self**.**\_make\_position**(**node**.**\_left**)**  **def** right**(**self**,**p**):**  """  Return the Position of p's right child  (or None if no right child).  """  node **=** self**.**\_validate**(**p**)**  **return** self**.**\_make\_position**(**node**.**\_right**)**  **def** num\_children**(**self**,**p**):**  """Return the number of children of Position p."""  node **=** self**.**\_validate**(**p**)**  count **=** 0  **if** node**.**\_left **is** **not** **None:** # left child exists  count **+=** 1  **if** node**.**\_right **is** **not** **None:** # right child exists  count **+=** 1  **return** count  **def** \_add\_root**(**self**,**e**):**  """Place element e at the root of an empty tree and return new Position.  Raise ValueError if tree is nonempty.  """  **if** self**.**\_root **is** **not** **None:** **raise** ValueError**(**'Root exists'**)**  self**.**\_size **=** 1  self**.**\_root **=** self**.**\_Node**(**e**)**  **return** self**.**\_make\_position**(**self**.**\_root**)**  **def** \_add\_left**(**self**,**p**,**e**):**  """Create a new left child for Position p, storing element e.  Return the Position of new node.  Raise ValueError if Position p is invalid or p already has a left child.  """  node **=** self**.**\_validate**(**p**)**  **if** node**.**\_left **is** **not** **None:** **raise** ValueError**(**'Left child exists'**)**  self**.**\_size **+=** 1  node**.**\_left **=** self**.**\_Node**(**e**,**node**)** # node is its parent  **return** self**.**\_make\_position**(**node**.**\_left**)**  **def** \_add\_right**(**self**,**p**,**e**):**  """Create a new right child for Position p, storing element e.  Return the Position of new node.  Raise ValueError if Position p is invalid or p already has  a righr child.  """  node **=** self**.**\_validate**(**p**)**  **if** node**.**\_right **is** **not** **None:** **raise** ValueError**(**'Right chid exists'**)**  self**.**size **+=** 1  node**.**\_right **=** self**.**\_Node**(**e**,**node**)** # node is its parent  **return** self**.**\_make\_position**(**node**.**\_right**)**  **def** \_replace**(**self**,**p**,**e**):**  """Replace the element at position p with e, and return old element."""  node **=** self**.**\_validate**(**p**)**  old **=** node**.**\_element  node**.**\_element **=** e  **return** old  **def** \_delete**(**self**,**p**):**  """Delete the node at Positon p, and replace it with its child, if any.    Return the element that had been stored at Position p.  Raise ValueError if Position p is invalid or p has two children.  """  node **=** self**.**\_validate**(**p**)**  **if** self**.**num\_children**(**p**)** **==** 2**:** **raise** ValueError**(**'p has two children'**)**  child **=** node**.**\_left **if** node**.**\_left **else** node**.**\_right # might be None  **if** child **is** **not** **None:**  child**.**\_parent **=** node**.**\_parent # child's grandparent becomes parent  **if** node **is** self**.**\_root**:**  self**.**\_root **=** child # child becomes root  **else:**  parent **=** node**.**\_parent  **if** node **is** parent**.**\_left**:**  parent**.**\_left **=** child  **else:**  parent**.**right **=** child  self**.**\_size **-=** 1  node**.**\_parent **=** node # convention for deprecated node  **return** node**.**\_element  **def** \_attach**(**self**,**p**,**t1**,**t2**):**  """Atach trees t1 and t2 as left and right subtrees of external p."""  node **=** self**.**\_validate**(**p**)**  **if** **not** self**.**is\_leaf**(**p**):** **raise** ValueError**(**'position must be leaf'**)**  **if** **not** type**(**self**)** **is** type**(**t1**)** **is** type**(**t2**):**  # all 3 trees must be same type  **raise** TypeError**(**'Tree type must match'**)**  self**.**\_size **+=** len**(**t1**)** **+** len**(**t2**)**  **if** **not** t1**.**is\_empty**():** # attached t1 as left subtree of node  t1**.**\_root**.**\_parent **=** node  node**.**\_left **=** t1**.**\_root  t1**.**\_root **=** **None** # set t1 instance to empty  t1**.**\_size **=** 0  **if** **not** t2**.**is\_empty**:** # attached t2 as right subtree of node  t2**.**\_root**.**\_parent **=** node  node**.**\_right **=** t2**.**\_root  t2**.**\_root **=** **None** # set t2 instance to empty  t2**.**\_size **=** 0  **class** **MutableMapping:**  **pass**  **class** **MapBase(**MutableMapping**):**  """Our own abstract base class that includes a nonpublic \_Item class."""  #----------------------- nested \_Item class -----------------------  **class** **\_Item:**  """Lightweight composite to store key-value pairs as map items."""  \_\_slots\_\_ **=** '\_key'**,**'\_value'  **def** \_\_init\_\_**(**self**,**k**,**v**):**  self**.**\_key **=** k  self**.**\_value **=** v  **def** \_\_eq\_\_**(**self**,**other**):**  **return** self**.**\_key **==** other**.**\_key # compare items vased on their keys  **def** \_\_ne\_\_**(**self**,**other**):**  **return** **not** **(**self **==** other**)** # opposite of \_\_eq\_\_  **def** \_\_It\_\_**(**self**,**other**):**  **return** self**.**\_key **<** other**.**\_key# compare items based on their keys  **class** **TreeMap(**LinkedBinaryTree**,**MapBase**):**  """Sorted map implementation using a binary search tree."""    #---------------------------- override Position class ----------------------------  **class** **Positon(**LinkedBinaryTree**.**Position**):**  **def** key**(**self**):**  """Return key of map's key-value pair."""  **return** self**.**element**().**\_key    **def** value**(**self**):**  """Return value of map's key-value pair."""  **return** self**.**element**().**\_value    #---------------------------- nonpublic utilities ----------------------------  **def** \_subtree\_search**(**self**,**p**,**k**):**  """Return Position of p's subtree having key k, or last node searched."""  **if** k **==** p**.**key**():** # found match  **return** p  **elif** k **<** p**.**key**():** # search left subtree  **if** self**.**left**(**p**)** **is** **not** **None:**  **return** self**.**\_subtree\_search**(**self**.**left**(**p**),**k**)**  **else:** # search right subtree  **if** self**.**right**(**p**)** **is** **not** **None:**  **return** self**.**\_subtree\_search**(**self**.**right**(**p**),**k**)**  **return** p # unsuccessful search    **def** \_subtree\_first\_postion**(**self**,**p**):**  """Return Position of first item in subtree rooted at p."""  walk **=** p  **while** self**.**left**(**walk**)** **is** **not** **None:** # keep walking left  walk **=** self**.**left**(**walk**)**  **return** walk    **def** \_subtree\_last\_position**(**self**,**p**):**  """Return Position of last item in subtree rooted at p."""  walk **=** p  **while** self**.**right**(**walk**)** **is** **not** **None:** # keep walking right  walk **=** self**.**right**(**walk**)**  **return** walk    **def** first**(**self**):**  """Return the first Position in the tree (or None if empty)."""  **return** self**.**\_subtree\_first\_postion**(**self**.**root**())** **if** len**(**self**)** **>** 0 **else** **None**    **def** last**(**self**):**  """Return the Position in the tree (or None if empty)."""  **return** self**.**\_subtree\_last\_position**(**self**.**root**())** **if** len**(**self**)** **>** o **else** **None**    **def** before**(**self**,**p**):**  """Return the Position just before p in the natural order.    Return None if p is the first position.  """  self**.**\_validate**(**p**)** # inherit from LinkedBinaryTree  **if** self**.**left**(**p**):**  **return** self**.**\_subtree\_last\_postion**(**self**.**left**(**p**))**  **else:**  # walk upward  walk **=** p  above **=** self**.**parent**(**walk**)**  **while** above **is** **not** **None** **and** walk **==** self**.**left**(**above**):**  walk **=** above  above **=** self**.**parent**(**walk**)**  **return** above    **def** after**(**self**,**p**):**  """Return the Position just after p in the natural order.    Return None if p is the last position.  """  # symmetric to before(p)    **def** find\_position**(**self**,**k**):**  """Return position with key k, or else neighbor (or None if empty)."""  **if** self**.**is\_empty**():**  **return** **None**  **else:**  p **=** self**.**\_subtree\_search**(**self**.**root**(),**k**)**  self**.**\_rebalance\_access**(**p**)** # hook for balanced tree subclasses  **return** p    **def** find\_min**(**self**):**  """Return (key,value) pair with minimum key (or None if empty)."""  **if** self**.**is\_empty**():**  **return** **None**  **else:**  p **=** self**.**first**()**  **return** **(**p**.**key**(),**p**.**value**)**    **def** find\_ge**(**self**,**k**):**  """Return (key,value) pair with least key greater than or equal to k.    Return None if there does not exist such a key.  """  **if** self**.**is\_empty**():**  **return** **None**  **else:**  p **=** self**.**find\_position**(**k**)** # nay not find exact match  **if** p**.**key**()** **<** k**:** # p's key is too small  p **=** self**.**after**(**p**)**  **return** **(**p**.**key**(),**p**.**value**())** **if** p **is** **not** **None** **else** **None**    **def** find\_range**(**self**,**start**,**stop**):**  """Iterate all (key,value) pairs such that start <= key < stop.    If start is None, iteration begins with minimum key of map.  If stop is None, iteration continues through the maximum key of map.  """  **if** **not** self**.**is\_empty**():**  **if** start **is** **None:**  p **=** self**.**first**()**  **else:**  # we initialize p with logic similar to find\_ge  p **=** self**.**find\_position**(**start**)**  **if** p**.**key**()** **<** start**:**  p **=** self**.**after**(**p**)**  **while** p **is** **not** **None** **and** **(**stop **is** **None** **or** p**.**key**()** **<** stop**):**  **yield** **(**p**.**key**(),**p**.**value**)**  p **=** self**.**after**(**p**)**    **def** \_\_getitem\_\_**(**self**,**k**):**  """Return value associated with key k (raise KeyError if not found)."""  **if** self**.**is\_empty**():**  **raise** KeyError**(**'Key Error: ' **+** repr**(**k**))**  **else:**  p **=** self**.**\_subtree\_search**(**self**.**root**(),**k**)**  self**.**\_rebalance\_access**(**p**)** # hook for balanced tree subclasses  **if** k **!=** p**.**key**():**  **raise** KeyError**(**'Key Error: ' **+** repr**(**k**))**  **return** p**.**value**()**    **def** \_\_setitem\_\_**(**self**,**k**,**v**):**  """Assign value v to key k, overwriting existing value if present."""  **if** self**.**is\_empty**():**  leaf **=** self**.**\_add\_root**(**self**.**\_Item**(**k**,**v**))** # from LinkedBinaryTree  **else:**  p **=** self**.**\_subtree\_search**(**self**.**root**(),**k**)**  **if** p**.**key**()** **==** k**:**  p**.**element**().**\_value **=** v # replace existing item's value  self**.**\_rebalance\_access**(**p**)** # hook for balanced tree subclasses  **return**  **else:**  item **=** self**.**\_Item**(**k**,**v**)**  **if** p**.**key**()** **<** k**:**  leaf **=** self**.**\_add\_right**(**p**,**item**)** # inherited from LinkedBinaryTree  **else:**  leaf **=** self**.**\_add\_left**(**p**,**item**)** # inherited from LinkedBinaryTree  **return** self**.**\_rebalance\_insert**(**leaf**)** # hook for balanced tree subclasses    **def** \_\_iter\_\_**(**self**):**  """Generate an iteration of all keys in the map in order."""  p **=** self**.**first**()**  **while** p **is** **not** **None:**  **yield** p**.**key**()**  p **=** self**.**after**(**p**)**    **def** delete**(**self**,**p**):**  """Remove the item at given Position."""  self**.**\_validate**(**p**)** # inherited form LinkedBinaryTree  **if** self**.**left**(**p**)** **and** self**.**right**(**p**):** # p has two children  replacement **=** self**.**\_subtree\_last\_position**(**self**.**left**(**p**))**  self**.**\_replace**(**p**,**replacement**.**element**())** # from LinkedBinaryTree  p **=** replacement  # now p has at most one child  parent **=** self**.**parent**(**p**)**  self**.**\_delete**(**p**)** # inherited form LinkedBinaryTree  self**.**\_rebalance\_delete**(**parent**)** # if root deleted, parent is None    **def** \_\_delitem\_\_**(**self**,**k**):**  """Remove item associated with key k (raise KeyError if not found)."""  **if** **not** self**.**is\_empty**():**  p **=** self**.**\_subtree\_search**(**self**.**root**(),**k**)**  **if** k **==** p**.**key**():**  self**.**delete**(**p**)** # rely on positional list version  **return** # successful deletion complete  self**.**\_reblance\_access**(**p**)** # hook for balanced tree subclasses  **raise** KeyError**(**'Key Error: ' **+** repr**(**k**))**    #----------------------- new methods in this section -----------------------  **def** \_rebalance\_insert**(**self**,**p**):** **pass**  **def** \_rebalance\_delete**(**self**,**p**):** **pass**  **def** \_rebalance\_access**(**self**,**p**):** **pass**    **def** \_relink**(**self**,**parent**,**child**,**make\_left\_child**):**  """Relink parent node with child node (we allow child to be None)."""  **if** make\_left\_child**:** # make it a left child  parent**.**\_left **=** child  **else:** # make it a right child  parent**.**\_right **=** child  **if** child **is** **not** **None:** # make child point to parent  child**.**\_parent **=** parent    **def** \_rotate**(**self**,**p**):**  """Rotate Position p above its parent."""  x **=** p**.**\_node  y **=** x**.**\_parent # we assume this exists  z **=** y**.**\_parent # grandparent (possibly None)  **if** z **is** **None:**  self**.**\_root **=** x # x becomes root  x**.**\_parent **=** **None**  **else:**  self**.**\_relink**(**z**,**x**,**y **==** z**.**\_left**)** # x becomes a direct child of z  # now rotate x and y, including transfer of middle subtree  **if** x **==** y**.**\_left**:**  self**.**\_relink**(**y**,**x**.**\_right**,True)** # x.\_right becomes left child of y  self**.**\_relink**(**x**,**y**,False)** # y becomes right child of x  **else:**  self**.**\_relink**(**y**,**x**.**\_left**,False)** # x.\_left becomes right child of y  self**.**\_relink**(**x**,**y**,True)** # y becomes left child of x    **def** \_restructure**(**self**,**x**):**  """Perform trinode restructure of Position x with parent/grandparent."""  y **=** self**.**parent**(**x**)**  z **=** self**.**parent**(**y**)**  **if** **(**x **==** self**.**right**(**y**))** **==** **(**y **==** self**.**right**(**z**)):** # matching alignments  self**.**\_rotate**(**y**)** # single rotation (of y)  **return** y # y is new subtree root  **else:** # opposite alignments  self**.**\_rotate**(**x**)** # double rotation (of x)  self**.**\_rotate**(**x**)**  **return** x # x is new subtree root  **class** **RedBlackTreeMap(**TreeMap**):**  """Sorted map implementation using a red-black tree."""  **class** **\_Node(**TreeMap**.**\_Node**):**  """Node class for red-black tree maintains bit that denotes color."""  \_\_slots\_\_ **=** '\_red' # add additional data menber to the Node class    **def** \_\_init\_\_**(**self**,**element**,**parent**=None,**left**=None,**right**=None):**  super**().**\_\_init\_\_**(**element**,**parent**,**left**,**right**)**  self**.**\_red **=** **True** # new node red by default  #------------------------- positional-based utility methods -------------------------  # we consider a nonemistent child to be trivially black  **def** \_set\_red**(**self**,**p**):** p**.**\_node**.**\_red **=** **True**  **def** \_set\_black**(**self**,**p**):** p**.**\_node**.**\_red **=** **False**  **def** \_set\_color**(**self**,**p**,**make\_red**):** p**.**\_node**.**\_red **=** make\_red  **def** \_is\_red**(**self**,**p**):** **return** p **is** **not** **None** **and** p**.**\_node**.**\_red  **def** \_is\_red\_leaf**(**self**,**p**):** **return** self**.**\_si\_red**(**p**)** **and** self**.**is\_leaf**(**p**)** |

# 五、教材翻译

# 六、实验体会

# 七、参考文献

[1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Data Structures and Algorithms in Python

[2] 数据结构与算法分析：C语言描述（原书第二版），（美）维斯著；冯舜玺译. 北京：机械工业出版社

[3] 算法导论（原书第三版），（美）科尔曼（Cormen，T.H.）等；殷建平等译. 北京：机械工业出版社