Branch: master ▾ Red-Black-tree-in-python / RBTree.py Find file Copy path

MSingh3012 Update RBTree.py d06d2ff on May 28, 2016

1 contributor

780 lines (629 sloc) | 20.5 KB

```
1
2   #   Red Black Tree implementaion in Python
3   #   Created By Manpreet Singh
4   #
5
6   import string
7
8   BLACK = 0
9   RED = 1
10
11  class RBNode(object):
12
13      def __init__(self, key = None, value = None, color = RED):
14          self.left = self.right = self.parent = None
15          self.color = color
16          self.key = key
17          self.value = value
18          self.nonzero = 1
19
20      def __str__(self):
21          return repr(self.key) + ': ' + repr(self.value)
22
23      def __nonzero__(self):
24          return self.nonzero
25
26      def __len__(self):
27          """imitate sequence"""
28          return 2
29
30      def __getitem__(self, index):
31          """imitate sequence"""
32          if index==0:
33              return self.key
34          if index==1:
35              return self.value
36          raise IndexError('only key and value as sequence')
37
38
39  class RBTreeIter(object):
40
41      def __init__ (self, tree):
42          self.tree = tree
43          self.index = -1  # ready to iterate on the next() call
44          self.node = None
45          self.stopped = False
46
47      def __iter__ (self):
48          """ Return the current item in the container
49          """
50          return self.node.value
51
52      def next (self):
53          """ Return the next item in the container
54              Once we go off the list we stay off even if the list changes
55          """
56          if self.stopped or (self.index + 1 >= self.tree.__len__()):
57              self.stopped = True
58              raise StopIteration
59          #
60          self.index += 1
61          if self.index == 0:
```

```
            self.node = self.tree.firstNode()
        else:
            self.node = self.tree.nextNode (self.node)
        return self.node.value


class RBTree(object):

    def __init__(self, cmpfn=cmp):
        self.sentinel = RBNode()
        self.sentinel.left = self.sentinel.right = self.sentinel
        self.sentinel.color = BLACK
        self.sentinel.nonzero = 0
        self.root = self.sentinel
        self.count = 0
        # changing the comparison function for an existing tree is dangerous!
        self.__cmp = cmpfn

    def __len__(self):
        return self.count

    def __del__(self):
        # unlink the whole tree

        s = [ self.root ]

        if self.root is not self.sentinel:
            while s:
                cur = s[0]
                if cur.left and cur.left != self.sentinel:
                    s.append(cur.left)
                if cur.right and cur.right != self.sentinel:
                    s.append(cur.right)
                cur.right = cur.left = cur.parent = None
                cur.key = cur.value = None
                s = s[1:]

        self.root = None
        self.sentinel = None

    def __str__(self):
        return "<RBTree object>"

    def __repr__(self):
        return "<RBTree object>"

    def __iter__ (self):
        return RBTreeIter (self)

    def rotateLeft(self, x):

        y = x.right

        # establish x.right link
        x.right = y.left
        if y.left != self.sentinel:
            y.left.parent = x

        # establish y.parent link
        if y != self.sentinel:
            y.parent = x.parent
        if x.parent:
            if x == x.parent.left:
                x.parent.left = y
            else:
                x.parent.right = y
        else:
            self.root = y

        # link x and y
        y.left = x
        if x != self.sentinel:
            x.parent = y

```

```
    def rotateRight(self, x):

        #****************************
        #  rotate node x to right
        #****************************

        y = x.left

        # establish x.left link
        x.left = y.right
        if y.right != self.sentinel:
            y.right.parent = x

        # establish y.parent link
        if y != self.sentinel:
            y.parent = x.parent
        if x.parent:
            if x == x.parent.right:
                x.parent.right = y
            else:
                x.parent.left = y
        else:
            self.root = y

        # link x and y
        y.right = x
        if x != self.sentinel:
            x.parent = y

    def insertFixup(self, x):
        #************************************
        #  maintain Red-Black tree balance  *
        #  after inserting node x           *
        #************************************

        # check Red-Black properties

        while x != self.root and x.parent.color == RED:

            # we have a violation

            if x.parent == x.parent.parent.left:

                y = x.parent.parent.right

                if y.color == RED:
                    # uncle is RED
                    x.parent.color = BLACK
                    y.color = BLACK
                    x.parent.parent.color = RED
                    x = x.parent.parent

                else:
                    # uncle is BLACK
                    if x == x.parent.right:
                        # make x a left child
                        x = x.parent
                        self.rotateLeft(x)

                    # recolor and rotate
                    x.parent.color = BLACK
                    x.parent.parent.color = RED
                    self.rotateRight(x.parent.parent)
            else:

                # mirror image of above code

                y = x.parent.parent.left

                if y.color == RED:
                    # uncle is RED
                    x.parent.color = BLACK
                    y.color = BLACK
                    x.parent.parent.color = RED
```

```
                    x = x.parent.parent

                else:
                    # uncle is BLACK
                    if x == x.parent.left:
                        x = x.parent
                        self.rotateRight(x)

                    x.parent.color = BLACK
                    x.parent.parent.color = RED
                    self.rotateLeft(x.parent.parent)

        self.root.color = BLACK

    def insertNode(self, key, value):
        #***********************************************
        #  allocate node for data and insert in tree  *
        #***********************************************

        # we aren't interested in the value, we just
        # want the TypeError raised if appropriate
        hash(key)

        # find where node belongs
        current = self.root
        parent = None
        while current != self.sentinel:
            # GJB added comparison function feature
            # slightly improved by JCG: don't assume that ==
            # is the same as self.__cmp(..) == 0
            rc = self.__cmp(key, current.key)
            if rc == 0:
                return current
            parent = current
            if rc < 0:
                current = current.left
            else:
                current = current.right

        # setup new node
        x = RBNode(key, value)
        x.left = x.right = self.sentinel
        x.parent = parent

        self.count = self.count + 1

        # insert node in tree
        if parent:
            if self.__cmp(key, parent.key) < 0:
                parent.left = x
            else:
                parent.right = x
        else:
            self.root = x

        self.insertFixup(x)
        return x

    def deleteFixup(self, x):
        #************************************
        #  maintain Red-Black tree balance  *
        #  after deleting node x            *
        #************************************

        while x != self.root and x.color == BLACK:
            if x == x.parent.left:
                w = x.parent.right
                if w.color == RED:
                    w.color = BLACK
                    x.parent.color = RED
                    self.rotateLeft(x.parent)
                    w = x.parent.right

                if w.left.color == BLACK and w.right.color == BLACK:
```

```
                    w.color = RED
                    x = x.parent
                else:
                    if w.right.color == BLACK:
                        w.left.color = BLACK
                        w.color = RED
                        self.rotateRight(w)
                        w = x.parent.right

                    w.color = x.parent.color
                    x.parent.color = BLACK
                    w.right.color = BLACK
                    self.rotateLeft(x.parent)
                    x = self.root

            else:
                w = x.parent.left
                if w.color == RED:
                    w.color = BLACK
                    x.parent.color = RED
                    self.rotateRight(x.parent)
                    w = x.parent.left

                if w.right.color == BLACK and w.left.color == BLACK:
                    w.color = RED
                    x = x.parent
                else:
                    if w.left.color == BLACK:
                        w.right.color = BLACK
                        w.color = RED
                        self.rotateLeft(w)
                        w = x.parent.left

                    w.color = x.parent.color
                    x.parent.color = BLACK
                    w.left.color = BLACK
                    self.rotateRight(x.parent)
                    x = self.root

        x.color = BLACK

    def deleteNode(self, z):
        #****************************
        #  delete node z from tree  *
        #****************************

        if not z or z == self.sentinel:
            return

        if z.left == self.sentinel or z.right == self.sentinel:
            # y has a self.sentinel node as a child
            y = z
        else:
            # find tree successor with a self.sentinel node as a child
            y = z.right
            while y.left != self.sentinel:
                y = y.left

        # x is y's only child
        if y.left != self.sentinel:
            x = y.left
        else:
            x = y.right

        # remove y from the parent chain
        x.parent = y.parent
        if y.parent:
            if y == y.parent.left:
                y.parent.left = x
            else:
                y.parent.right = x
        else:
            self.root = x

```

```
        if y != z:
            z.key = y.key
            z.value = y.value

        if y.color == BLACK:
            self.deleteFixup(x)

        del y
        self.count = self.count - 1

    def findNode(self, key):
        #*******************************
        #  find node containing data
        #*******************************

        # we aren't interested in the value, we just
        # want the TypeError raised if appropriate
        hash(key)

        current = self.root

        while current != self.sentinel:
            # GJB added comparison function feature
            # slightly improved by JCG: don't assume that ==
            # is the same as self.__cmp(..) == 0
            rc = self.__cmp(key, current.key)
            if rc == 0:
                return current
            else:
                if rc < 0:
                    current = current.left
                else:
                    current = current.right

        return None

    def traverseTree(self, f):
        if self.root == self.sentinel:
            return
        s = [ None ]
        cur = self.root
        while s:
            if cur.left:
                s.append(cur)
                cur = cur.left
            else:
                f(cur)
                while not cur.right:
                    cur = s.pop()
                    if cur is None:
                        return
                    f(cur)
                cur = cur.right
        # should not get here.
        return

    def nodesByTraversal(self):
        """return all nodes as a list"""
        result = []
        def traversalFn(x, K=result):
            K.append(x)
        self.traverseTree(traversalFn)
        return result

    def nodes(self):
        """return all nodes as a list"""
        cur = self.firstNode()
        result = []
        while cur:
            result.append(cur)
            cur = self.nextNode(cur)
        return result

    def firstNode(self):
```

```
        cur = self.root
        while cur.left:
            cur = cur.left
        return cur

    def lastNode(self):
        cur = self.root
        while cur.right:
            cur = cur.right
        return cur

    def nextNode(self, prev):
        """returns None if there isn't one"""
        cur = prev
        if cur.right:
            cur = prev.right
            while cur.left:
                cur = cur.left
            return cur
        while 1:
            cur = cur.parent
            if not cur:
                return None
            if self.__cmp(cur.key, prev.key)>=0:
                return cur

    def prevNode(self, next):
        """returns None if there isn't one"""
        cur = next
        if cur.left:
            cur = next.left
            while cur.right:
                cur = cur.right
            return cur
        while 1:
            cur = cur.parent
            if cur is None:
                return None
            if self.__cmp(cur.key, next.key)<0:
                return cur


class RBList(RBTree):
    """ List class uses same object for key and value
        Assumes you are putting sortable items into the list.
    """

    def __init__(self, list=[], cmpfn=cmp):
        RBTree.__init__(self, cmpfn)
        for item in list:
            self.insertNode (item, item)

    def __getitem__ (self, index):
        node = self.findNodeByIndex (index)
        return node.value

    def __delitem__ (self, index):
        node = self.findNodeByIndex (index)
        self.deleteNode (node)

    def __contains__ (self, item):
        return self.findNode (item) is not None

    def __str__ (self):
        # eval(str(self)) returns a regular list
        return '['+ string.join(map(lambda x: str(x.value), self.nodes()), ', ')+']'

    def findNodeByIndex (self, index):
        if (index < 0) or (index >= self.count):
            raise IndexError ("pop index out of range")
        #
        if index < self.count / 2:
            # simple scan from start of list
            node = self.firstNode()
```

```
            currIndex = 0
            while currIndex < index:
                node = self.nextNode (node)
                currIndex += 1
        else:
            # simple scan from end of list
            node = self.lastNode()
            currIndex = self.count - 1
            while currIndex > index:
                node = self.prevNode (node)
                currIndex -= 1
        #
        return node

    def insert (self, item):
        node = self.findNode (item)
        if node is not None:
            self.deleteNode (node)
        # item is both key and value for a list
        self.insertNode (item, item)

    def append (self, item):
        # list is always sorted
        self.insert (item)

    def count (self):
        return len (self)

    def index (self, item):
        index = -1
        node = self.findNode (item)
        while node is not None:
            node = self.prevNode (node)
            index += 1
        #
        if index < 0:
            raise ValueError ("RBList.index: item not in list")
        return index

    def extend (self, otherList):
        for item in otherList:
            self.insert (item)

    def pop (self, index=None):
        if index is None:
            index = self.count - 1
        #
        node = self.findNodeByIndex (index)
        value = node.value      # must do this before removing node
        self.deleteNode (node)
        return value

    def remove (self, item):
        node = self.findNode (item)
        if node is not None:
            self.deleteNode (node)

    def reverse (self): # not implemented
        raise AssertionError ("RBlist.reverse Not implemented")

    def sort (self): # Null operation
        pass

    def clear (self):
        """delete all entries"""
        self.__del__()
        #copied from RBTree constructor
        self.sentinel = RBNode()
        self.sentinel.left = self.sentinel.right = self.sentinel
        self.sentinel.color = BLACK
        self.sentinel.nonzero = 0
        self.root = self.sentinel
        self.count = 0

```

```
    def values (self):
        return map (lambda x: x.value, self.nodes())

    def reverseValues (self):
        values = map (lambda x: x.value, self.nodes())
        values.reverse()
        return values


class RBDict(RBTree):

    def __init__(self, dict={}, cmpfn=cmp):
        RBTree.__init__(self, cmpfn)
        for key, value in dict.items():
            self[key]=value

    def __str__(self):
        # eval(str(self)) returns a regular dictionary
        return '{'+ string.join(map(str, self.nodes()), ', ')+'}'

    def __repr__(self):
        return "<RBDict object " + str(self) + ">"

    def __getitem__(self, key):
        n = self.findNode(key)
        if n:
            return n.value
        raise IndexError

    def __setitem__(self, key, value):
        n = self.findNode(key)
        if n:
            n.value = value
        else:
            self.insertNode(key, value)

    def __delitem__(self, key):
        n = self.findNode(key)
        if n:
            self.deleteNode(n)
        else:
            raise IndexError

    def get(self, key, default=None):
        n = self.findNode(key)
        if n:
            return n.value
        return default

    def keys(self):
        return map(lambda x: x.key, self.nodes())

    def values(self):
        return map(lambda x: x.value, self.nodes())

    def items(self):
        return map(tuple, self.nodes())

    def has_key(self, key):
        return self.findNode(key) <> None

    def clear(self):
        """delete all entries"""

        self.__del__()

        #copied from RBTree constructor
        self.sentinel = RBNode()
        self.sentinel.left = self.sentinel.right = self.sentinel
        self.sentinel.color = BLACK
        self.sentinel.nonzero = 0
        self.root = self.sentinel
        self.count = 0

```

```
    def copy(self):
        """return shallow copy"""
        # there may be a more efficient way of doing this
        return RBDict(self)

    def update(self, other):
        """Add all items from the supplied mapping to this one.

        Will overwrite old entries with new ones.

        """
        for key in other.keys():
            self[key] = other[key]

    def setdefault(self, key, value=None):
        if self.has_key(key):
            return self[key]
        self[key] = value
        return value


""" --------------------------------------------------------------------------
    TEST ROUTINES
"""
def testRBlist():
    import random
    print "--- Testing RBList ---"
    print "    Basic tests..."

    initList = [5,3,6,7,2,4,21,8,99,32,23]
    rbList = RBList (initList)
    initList.sort()
    assert rbList.values() == initList
    initList.reverse()
    assert rbList.reverseValues() == initList
    #
    rbList = RBList ([0,1,2,3,4,5,6,7,8,9])
    for i in range(10):
        assert i == rbList.index (i)

    # remove odd values
    for i in range (1,10,2):
        rbList.remove (i)
    assert rbList.values() == [0,2,4,6,8]

    # pop tests
    assert rbList.pop() == 8
    assert rbList.values() == [0,2,4,6]
    assert rbList.pop (1) == 2
    assert rbList.values() == [0,4,6]
    assert rbList.pop (0) == 0
    assert rbList.values() == [4,6]

    # Random number insertion test
    rbList = RBList()
    for i in range(5):
        k = random.randrange(10) + 1
        rbList.insert (k)
    print "    Random contents:", rbList

    rbList.insert (0)
    rbList.insert (1)
    rbList.insert (10)

    print "    With 0, 1 and 10:", rbList
    n = rbList.findNode (0)
    print "    Forwards:",
    while n is not None:
        print "(" + str(n) + ")",
        n = rbList.nextNode (n)
    print

    n = rbList.findNode (10)
    print "    Backwards:",
```

```
    while n is not None:
        print "(" + str(n) + ")",
        n = rbList.prevNode (n)

    if rbList.nodes() != rbList.nodesByTraversal():
        print "node lists don't match"
    print

def testRBdict():
    import random
    print "--- Testing RBDict ---"

    rbDict = RBDict()
    for i in range(10):
        k = random.randrange(10) + 1
        rbDict[k] = i
    rbDict[1] = 0
    rbDict[2] = "testing..."

    print "    Value at 1", rbDict.get (1, "Default")
    print "    Value at 2", rbDict.get (2, "Default")
    print "    Value at 99", rbDict.get (99, "Default")
    print "    Keys:", rbDict.keys()
    print "    values:", rbDict.values()
    print "    Items:", rbDict.items()

    if rbDict.nodes() != rbDict.nodesByTraversal():
        print "node lists don't match"

    # convert our RBDict to a dictionary-display,
    # evaluate it (creating a dictionary), and build a new RBDict
    # from it in reverse order.
    revDict = RBDict(eval(str(rbDict)),lambda x, y: cmp(y,x))
    print "    " + str(revDict)
    print


if __name__ == "__main__":

    import sys

    if len(sys.argv) <= 1:
        testRBlist()
        testRBdict()
    else:

        from distutils.core import setup, Extension

    sys.exit(0)


# end of file.
```