

Branch: master ▾ Red-Black-tree-in-python / RBTTree.py

Find file Copy path

MSingh3012 Update RBTTree.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 RBTIter(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:
```

```

62         self.node = self.tree.firstChild()
63     else:
64         self.node = self.tree.nextNode (self.node)
65     return self.node.value
66
67
68 class RBTREE(object):
69
70     def __init__(self, cmpfn=cmp):
71         self.sentinel = RBNode()
72         self.sentinel.left = self.sentinel.right = self.sentinel
73         self.sentinel.color = BLACK
74         self.sentinel.nonzero = 0
75         self.root = self.sentinel
76         self.count = 0
77         # changing the comparison function for an existing tree is dangerous!
78         self.__cmp = cmpfn
79
80     def __len__(self):
81         return self.count
82
83     def __del__(self):
84         # unlink the whole tree
85
86         s = [ self.root ]
87
88         if self.root is not self.sentinel:
89             while s:
90                 cur = s[0]
91                 if cur.left and cur.left != self.sentinel:
92                     s.append(cur.left)
93                 if cur.right and cur.right != self.sentinel:
94                     s.append(cur.right)
95                 cur.right = cur.left = cur.parent = None
96                 cur.key = cur.value = None
97                 s = s[1:]
98
99         self.root = None
100         self.sentinel = None
101
102     def __str__(self):
103         return "<RBTREE object>"
104
105     def __repr__(self):
106         return "<RBTREE object>"
107
108     def __iter__(self):
109         return RBTREEIter (self)
110
111     def rotateLeft(self, x):
112
113         y = x.right
114
115         # establish x.right link
116         x.right = y.left
117         if y.left != self.sentinel:
118             y.left.parent = x
119
120         # establish y.parent link
121         if y != self.sentinel:
122             y.parent = x.parent
123         if x.parent:
124             if x == x.parent.left:
125                 x.parent.left = y
126             else:
127                 x.parent.right = y
128         else:
129             self.root = y
130
131         # link x and y
132         y.left = x
133         if x != self.sentinel:
134             x.parent = y
135

```

```

136 def rotateRight(self, x):
137
138     #*****
139     # rotate node x to right
140     #*****
141
142     y = x.left
143
144     # establish x.left link
145     x.left = y.right
146     if y.right != self.sentinel:
147         y.right.parent = x
148
149     # establish y.parent link
150     if y != self.sentinel:
151         y.parent = x.parent
152     if x.parent:
153         if x == x.parent.right:
154             x.parent.right = y
155         else:
156             x.parent.left = y
157     else:
158         self.root = y
159
160     # link x and y
161     y.right = x
162     if x != self.sentinel:
163         x.parent = y
164
165 def insertFixup(self, x):
166     #*****
167     # maintain Red-Black tree balance *
168     # after inserting node x          *
169     #*****
170
171     # check Red-Black properties
172
173     while x != self.root and x.parent.color == RED:
174
175         # we have a violation
176
177         if x.parent == x.parent.parent.left:
178
179             y = x.parent.parent.right
180
181             if y.color == RED:
182                 # uncle is RED
183                 x.parent.color = BLACK
184                 y.color = BLACK
185                 x.parent.parent.color = RED
186                 x = x.parent.parent
187
188             else:
189                 # uncle is BLACK
190                 if x == x.parent.right:
191                     # make x a left child
192                     x = x.parent
193                     self.rotateLeft(x)
194
195                 # recolor and rotate
196                 x.parent.color = BLACK
197                 x.parent.parent.color = RED
198                 self.rotateRight(x.parent.parent)
199             else:
200
201                 # mirror image of above code
202
203                 y = x.parent.parent.left
204
205                 if y.color == RED:
206                     # uncle is RED
207                     x.parent.color = BLACK
208                     y.color = BLACK
209                     x.parent.parent.color = RED

```

```

210         x = x.parent.parent
211
212     else:
213         # uncle is BLACK
214         if x == x.parent.left:
215             x = x.parent
216             self.rotateRight(x)
217
218         x.parent.color = BLACK
219         x.parent.parent.color = RED
220         self.rotateLeft(x.parent.parent)
221
222     self.root.color = BLACK
223
224 def insertNode(self, key, value):
225     #*****
226     # allocate node for data and insert in tree *
227     #*****
228
229     # we aren't interested in the value, we just
230     # want the TypeError raised if appropriate
231     hash(key)
232
233     # find where node belongs
234     current = self.root
235     parent = None
236     while current != self.sentinel:
237         # GJB added comparison function feature
238         # slightly improved by JCG: don't assume that ==
239         # is the same as self.__cmp(..) == 0
240         rc = self.__cmp(key, current.key)
241         if rc == 0:
242             return current
243         parent = current
244         if rc < 0:
245             current = current.left
246         else:
247             current = current.right
248
249     # setup new node
250     x = RBNode(key, value)
251     x.left = x.right = self.sentinel
252     x.parent = parent
253
254     self.count = self.count + 1
255
256     # insert node in tree
257     if parent:
258         if self.__cmp(key, parent.key) < 0:
259             parent.left = x
260         else:
261             parent.right = x
262     else:
263         self.root = x
264
265     self.insertFixup(x)
266     return x
267
268 def deleteFixup(self, x):
269     #*****
270     # maintain Red-Black tree balance *
271     # after deleting node x *
272     #*****
273
274     while x != self.root and x.color == BLACK:
275         if x == x.parent.left:
276             w = x.parent.right
277             if w.color == RED:
278                 w.color = BLACK
279                 x.parent.color = RED
280                 self.rotateLeft(x.parent)
281                 w = x.parent.right
282
283         if w.left.color == BLACK and w.right.color == BLACK:

```

```

284         w.color = RED
285         x = x.parent
286     else:
287         if w.right.color == BLACK:
288             w.left.color = BLACK
289             w.color = RED
290             self.rotateRight(w)
291             w = x.parent.right
292
293             w.color = x.parent.color
294             x.parent.color = BLACK
295             w.right.color = BLACK
296             self.rotateLeft(x.parent)
297             x = self.root
298
299     else:
300         w = x.parent.left
301         if w.color == RED:
302             w.color = BLACK
303             x.parent.color = RED
304             self.rotateRight(x.parent)
305             w = x.parent.left
306
307         if w.right.color == BLACK and w.left.color == BLACK:
308             w.color = RED
309             x = x.parent
310         else:
311             if w.left.color == BLACK:
312                 w.right.color = BLACK
313                 w.color = RED
314                 self.rotateLeft(w)
315                 w = x.parent.left
316
317             w.color = x.parent.color
318             x.parent.color = BLACK
319             w.left.color = BLACK
320             self.rotateRight(x.parent)
321             x = self.root
322
323     x.color = BLACK
324
325 def deleteNode(self, z):
326     """*****
327     # delete node z from tree *
328     *****"""
329
330     if not z or z == self.sentinel:
331         return
332
333     if z.left == self.sentinel or z.right == self.sentinel:
334         # y has a self.sentinel node as a child
335         y = z
336     else:
337         # find tree successor with a self.sentinel node as a child
338         y = z.right
339         while y.left != self.sentinel:
340             y = y.left
341
342     # x is y's only child
343     if y.left != self.sentinel:
344         x = y.left
345     else:
346         x = y.right
347
348     # remove y from the parent chain
349     x.parent = y.parent
350     if y.parent:
351         if y == y.parent.left:
352             y.parent.left = x
353         else:
354             y.parent.right = x
355     else:
356         self.root = x
357

```

```

358         if y != z:
359             z.key = y.key
360             z.value = y.value
361
362         if y.color == BLACK:
363             self.deleteFixup(x)
364
365         del y
366         self.count = self.count - 1
367
368     def findNode(self, key):
369         """*****
370         # find node containing data
371         #*****
372
373         # we aren't interested in the value, we just
374         # want the TypeError raised if appropriate
375         hash(key)
376
377         current = self.root
378
379         while current != self.sentinel:
380             # GJB added comparison function feature
381             # slightly improved by JCG: don't assume that ==
382             # is the same as self.__cmp(..) == 0
383             rc = self.__cmp(key, current.key)
384             if rc == 0:
385                 return current
386             else:
387                 if rc < 0:
388                     current = current.left
389                 else:
390                     current = current.right
391
392         return None
393
394     def traverseTree(self, f):
395         if self.root == self.sentinel:
396             return
397         s = [ None ]
398         cur = self.root
399         while s:
400             if cur.left:
401                 s.append(cur)
402                 cur = cur.left
403             else:
404                 f(cur)
405                 while not cur.right:
406                     cur = s.pop()
407                     if cur is None:
408                         return
409                     f(cur)
410                 cur = cur.right
411         # should not get here.
412         return
413
414     def nodesByTraversal(self):
415         """return all nodes as a list"""
416         result = []
417         def traversalFn(x, K=result):
418             K.append(x)
419         self.traverseTree(traversalFn)
420         return result
421
422     def nodes(self):
423         """return all nodes as a list"""
424         cur = self.firstNode()
425         result = []
426         while cur:
427             result.append(cur)
428             cur = self.nextNode(cur)
429         return result
430
431     def firstNode(self):

```

```

432     cur = self.root
433     while cur.left:
434         cur = cur.left
435     return cur
436
437 def lastNode(self):
438     cur = self.root
439     while cur.right:
440         cur = cur.right
441     return cur
442
443 def nextNode(self, prev):
444     """returns None if there isn't one"""
445     cur = prev
446     if cur.right:
447         cur = prev.right
448         while cur.left:
449             cur = cur.left
450         return cur
451     while 1:
452         cur = cur.parent
453         if not cur:
454             return None
455         if self.__cmp(cur.key, prev.key)>=0:
456             return cur
457
458 def prevNode(self, next):
459     """returns None if there isn't one"""
460     cur = next
461     if cur.left:
462         cur = next.left
463         while cur.right:
464             cur = cur.right
465         return cur
466     while 1:
467         cur = cur.parent
468         if cur is None:
469             return None
470         if self.__cmp(cur.key, next.key)<0:
471             return cur
472
473
474 class RBLIST(RBTree):
475     """ List class uses same object for key and value
476     Assumes you are putting sortable items into the list.
477     """
478
479     def __init__(self, list=[], cmpfn=cmp):
480         RBTree.__init__(self, cmpfn)
481         for item in list:
482             self.insertNode (item, item)
483
484     def __getitem__ (self, index):
485         node = self.findNodeByIndex (index)
486         return node.value
487
488     def __delitem__ (self, index):
489         node = self.findNodeByIndex (index)
490         self.deleteNode (node)
491
492     def __contains__ (self, item):
493         return self.findNode (item) is not None
494
495     def __str__ (self):
496         # eval(str(self)) returns a regular list
497         return '['+ string.join(map(lambda x: str(x.value), self.nodes()), ', ')+']'
498
499     def findNodeByIndex (self, index):
500         if (index < 0) or (index >= self.count):
501             raise IndexError ("pop index out of range")
502         #
503         if index < self.count / 2:
504             # simple scan from start of list
505             node = self.firstNode()

```

```

506         currIndex = 0
507         while currIndex < index:
508             node = self.nextNode (node)
509             currIndex += 1
510     else:
511         # simple scan from end of list
512         node = self.lastNode()
513         currIndex = self.count - 1
514         while currIndex > index:
515             node = self.prevNode (node)
516             currIndex -= 1
517     #
518     return node
519
520 def insert (self, item):
521     node = self.findNode (item)
522     if node is not None:
523         self.deleteNode (node)
524     # item is both key and value for a list
525     self.insertNode (item, item)
526
527 def append (self, item):
528     # list is always sorted
529     self.insert (item)
530
531 def count (self):
532     return len (self)
533
534 def index (self, item):
535     index = -1
536     node = self.findNode (item)
537     while node is not None:
538         node = self.prevNode (node)
539         index += 1
540     #
541     if index < 0:
542         raise ValueError ("RBLlist.index: item not in list")
543     return index
544
545 def extend (self, otherList):
546     for item in otherList:
547         self.insert (item)
548
549 def pop (self, index=None):
550     if index is None:
551         index = self.count - 1
552     #
553     node = self.findNodeByIndex (index)
554     value = node.value # must do this before removing node
555     self.deleteNode (node)
556     return value
557
558 def remove (self, item):
559     node = self.findNode (item)
560     if node is not None:
561         self.deleteNode (node)
562
563 def reverse (self): # not implemented
564     raise AssertionError ("RBLlist.reverse Not implemented")
565
566 def sort (self): # Null operation
567     pass
568
569 def clear (self):
570     """delete all entries"""
571     self.__del__()
572     #copied from RBTree constructor
573     self.sentinel = RBNode()
574     self.sentinel.left = self.sentinel.right = self.sentinel
575     self.sentinel.color = BLACK
576     self.sentinel.nonzero = 0
577     self.root = self.sentinel
578     self.count = 0
579

```



```

580     def values (self):
581         return map (lambda x: x.value, self.nodes())
582
583     def reverseValues (self):
584         values = map (lambda x: x.value, self.nodes())
585         values.reverse()
586         return values
587
588
589 class RBDict(RBTree):
590
591     def __init__(self, dict={}, cmpfn=cmp):
592         RBTree.__init__(self, cmpfn)
593         for key, value in dict.items():
594             self[key]=value
595
596     def __str__(self):
597         # eval(str(self)) returns a regular dictionary
598         return '{'+ string.join(map(str, self.nodes()), ', ')+'}'
599
600     def __repr__(self):
601         return "<RBDict object " + str(self) + ">"
602
603     def __getitem__(self, key):
604         n = self.findNode(key)
605         if n:
606             return n.value
607         raise IndexError
608
609     def __setitem__(self, key, value):
610         n = self.findNode(key)
611         if n:
612             n.value = value
613         else:
614             self.insertNode(key, value)
615
616     def __delitem__(self, key):
617         n = self.findNode(key)
618         if n:
619             self.deleteNode(n)
620         else:
621             raise IndexError
622
623     def get(self, key, default=None):
624         n = self.findNode(key)
625         if n:
626             return n.value
627         return default
628
629     def keys(self):
630         return map(lambda x: x.key, self.nodes())
631
632     def values(self):
633         return map(lambda x: x.value, self.nodes())
634
635     def items(self):
636         return map(tuple, self.nodes())
637
638     def has_key(self, key):
639         return self.findNode(key) <> None
640
641     def clear(self):
642         """delete all entries"""
643
644         self.__del__()
645
646         #copied from RBTree constructor
647         self.sentinel = RBNode()
648         self.sentinel.left = self.sentinel.right = self.sentinel
649         self.sentinel.color = BLACK
650         self.sentinel.nonzero = 0
651         self.root = self.sentinel
652         self.count = 0
653

```

```

654     def copy(self):
655         """return shallow copy"""
656         # there may be a more efficient way of doing this
657         return RBDict(self)
658
659     def update(self, other):
660         """Add all items from the supplied mapping to this one.
661
662         Will overwrite old entries with new ones.
663
664         """
665         for key in other.keys():
666             self[key] = other[key]
667
668     def setdefault(self, key, value=None):
669         if self.has_key(key):
670             return self[key]
671         self[key] = value
672         return value
673
674
675     """ -----
676     TEST ROUTINES
677     """
678     def testRBLlist():
679         import random
680         print "---- Testing RBLlist ----"
681         print "    Basic tests..."
682
683         initList = [5,3,6,7,2,4,21,8,99,32,23]
684         rbList = RBLlist (initList)
685         initList.sort()
686         assert rbList.values() == initList
687         initList.reverse()
688         assert rbList.reverseValues() == initList
689         #
690         rbList = RBLlist ([0,1,2,3,4,5,6,7,8,9])
691         for i in range(10):
692             assert i == rbList.index (i)
693
694         # remove odd values
695         for i in range (1,10,2):
696             rbList.remove (i)
697         assert rbList.values() == [0,2,4,6,8]
698
699         # pop tests
700         assert rbList.pop() == 8
701         assert rbList.values() == [0,2,4,6]
702         assert rbList.pop (1) == 2
703         assert rbList.values() == [0,4,6]
704         assert rbList.pop (0) == 0
705         assert rbList.values() == [4,6]
706
707         # Random number insertion test
708         rbList = RBLlist()
709         for i in range(5):
710             k = random.randrange(10) + 1
711             rbList.insert (k)
712         print "    Random contents:", rbList
713
714         rbList.insert (0)
715         rbList.insert (1)
716         rbList.insert (10)
717
718         print "    With 0, 1 and 10:", rbList
719         n = rbList.findNode (0)
720         print "    Forwards:",
721         while n is not None:
722             print "(" + str(n) + ")",
723             n = rbList.nextNode (n)
724         print
725
726         n = rbList.findNode (10)
727         print "    Backwards:",

```

```

728     while n is not None:
729         print "(" + str(n) + ")",
730         n = rbList.prevNode (n)
731
732     if rbList.nodes() != rbList.nodesByTraversal():
733         print "node lists don't match"
734     print
735
736 def testRBDict():
737     import random
738     print "---- Testing RBDict ----"
739
740     rbDict = RBDict()
741     for i in range(10):
742         k = random.randrange(10) + 1
743         rbDict[k] = i
744     rbDict[1] = 0
745     rbDict[2] = "testing..."
746
747     print "    Value at 1", rbDict.get (1, "Default")
748     print "    Value at 2", rbDict.get (2, "Default")
749     print "    Value at 99", rbDict.get (99, "Default")
750     print "    Keys:", rbDict.keys()
751     print "    values:", rbDict.values()
752     print "    Items:", rbDict.items()
753
754     if rbDict.nodes() != rbDict.nodesByTraversal():
755         print "node lists don't match"
756
757     # convert our RBDict to a dictionary-display,
758     # evaluate it (creating a dictionary), and build a new RBDict
759     # from it in reverse order.
760     revDict = RBDict(eval(str(rbDict)), lambda x, y: cmp(y,x))
761     print "    " + str(revDict)
762     print
763
764
765 if __name__ == "__main__":
766
767     import sys
768
769     if len(sys.argv) <= 1:
770         testRBList()
771         testRBDict()
772     else:
773
774         from distutils.core import setup, Extension
775
776     sys.exit(0)
777
778
779 # end of file.

```