AVL-tree type for Python

Release 1.1

R. McGraw

typeset December 18, 2008

dasnar@fastmail.fm

Abstract

This document describes how to use the `avl` extension module (dynamically loadable library) for Python, which implements a dual-personality object using AVL trees. AVL trees (named after the inventors, Adel'son-Vel'skiĭ and Landis) are balanced binary search trees. While these objects can be seen as implementing ordered containers, allowing fast lookup and deletion of any item, they can also act as sequential lists. The `avl` module is based on a C written library. It is based on an extension module by Sam Rushing ([1]) and on Ben Pfaff's `libavl` ([2]).

Contents

1 `avl` — AVL-tree type for Python

The `avl` module defines dual-personality objects for Python programmers to enjoy. An object of type `'avl_tree'` can be seen as implementing a dictionary, or it can act as a sequential list since the underlying implementation maintains a `RANK` field at each node in the tree. Contrary to objects of type `'dict'` which record key/value pairs, objects of type `'avl_tree'` record single values, hereinafter referred to as 'items'; it's possible to insert duplicates.

The `avl` module defines exactly these functions :

`new`([*source*[, *compare*[, *unique*]]])

Create a new tree. It is created empty if no argument is passed. The optional *source* argument can be either of type `'list'` or `'avl_tree'` : in the former case, each item from the list is inserted into the tree; in the latter case, a copy of *object* is returned. Note that if *source* is a list which is known to be sorted with respect to some *compare* function, it is more efficient to call '`avl.from_iter(iter(source),len(source),compare)`', see below.

The optional *compare* argument is a Python function that will be used to order the tree instead of the default built-in mechanism.

If *`bool`==1* and source object is a list, duplicates are removed (default: 0). For example,

```
>>> import avl
>>> a = avl.new([2,1,2], None, 1)
>>> type(a)
<type 'avl_tree'>
>>> a
[1, 2]
```

dump(*tree,pickler*)
Convenience function to pickle a tree, as in `tree.dump(pickler)`. First we pickle the size of the tree as a `PyInt` object, then its compare function, then each item in order by `pickler.dump()`. The `cPickle` module has no exported API. The advantage of visiting the tree in inorder is that no comparison is necessary at unpickling time.

Note that *pickler* is not type-checked (the function only checks for the existence of a callable '*dump*' attribute).

load(*unpickler*)
Convenience function to unpickle a tree which was pickled with the simple method applied by `avl.dump` (see above).

Note that *unpickler* is not type-checked (the function only checks for the existence of a callable '*load*' attribute).

from_iter(*iter*[, *len*],[, *compare*])
Load a tree from *iter* if the sequence is in sorted order with respect to some *compare* function. This can't be reproduced in Python since it relies on `avl_xload` in `avl.c` (which proceeds recursively like `avl_slice` and has to know the items count in advance). If no *len* is specified, it will be read by the first call to `iter.next()`. It can be specified as an `intobject` or a `longobject`. If no *compare* function is specified, `None` is assumed.

A `StopIteration` exception occurs if *len* is too large.

Note that *iter* is not type-checked (the function only checks for the existence of a callable '*next*' attribute).

exception avl.Error
Exception raised when an operation fails for some `avl` specific reason. The exception argument is a string describing the reason for failure or just where it occurred.

1.1 'avl_tree' objects

AVL objects, as returned by `new()` above, have the following functions **Note:** functions which do comparisons to do their work will raise an exception as soon as some comparison procedure fails.

First, here is a function that is included by defining the `HAVE_AVL_VERIFY` compile flag :

verify()
Verify internal AVL tree structure, including *ordering*. Return `1` if tree is valid, `0` otherwise.

lookup(*item*)
Return new reference to an item in the tree that compares equal to passed *item*. Raise a `LookupError` exception if *item* is not contained in the tree.

insert(*item*[, *index*])
Insert *item* in the tree. If no *index* is specified, *item* is inserted based on its rank with respect to the built-in Python compare function `PyObject_Compare()`, or a possible user-supplied one. Indeed items of any type can be inserted as long as they are comparable to one another. If some error occurs during a compare operation, the tree remains unchanged and an exception is raised. Otherwise insertion is carried out whether or not an item comparing equal to *item* is already present in the tree.

Use the call '`t.insert(o,j)`' to insert `o` in front of index `j` in `t` regardless of order, if it's what you really want (an `IndexError` exception is raised if `j` is out of range). **Note:** order may be broken as a result. See also the `span()` method.

append(*item*)
Shortcut to append *item* to tree regardless of order.

remove(*item*)
Remove *item* from the tree. Nothing is done if *item* is not found.

clear()
Make the tree logically empty in one sweep.

remove_at(*index*)
Remove item in front of specified *index*. An `IndexError` exception is raised if *index* is out of range.

has_key(*item*)
Return `1` if *item* is in tree, `0` otherwise.

index(*item*)
'`t.index(item)`' returns smallest index `j` such that `t[j] ==` *item*, or `-1` if *item* is not in `t`.

concat(*arg*)
In-place concatenation : append tree *arg* to the tree, regardless of order.

min()
Return smallest item in tree, except if it's empty.

max()
Return greatest item in tree, except if it's empty.

at_least(*item*)
Return smallest item that compares greater than or equal to *item* if any, or raise a `ValueError` exception.

at_most(*item*)
Return greatest item that compares less than or equal to *item* if any, or raise a `ValueError` exception.

span(*o1*[, *o2*])
'`t.span(o1)`' returns a pair of indices `(i,j)` such that `t[i:j]` is the longest slice that spans `o1` if it's in `t`, otherwise `i==j` in which '`t.insert(o1,j)`' puts `o1` where it should be with no need to redo comparisons.

'`t.span(o1,o2)`' returns `(i,j)` such that `t[i:j]` is the longest slice that spans `o1,o2`. For example (see below for slice support),

```
>>> a= avl.new(map (lambda x: random.randint(0,1000), range(10)))
>>> a
[28, 66, 82, 95, 109, 114, 268, 335, 761, 851]
>>> a.span(100,500)
(4, 8)
>>> a[4:8]
[109, 114, 268, 335]
>>> a.span(500,100)
(4, 8)
>>> a.span(900,950)
(10, 10)
>>> len(a)
10
>>> a.span(a[0],a[-1])
(0, 10)
>>> a.span(95,109)
(3, 5)
```

dump(*pickler*)
See `avl.dump` module function.

iter([*pre_or_post*])

Return a new iterator over the items in this tree, either in pre-position if *pre_or_post* is zero/false (the default), or in post-position.

There is support for the **sequence** protocol.

- 'len(t)' returns the size of tree t.
- 'a+b' returns a new tree object resulting from the concatenation of trees 'a' and 'b'. This is done regardless of order.
- The repeat operation is undefined.
- 't[j]' returns a new reference to the item whose inorder index in t is j (starting from 0). The usual defaults apply : j should be in range [-len(t):len(t)], otherwise an IndexError exception is raised.
- A **slice** of a tree can be obtained as a new tree, with the usual defaults. Thus 'b = a[:]' is equivalent to 'b = avl.new(a)'. No exception.
- Item and slice assignments are undefined.
- 'o in t' is equivalent to 't.has_key(o)'.
- To perform in-place concatenation please call 'a += b' or 'a.concat(b)'.

1.2 Support for the **iterator** protocol: 'avl_tree_iterator' objects

See Also:

PEP 234, "*Iterators*"
New protocol in Python 2.2

The avl module implements the iterator protocol. Objects of type 'avl_tree_iterator' are implicitly called upon if a for loop is used to iterate over a tree, like in 'for o in t: print o'. This is more efficient than retrieving the i^{th} item for all i.

There are also a couple of explicit methods.

To get a new iterator (in pre-position) for some tree t, say 'iter(t)' in Python :

```
>>> j = iter(t)
>>> type(j)
<type 'avl_tree_iterator'>
```

Note that it increments the tree object's refcount.

To obtain an iterator in either pre- or post-position, use the iter instance method, see above. The call 't.iter()' is equivalent to 'iter(t)'.

Here are the methods :

next()
Return new reference to next item if there is one, or raise a StopIteration exception. This is part of the protocol.

prev()
Symmetrically, return new reference to previous item. This is not part of the protocol.

index()
Return inorder index of current item in iteration, or -1 if it's in 'pre-position', i.e. right before the first item, or tree's len if it's in 'post-position'.

Since the tree implementation maintains a parent pointer for each node, any iterator remains able to proceed if insertions are done while it's in use, or if any item ***other than the current one*** is deleted via the `remove()` tree object method. The module provides another iterator method :

`remove``()`

Remove current item in iteration, or raise an `avl.Error` exception. Current iterator position is set to next position or in post-position. This is not part of the protocol.

1.3 Example

The following example demonstrates usage of the `avl` module.

```
Python 2.3 (#1, Sep 13 2003, 00:49:11)
[GCC 3.3 20030304 (Apple Computer, Inc. build 1495)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import random
>>> import avl

# Create new avl_tree:
>>> t = avl.new()
>>> type(t)
<type 'avl_tree'>

# Insert new items:
>>> for x in range(20): t.insert(random.randint(0, 100))
...
# verify() --> 1 if tree is valid, 0 otherwise
>>> t.verify()
1
>>> t
[2, 5, 9, 18, 24, 25, 29, 42, 45, 51, 58, 58, 60, 64, 66, 80, 85, 87, 92, 99]

# Lookup by index with usual syntax:
>>> t[0], t.min()
(2, 2)
>>> t[-1], t.max()
(99, 99)
>>> t[4], t[-16]
(24, 24)

# Create a tree from list of items:
>>> list = [3,8,3,5,1,2,8,7]
>>> u = avl.new(list)
# As a result, list is sorted ; this is an _unstable_ sort
>>> u
[1, 2, 3, 3, 5, 7, 8, 8]
# A second optional argument to new() is a Python compare function:
>>> u= avl.new(list, None)
>>> u
[1, 2, 3, 3, 5, 7, 8, 8]
# Same as above except duplicates are removed:
>>> u = avl.new(list, None, 1)
>>> u
[1, 2, 3, 5, 7, 8]

# Lookup by key:
>>> t.lookup(42)
42
# Failure:
>>> t.lookup(36)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
LookupError: 36

# Smallest index of item, or -1:
>>> t.index(58)
10
>>> i,j = t.span(58); i,j
(10, 12)
>>> t[10:12]
[58, 58]
>>> t.span(36)
(7, 7)
# Put it where it should be without comparing:
>>> t.insert(36,7); t.verify()
1
>>> t.index(36)
7

# Slices with usual defaults:
>>> u = t[:10]
>>> type(u)
<type 'avl_tree'>
>>> u
[2, 5, 9, 18, 24, 25, 29, 36, 42, 45]
>>> 42 in u
True
# otherwise put:
>>> u.has_key(42)
1

# new() accepts a tree as argument
# this is equivalent to `a = u[:]':
>>> a = avl.new(u)
>>> a.verify()
1
>>> b = avl.new(range(60,70))

### Concatenation with usual syntax:
>>> c = a + b
>>> c.verify()
1
>>> c
[2, 5, 9, 18, 24, 25, 29, 36, 42, 45, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69]

# Concatenation is done regardless of order:
>>> avl.new([5,1,2]) + avl.new([2,8,6])
[1, 2, 5, 2, 6, 8]

# In-place concatenation:
>>> a.concat(b)
>>> a
[2, 5, 9, 18, 24, 25, 29, 36, 42, 45, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69]

### ITERATION:
>>> n = 0
>>> for o in u:
...         print n, ":", o
...         n += 1
...
0 : 2
1 : 5
2 : 9
3 : 18
4 : 24
5 : 25
6 : 29
```

```
7 : 36
8 : 42
9 : 45

### Explicit iteration:
>>> j = iter(u)
>>> type(j)
<type 'avl_tree_iterator'>
>>> j.next()
2
>>> j.next()
5
>>> u
[2, 5, 9, 18, 24, 25, 29, 36, 42, 45]
>>> j.index()
1
>>> u[2]
9
>>> u.insert(11)
>>> j.next()
9
>>> j.next()
11
>>> j.cur()
11
>>> j.remove()
>>> j.next()
18
>>> 11 in u
False
>>> u.verify()
1
# Get previous item:
>>> j.prev()
9

### Removing items:

# Remove u[3]:
>>> u.remove_at(3)
>>> u.verify()
1
# Remove by key:
>>> u.remove(24)
>>> u.remove(36)
>>> u
[2, 5, 9, 25, 29, 42, 45]
>>> u.verify()
1
# Nothing is done if item not found:
>>> u.remove(20)
>>> u
[2, 5, 9, 25, 29, 42, 45]

### at_least() and at_most(), with exceptions:
>>> u.at_least(30)
42
>>> u.at_most(30)
29
>>> u.at_least(50)
Traceback (most recent call last):
  File "<input>", line 1, in ?
ValueError: 50

>>> u.span(1,9)
```

```
(0, 3)
>>> u[0:3]
[2, 5, 9]
>>> u.span(2,8)
(0, 2)
>>> u.span(30,35)
(5, 5)
>>> u[5]
42
>>> u.span(25,44)
(3, 6)
>>> u[3:6]
[25, 29, 42]
>>> u.span(24,44)
(3, 6)
>>> u.span(24, 45)
(3, 7)
>>> u[3:7]
[25, 29, 42, 45]
>>> u.span(26,45)
(4, 7)
>>>
```

References

[1] Sam Rushing's own iterative C implementation, on which this one is based, is to be found at http://www.python.org/ftp/python/contrib-09-Dec-1999/DataStructures/avl.README

[2] Good ideas came from browsing Ben Pfaff's GNU libavl home at http://adtinfo.org/

[3] Adel'son-Vel'skiĭ (G.M.) and Landis (E.M.) — *An algorithm for the organization of information.* Soviet Mathematics Doklady, vol. 3, 1962, pp. 1259–1263

Index