

A skip list container class in Python



An alternative to balanced trees

John W. Shipman

2009-10-10 13:35

Abstract

Describes a module in the Python programming language that implements a “skip list”, a data structure for storing and retrieving sorted elements in $O(\log n)$ time.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to tcc-doc@nmt.edu.

Table of Contents

1. Why skip lists?	2
2. An implementation of skip lists in Python	2
2.1. Definitions	2
2.2. Creating a Python skip list	3
2.3. Methods and attributes of the SkipList object	4
3. Internals of SkipList	5
3.1. The skip list data structure	5
3.2. Skip list algorithms	6
3.3. Iterators: toward a more Pythonic class	8
3.4. Classes in this module	9
4. The source code	9
4.1. Declarations	9
4.2. Verification functions	10
4.3. The _SkipItem internal class	13
4.4. The _SkipListIterator class	14
4.5. The SkipList class	15
4.6. SkipList.__init__()	17
4.7. SkipList.__estimateLevels()	18
4.8. SkipList.insert()	19
4.9. SkipList.__keyOf()	20
4.10. SkipList.__insertCutList()	20
4.11. SkipList.__insertPoint()	21
4.12. SkipList.__insertionPrecedes()	22
4.13. SkipList.__compareItemKey()	23
4.14. SkipList.__insertItem()	24
4.15. SkipList.__pickLevel()	25
4.16. SkipList.__insertRelink()	25
4.17. SkipList.delete()	26

¹ <http://www.nmt.edu/tcc/help/lang/python/examples/pyskip/>

² <http://www.nmt.edu/tcc/help/lang/python/examples/pyskip/pyskip.pdf>

4.18. <code>SkipList.__searchCutList()</code>	28
4.19. <code>SkipList.__searchPoint()</code>	29
4.20. <code>SkipList.__searchPrecedes()</code>	30
4.21. <code>SkipList.match()</code>	30
4.22. <code>SkipList.__searchCutItem</code>	31
4.23. <code>SkipList.find()</code>	32
4.24. <code>SkipList.__len__()</code>	32
4.25. <code>SkipList.__iter__()</code>	32
4.26. <code>SkipList.__contains__()</code>	33
4.27. <code>SkipList.__delitem__()</code>	33
4.28. <code>SkipList.__getitem__()</code>	33

1. Why skip lists?

A skip list is a data structure for representing an ordered set of objects so that insertion and retrieval take minimal time.

This ingenious data structure was described in:

Pugh, William. Skip lists: a probabilistic alternative to balanced trees. In: Communications of the ACM, June 1990, 33(6)668-676.

Like balanced tree structures, skip lists support search whose time complexity varies as the log of the number of elements. However, compared to 2-3 trees and other tree-oriented techniques, skip lists require much simpler algorithms and much less storage overhead within the data structure.

There is also a probabilistic element in the construction of a skip list that makes it an interesting example of the useful of randomness in algorithms.

2. An implementation of skip lists in Python

The Python module `pyskip.py` is an implementation of Pugh's skip list data structure.

2.1. Definitions

It is necessary to define a few terms before we explore the use and construction of skip lists.

2.1.1. The child domain

A child object is any object that you want to store in a skip list. Typically all child objects are the same type, but there's nothing preventing you from using more than one child object type in a skip list, if you need to.

By *child domain* we mean the type (or types) of child objects.

2.1.2. The key domain

Since skip lists are designed to allow rapid lookup of random child objects, and because child objects are kept in sorted order, we must define how two child objects are to be compared so they can be placed in order.

By *key* we mean the value used to compare two child objects. Depending on the application, the key may be a value stored inside the child object, or the key may be the child object itself, or a value derived from the child object.

For example, you might have a set of **FishLicense** objects representing fish licenses, each containing a unique ten-character license number. You might use a skip list to hold this set of objects, and designate the license number as the key. Then you could rapidly retrieve fish licenses by number.

We use the term *key domain* to mean the set of possible key values.

2.1.3. Stability

If duplicates are allowed, equal members will be kept in their order of insertion. For example, if you insert three equal elements *a*, *b*, and *c* into a skip list in that order, if you then iterate over the members of the list, those three elements will be returned in the same order.

This property of a container class is called *stability*. The term comes from sorting theory: a sort that does not perturb the order of elements with equal keys is called a *stable sort*.

2.2. Creating a Python skip list

Typically you will import the Python skip list module as:

```
import pyskip
```

To create a new, empty skip list, use this syntax:

```
pyskip.SkipList ( keyFun=None, cmpFun=None, allowDups=0,  
                 count=1000000 )
```

where the arguments have these values:

keyFun

This argument is a function that takes a child argument and returns the corresponding key value. The default value, **None**, causes **SkipList** to treat the child elements themselves as keys.

Suppose, for example, that you are building a skip list full of **HockeyTeam** objects, and each object has an attribute **.teamName** that contains the team's name as a string. You could write that key function as:

```
def teamKey(hockeyTeam):  
    return hockeyTeam.teamName
```

Then to create the skip list you'd say something like:

```
skip = SkipList ( keyFun=teamKey )
```

and the objects would be ordered alphabetically by team name.

To make the ordering case-insensitive, you could write the key function like this, uppercasing the keys for comparison:

```
def teamKey(hockeyTeam):  
    return hockeyTeam.teamName.upper()
```

cmpFun

A function that defines the ordering relation on values in the list. The default value, **None**, means to use the usual Python **cmp()** function to compare values. You may supply your own function,

adhering to the usual convention for this function: when called to compare two elements **a** and **b**, **cmp(a, b)** should return:

- a negative value, if **a** should precede **b**;
- zero, if **a** and **b** are considered equal; or
- a positive value if **a** should follow **b**.

allowDups

If you use **allowDups=1**, you will be allowed to store multiple objects in the skip list even though they compare equal to each other. If, however, you set **allowDups=0**, the skip list object will raise an exception if you attempt to insert an element into the list that is equal to an existing element, according to the **cmpFun** function (or its default).

count

The purpose of this argument is to tune performance with very large skip lists. With the default value for **count** equal to a million, the skip list should provide access in logarithmic time for up to about a million elements. If you plan to store more than that, set **count** to an estimated number of elements.

If the number of elements in the skip list greatly exceeds the **count** value, it will still function correctly, but the performance may suffer.

2.3. Methods and attributes of the **SkipList** object

These attributes and methods are defined on a **SkipList** object:

.insert (c)

Inserts a new child element **c** into the skip list.

If you created the skip list with **allowDups=0** and the new element has the same key as an existing member of the list, this method will raise a **KeyError** exception.

If duplicates are allowed (the **allowDups=1** option to the constructor), and the new element's key duplicates one or more already in the list, the new child will be added in sequence after the other duplicates. This makes the ordering stable; see Section 2.1.3, "Stability" (p. 3).

.delete (k)

Deletes the first or only element whose key is equal to **k**, and returns the deleted child element (or **None** if there are no matching elements).

If duplicates are allowed and there are multiple elements equal to **e**, only the first one will be deleted.

.match (k)

Returns the first or only object whose key equals **k**. If there are no such objects, this method raises a **KeyError** exception.

.find (k)

Searches for the first element whose key is equal to or greater than **k**, and returns an iterator that iterates over all those elements.

.first()

If there are any elements in the skip list, this method returns the first element. If the skip list is empty, it raises **KeyError**.

.__len__(self)

This special method defines the meaning of the ordinary **len()** function when applied to a **SkipList**. It returns the number of child elements in the list.

• **__iter__(self)**

Returns an iterator object that can be used to iterate over the elements of the skip list. You can have multiple independent iterators pointing at the same skip list, and each will move independently of the others. This allows constructs like:

```
for i in s: ...
```

where **s** is a **SkipList** object. Such a loop will set **i** to each child value from **s** in turn.

• **__contains__(self, k)**

Defines the meaning of the Python “**in**” and “**not in**” operators in the usual way for container classes.

If **s** is a **SkipList**, the expression **k in s** returns true if and only if **k** is equal to at least one member of **s**. Similarly, the expression **k not in s** returns true iff **k** is not equal to any member of **s**.

• **__delitem__(self, k)**

Same as **.delete(k)**.

• **__getitem__(self, k)**

This method defines the usual meaning of **s[k]**, where **s** is a **SkipList**. It has the same semantics as **s.match[k]**.

This syntax is useful mainly for skip lists that don't allow duplicates. If you need to retrieve multiple members that all compare equal, use the **.find()** method.

• **nSearches**

Read-only; equal to the number of times the list has been searched. Each call to the **.match()**, **.find()**, or **.delete()** method is counted as one search.

• **nCompares**

Read-only; equal to the number of times two elements have been compared with the **cmpFun** function (or its default).

Statistically, the ratio of **nCompares** to **nSearches** should be proportional to the log of the number of elements. For analysis, see Pugh's original article.

3. Internals of SkipList

Before examining the actual code, let's discuss the data structure and some other implementation considerations.

3.1. The skip list data structure

To implement the functionality of this ordered set container, we could just keep all the objects in an ordinary linked list. However, searching a linked list has a time complexity that varies linearly with the number of elements. To be competitive with balanced tree approaches such as 2-3 trees, we need to be able to search in log time.

Pugh's idea was to set up a number of linked lists, numbered starting at 0, such that:

- Every object is in list 0, ordered from lowest to highest key.
- List *i* visits a subset of the objects in list (*i*-1), but still in order by key.

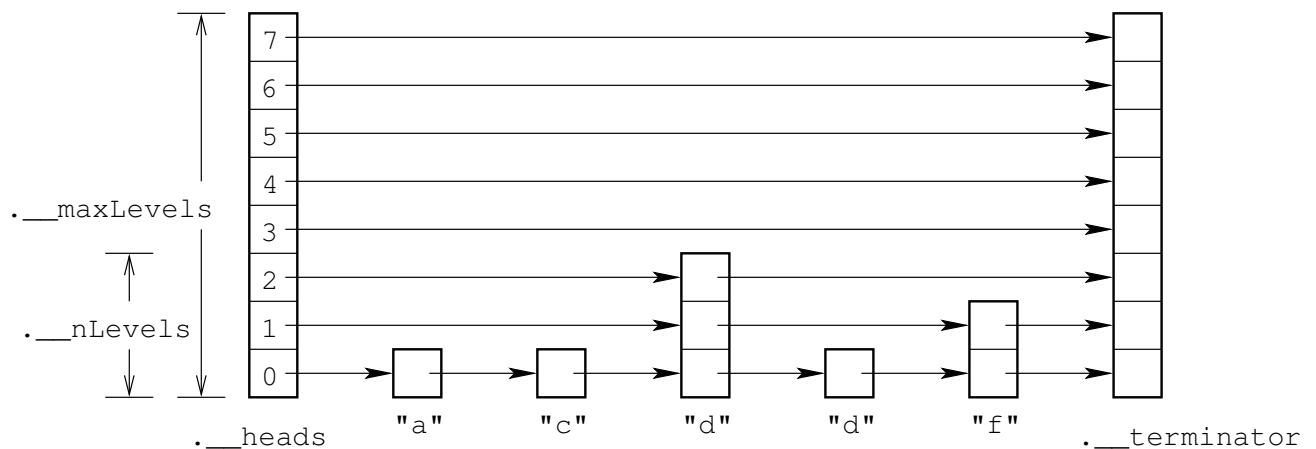
In practice, when each new element is inserted, it is always added to list 0, and it is also added to a random number of higher-numbered lists, where each higher level is less likely.

With this structure, the higher-numbered linked lists are more and more sparse. Therefore, the algorithm for searching a skip list is:

1. Search the highest-numbered list until you find either the desired item or one that is beyond the desired item.
2. If the desired item is not in the highest-numbered list, back up to the preceding item, move down one list, and search that list.
3. Repeat until either the desired item is found or it is not found in list 0.

So the search begins by “skipping” rapidly down the list using the highest-level, sparse linked list, then zeroes in on the desired item by moving down to denser and denser linked lists.

Here is an example of a small skip list containing some short character strings in lexical order. This list has a maximum of 8 levels:



This picture shows five strings, "a", "c", two copies of "d", and "f".

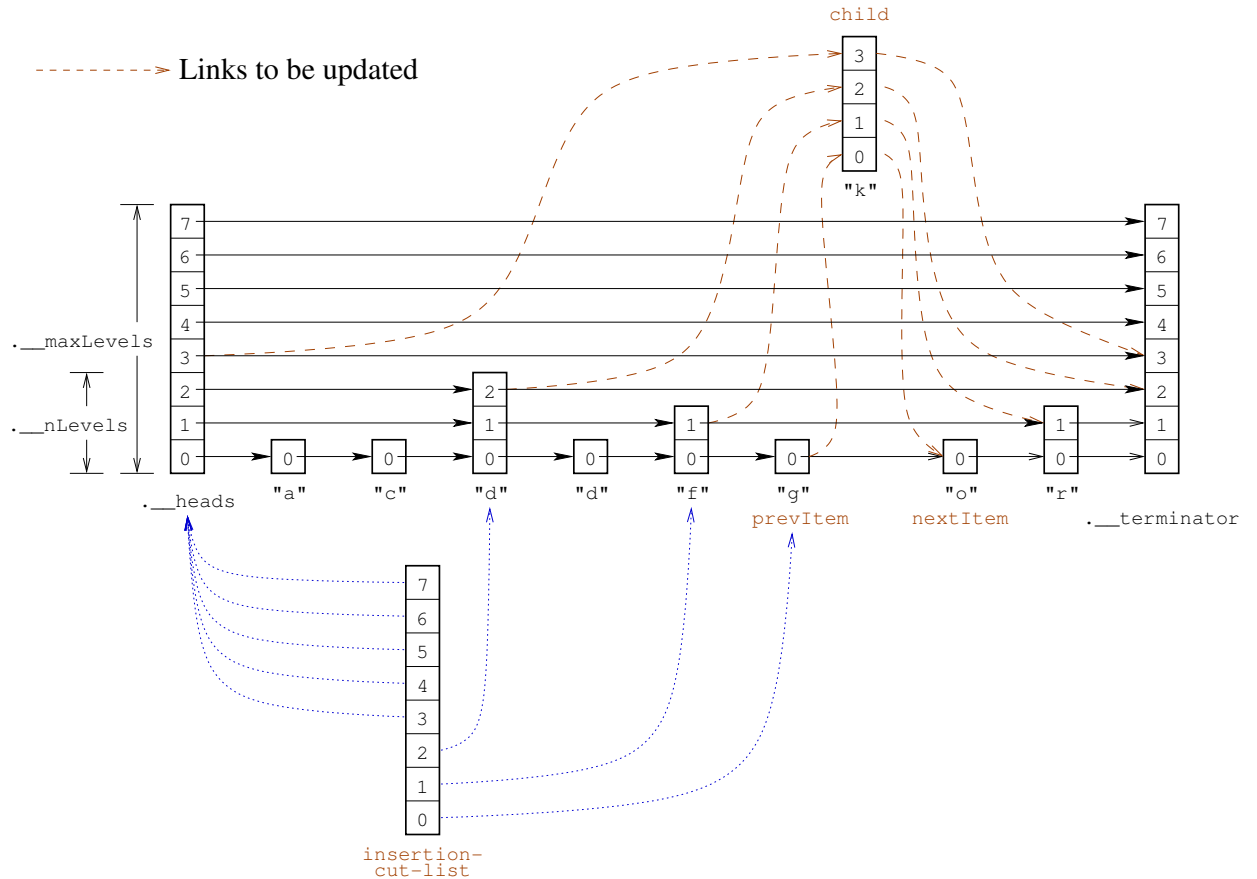
The field names refer to members of the **SkipList** object. The item labeled `.__heads` contains the heads of all the lists. Each list terminates with a pointer to the item labeled `.__terminator`. Field `.__maxLevels` is the maximum number of linked lists (8 in the figure), and `.__nLevels` is the number of lists that are currently in use.

3.2. Skip list algorithms

Now that we've seen the static structure of a skip list, let's move on to the basic operations: inserting a new child, finding a child, and deleting a child.

3.2.1. The insertion algorithm

Here is a figure showing the insertion of a new element into a skip list.



The dashed lines show links that need to be updated.

3.2.1.1. The insertion cut list

In the figure above, note the structure labeled “**insertion-cut-list**” is a list of pointers to the predecessors of the new element at each level. The first element of this list points to the item preceding the insertion in the level-0 list, which visits every element of the list. The second element in the insertion cut list points to the predecessor in the level-1 list, and so on.

This *insertion cut list* shows the position of a child element relative to each of the stacked linked lists. It tells us which links have to be repaired to include the newly inserted child.

We call it the insertion cut list to distinguish it from the *search cut list*; for the reasons behind this distinction, see Section 3.2.2, “The search algorithm” (p. 8).

3.2.1.2. Picking a level count

When we insert a new element, how many of the stacked linked lists should contain the new element? Pugh recommended using a random process based on a probability value p , where the number of levels used for each new element is given by this table:

1	$1-p$
2	$p(1-p)$
3	$p^2(1-p)$

$$\frac{\dots}{n} \quad | \quad \frac{\dots}{p^{n-1}(1-p)}$$

For example, for $p=0.25$, an element has a 3 in 4 probability of being linked into only level 0, a 3/16 probability of being in levels 0 and 1, a 3/64 probability of being in levels 0 through 2, and so forth.

The algorithm for deciding the number of levels is straightforward. For $p=0.25$, it's like throwing a 4-sided die. If it comes up 1, 2, or 3, use one level. If it comes up 4, roll again. If the second roll comes up 1, 2, or 3, use two levels. If it comes up 4, roll again, and so forth.

We will incorporate the “dirty hack” described in Pugh's article, limiting the growth in the total number of levels in use to one per insertion. This prevents the pathological case where a series of “bad rolls” might create many new levels at once for only a single insertion. Otherwise we might get silly behavior like a 6-level skip list to store ten elements.

3.2.2. The search algorithm

The process of searching for a given key value is nearly identical to the process of building the insert cut list (see Section 3.2.1.1, “The insertion cut list” (p. 7)).

The only difference between the *search cut list* and the insert cut list is in the case where we have elements with duplicate keys (that is, when the constructor was called with **allowDups=1**).

If we're inserting a new child whose key is a duplicate of a key already in the list, we want the new child to go *after* any duplicates. This is required for stability; see Section 2.1.3, “Stability” (p. 3).

However, if we're searching for an element (or deleting an element—see Section 3.2.3, “Deleting a child” (p. 8)), we want to position before any duplicate elements.

3.2.3. Deleting a child

Deletion of a child starts by finding the search cut list. For a discussion of the search cut list, see Section 3.2.2, “The search algorithm” (p. 8).

The search cut list contains all the predecessors whose links need to be updated when a child element is deleted. To delete a child, we work through the search cut list, repairing each predecessor link to point to the successor of the deleted child.

3.3. Iterators: toward a more Pythonic class

Starting with version 2.2, the Python language now supports a powerful new concept called the *iterator*.

An iterator is just an object that keeps track of a position within some sequence. The only requirement is that it have a **.next()** method that, when called:

- If there are any elements left in the sequence, the method returns the next element in the sequence, and the iterator also advances its own internal state to point to the following item, if there is one.
- When the sequence is exhausted, the method raises the **StopIteration** exception to signal that no more elements remain.

Some container classes return themselves as the result of the **.__iter__()** special method, so that the required **.next()** method uses some internal state stored inside the container class instance. However, the drawback to this approach is that you can't have two or more iterators pointing at different places in the sequence at the same time.

When we want to return a pointer at a position within a **SkipList** object, we can do so by pointing at a **_SkipItem** object. When we need to advance through the rest of the sequence, we can just follow the forward links in that object. This means there can exist two or more iterators walking through a skip list at the same time.

3.4. Classes in this module

When considering the class structure of the application, two classes are obvious:

- The user sees only the **SkipList** class, a classic container object.
- It makes sense to create a class to hold all the information for one node of the skip list data structure. This class is called **_SkipItem**; the single underbar in its name signifies that it is private to the module.

However, we need a third class to represent the iterators that are returned by the **.find()** method in the **SkipList** class, and by that class's special **.__iter__()** method.

So we define a third class, **_SkipListIterator**, that defines a **.next()** method. This object keeps track of its position in the list by pointing to a **_SkipItem**.

How does this iterator know when it has reached the end of the skip list? It can't compare the **_SkipItem** to **.__terminator**, which is private to the **SkipList** class. Even if we made that attribute public, a **_SkipItem** object doesn't contain a reference to its containing **SkipList**.

The solution is to stipulate that the **.__terminator** object's forward links all point to itself. That way, a **_SkipListIterator** instance can detect when it is pointing at the terminator.

There is one more complication with this technique. What happens if someone has a copy of an iterator that points to a **_SkipItem** that has been *deleted* from its containing **SkipList**?

In order that the **_SkipItem** object correctly detects this erroneous situation, it must know when it represents a deleted element.

Therefore, we stipulate that in a **_SkipItem** instance the level-0 forward link must be set to **None** when it is deleted. That way, the **.next()** method will know to signal an error and not attempt to chase its forward link.

4. The source code

The code for **pyskip.py** follows in literate programming style. The author is indebted to his colleague Dr. Allan M. Stavelly for literate programming tools.

The development uses the Cleanroom or “zero-defect” methodology described in Dr. Stavelly's book *Toward Zero-Defect Programming* (Addison-Wesley, 1999, ISBN 0-201-38595-3). The author's commenting conventions and other adaptations for Python practice are described online³.

4.1. Declarations

We'll start with a rudimentary documentation string for the module, pointing back to the online documentation:

³ <http://www.nmt.edu/~shipman/soft/clean/>

```

"""pyskip.py: A container class for ordered sets in Python.

    $Revision: 1.23 $   $Date: 2009/10/10 19:34:47 $

For documentation, see:
    http://www.nmt.edu/tcc/help/lang/python/examples/pyskip/
"""

```

Next we'll need to import one of the Python standard modules, the random number generator:

```
import random
```

4.2. Verification functions

The Cleanroom methodology relies on “verification functions” to describe notational shorthand. Here are all the verification functions used in this project, in alphabetical order.

All verification function names contain a hyphen, so you can distinguish them from program objects.

4.2.1. children-are-ordered

```

#=====
# Verification functions
#-----
# children-are-ordered ( x, y ) ==
#   if self.allowDups ->
#       cmp ( key-of ( x ), key-of ( y ) ) <= 0
#   else ->
#       cmp ( key-of ( x ), key-of ( y ) ) < 0
#--
# Similar to the keys-are-ordered predicate, but operates on
# child objects.

```

4.2.2. insertion-cut-list

```

#-----
# insertion-cut-list ( key ) ==
#   a list L of size self.__maxLevels, such that each element
#   L[i] equals insertion-point ( i, key )
#--
# An ``insertion cut list'' is a list of the _SkipItem objects
# that must be updated when a new elements is inserted. Element
# [i] of the cut list points to the element whose forward
# pointer has to be repaired in the (i)th list.

```

4.2.3. insertion-point

pyskip.py

```
#-----  
# insertion-point ( level, key ) ==  
#   the last element E in nth-list ( self.__heads, level ) such  
#   that insertion-precedes ( E, key ) is true  
#--  
#   This describes the predecessor _SkipItem whose forward link  
#   must be updated when a new item with key (key) is inserted.
```

4.2.4. insertion-point-after

pyskip.py

```
#-----  
# insertion-point-after ( level, key, searchItem ) ==  
#   the last element E in nth-list(searchItem,level) such that  
#   insertion-precedes(E, key) is true  
#--  
#   Just like insertion-point(), except that it starts at an  
#   arbitrary _SkipItem instead of self.__heads.
```

4.2.5. insertion-precedes

pyskip.py

```
#-----  
# insertion-precedes ( skipItem, key ) ==  
#   if skipItem is self.__terminator -> F  
#   else if skipItem is self.__heads -> T  
#   else ->  
#     keys-are-ordered ( key-of ( skipItem's child ), key )  
#--  
#   This predicate is true when (skipItem) should be before  
#   an item with key (key) in the level-0 list.
```

4.2.6. key-of

pyskip.py

```
#-----  
# key-of ( child ) ==  
#   if not self.keyFunction ->  
#     child  
#   else ->  
#     self.keyFunction ( child )
```

4.2.7. keys-are-ordered

pyskip.py

```
#-----  
# keys-are-ordered ( x, y ) ==  
#   if self.allowDups ->  
#     cmp ( x, y ) <= 0
```

```

# else ->
#     cmp ( x, y ) < 0
#--
# This is the ordering relation used in the key domain.
# - If we don't allow duplicates, then each key must be
#   strictly less than its successor.
# - If we allow duplicates, then two successive keys can
#   be equal.

```

4.2.8. nth-list

pyskip.py

```

#-----
# nth-list ( root, n ) ==
#   the linked list of _SkipItem objects rooted at (root) and
#   using _SkipItem.links[n] as the next-item method
#--
# This define is used to describe positions in the linked list
# of objects at one level of the structure. In particular,
#   nth-list ( self.__heads, n )
# describes the entire skip list at level (n).

```

4.2.9. search-cut-list

pyskip.py

```

#-----
# search-cut-list ( key ) ==
#   a list L of size self.__maxLevels such that
#   L[i] := search-point ( i, key )
#--
# Like insert-cut-list(), but used for .delete() and .find()
# operations.

```

4.2.10. search-point

pyskip.py

```

#-----
# search-point ( level, key ) ==
#   the last _SkipItem E in nth-list(self.__heads, level) such that
#   search-precedes(E, key) is true
#--
# The predecessor whose forward link must be updated when
# the item with key (key) is deleted. Also used in
# .find().

```

4.2.11. search-point-after

pyskip.py

```

#-----
# search-point-after ( level, key, searchItem ) ==
#   the last element E in nth-list(searchItem, level) such that
#   search-precedes(E, key) is true

```

```
#--
# Like search-point except that the search starts at some
# given item rather than at self.__heads.
```

4.2.12. search-precedes

pyskip.py

```
#-----
# search-precedes ( skipItem, key ) ==
#   if skip-compare ( skipitem, key ) < 0  ->  true
#   else -> false
#--
# A predicate, true when the child in skipItem is before the
# key (key).
```

4.2.13. skip-compare

pyskip.py

```
#-----
# skip-compare ( skipItem, key ) ==
#   if skipItem is self.__terminator -> 1
#   else -> cmp ( key-of ( skipItem's child ), key )
#--
# Like cmp(), but we want to avoid trying to extract a key
# from self.__terminator, because it doesn't have one. If
# skipItem is the terminator, we return 1 because the terminator
# goes after all other elements.
#-----
```

4.3. The `_SkipItem` internal class

Before we discuss the main **SkipList** class, we need a helper object to hold the structural bookkeeping information associated with one element of the skip list.

This helper object doesn't have much to it:

- Its **.child** attribute is a pointer to the child object at that position in the skip list.
- Its **.links** attribute is a list containing the one or more forward links connecting this element of the skip list to the following element (or pointing to the **__terminator** if this element is the last one).

Here, then, is the **_SkipItem** class:

pyskip.py

```
# - - - - - c l a s s   _ S k i p I t e m   - - - - -
class _SkipItem:
    """Represents one child element of a SkipList.

    Exports:
        _SkipItem ( child, nLevels ):
            [ (child is a child object) and
              (nLevels is an integer >= 1) ->
              return a new _SkipItem with that child object and
```

```

        (nLevels) forward links, each set to None ]
    .child:      [ as passed to constructor, read-only ]
    .links:
        [ if self is an active SkipList element ->
          a list of nLevels elements, read-write, containing
          pointers to a _SkipItem instance
          else ->
            a list of at least 1 whose first element is None ]
    """
# - - - _ S k i p I t e m . _ _ i n i t _ _ - - -

def __init__( self, child, nLevels ):
    """Constructor for _SkipItem"""
    self.child = child
    self.links = [None]*nLevels

```

The final line uses the Python convention that an expression of the form “ $[x]*n$ ”, where n is an integer, gives you a list containing n copies of x .

4.4. The `_SkipListIterator` class

The `_SkipListIterator` class is used to represent an iterator that walks down a skip list. It has one piece of internal state: a pointer to the `_SkipItem` instance that marks the next item, or to the terminator instance if there are no more items.

It has only three methods, a constructor and the `__iter__()` and `next()` methods required for Python iterators. The `__iter__()` method just returns the instance; without this method, you can't use this object in constructs like “`for k in ...`”.

pyskip.py

```

# - - - - - c l a s s _ S k i p L i s t I t e r a t o r - - - - -

class _SkipListIterator:
    """Represents an active iterator over a SkipList object.

    Exports:
    _SkipListIterator ( skipItem ):
        [ skipItem is a _SkipItem ->
          return a new iterator whose next item is skipItem,
          or which is at end of list if skipItem's forward
          link points to itself ]
    .skipItem:
        [ if self is exhausted ->
          a terminator _SkipItem
          else ->
            a _SkipItem containing the value that will be returned
            next time ]
    __iter__(self):    [ returns self ]
    .next():
        [ if self.skipItem's level-0 link is None ->
          raise ValueError
          else if self.skipItem.links[0] == self.skipItem ->

```

```

        raise StopIteration
    else ->
        self.skipItem := self.skipItem.links[0]
        return self.skipItem.child ] # *Before* advancing!
    """

# - - - _ S k i p L i s t I t e r a t o r . _ _ i n i t _ _ - - -

def __init__ ( self, skipItem ):
    """Constructor for _SkipListIterator"""
    self.skipItem = skipItem

# - - - _ S k i p L i s t I t e r a t o r . n e x t - - -

def next ( self ):
    """Return the next child item and advance"""
    if self.skipItem.links[0] is None:
        raise ValueError, "Iterator points to a deleted item."
    elif self.skipItem.links[0] is self.skipItem:
        raise StopIteration
    else:
        result = self.skipItem.child
        self.skipItem = self.skipItem.links[0]
        return result

```

In order for a **_SkipListIterator** object to qualify as an iterator, and be useable in contexts such as “for k in S”, the object must have a special **__iter__()** method that returns the iterator.

pyskip.py

```

# - - - _ S k i p L i s t I t e r a t o r . _ _ i t e r _ _ - - -

def __iter__ ( self ):
    """Returns the iterator itself."""
    return self

```

4.5. The SkipList class

Now we turn to the actual **SkipList** class. First, the external interface with intended functions:

pyskip.py

```

# - - - - - c l a s s   S k i p L i s t   - - - - -

class SkipList:
    """Container class for ordered sets.

    Exports:
    SkipList ( keyFun=None, cmpFun=None, allowDups=0,
              count=1000000 ):
        [ (keyFun is a function that returns the key for a
          given child object) and
          (cmpFun is a function that compares two keys, using
          the usual Python convention for the cmp() function,
          or None use the built-in cmp() function) and
          (allowDups is 0 to refuse duplicate entries or 1 to

```

```

        allow them) and
        (count is a worst-case maximum element count) ->
        return a new, empty SkipList instance with those
        properties ]
.insert(c):
    [ c is a child object ->
      if ((self contains an object whose key equals the key
          of e) and (self.allowDups is false)) ->
          raise KeyError
      else ->
          self := self with e added as a new child object ]
.delete(k):
    [ k is in the key domain ->
      if self contains any child objects with keys equal to k ->
          self := self with the first-inserted such child object
          deleted
      return that child object
      else ->
          return None ]
.match(k):
    [ k is in the key domain ->
      if self contains any child objects whose keys equal k ->
          return the first such child object
      else -> raise KeyError ]
.find(k):
    [ k is in the key domain ->
      if self contains any child objects whose keys are >= k ->
          return an iterator that will iterate over all
          child objects whose keys are >= k, in order ]
.__len__(self):
    [ return the number of child objects in self ]
.__iter__(self):
    [ return an iterator that will iterate over all the
      child objects in self in order ]
.__contains__(self, k):
    [ if self contains any child objects whose keys equal k ->
      return 1
      else -> return 0 ]
.__delitem__(self, k):      [ same as self.delete(k) ]
.__getitem__(self, k):     [ same as self.match(k) ]
.nSearches: [INVARIANT: Number of searches performed ]
.nCompares: [INVARIANT: Number of child pairs compared ]
"""

```

Next, we declare a manifest constant for the Pugh's p :

pyskip.py

```

#--
# Manifest constants
#--
    NEW_LEVEL_PROBABILITY = 0.25 # The `p' of Pugh's article

```

Next, we need to declare the class's internal state and invariants.

```

#=====
# State and invariants
#-----
# .keyFun:      [ as passed to constructor, read-only ]
# .cmpFun:      [ as passed to constructor, read-only ]
# .allowDups:   [ as passed to constructor, read-only ]
# .__maxLevels:
#   [ 1 + ceiling ( log (base 4) of count argument to constructor ) ]
# .__nLevels:
#   [ number of levels currently in use ]
#   INVARIANT: 1 <= self.__nLevels <= self.__maxLevels ]
# .__heads:
#   [ list of head pointers to each level ]
#   INVARIANT: self.__heads.links[0] is the head of a linked list
#   of _SkipItem objects, terminated by a link to self.__terminator,
#   and whose members contain all the child objects currently in
#   self, such that for any two adjacent objects (Xi, Xj),
#   children-are-ordered ( Xi, Xj ) is true.
#
#   INVARIANT: For i in [0,self.__maxLevels), the list rooted in
#   heads.links[i] is a linked list of _SkipItem objects,
#   terminated by a link to self.terminator, and containing
#   a subset of the linked list in heads.links[i-1] in the same order.
# .__terminator: [ terminator for all linked lists ]
#   INVARIANT: self.terminator is a _SkipItem object with
#   self.__maxLevels forward links all pointing to itself
# .__nItems:     [ INVARIANT: number of child items in self ]
#--

```

4.6. SkipList.__init__()

Here's the **SkipList** constructor. First we make local copies of all the constructor arguments except **count**, and set up the invariants for several internal state values:

```

# - - - S k i p L i s t . _ _ i n i t _ _ - - -

def __init__ ( self, keyFun=None, cmpFun=None, allowDups=0,
               count=1000000 ):
    """Constructor for SkipList"""

    #-- 1 --
    self.keyFun      = keyFun
    self.cmpFun      = cmpFun
    self.allowDups   = allowDups
    self.nSearches  = 0
    self.nCompares  = 0
    self.__nLevels  = 1
    self.__nItems   = 0

```

Then we compute **self.__maxLevels** based on the **count** argument. Based on our p value of 0.25, we'd expect to need two levels when we have about four elements, three levels when we have sixteen,

and so forth. To be generous, we'll use one plus the ceiling of the log (base 4) of **count**. See Section 4.7, "**SkipList.__estimateLevels()**" (p. 18).

pyskip.py

```
#-- 2 --
# [ self.__maxLevels := (number of bits required to
#     represent count, rounded to the next even number)/2+1 ]
self.__maxLevels = self.__estimateLevels ( count )
```

Next we have to set up the **.__heads** and **.__terminator** elements. They are both special instances of the **_SkipItem** class; in the former, all links point to the latter, and in the latter, all links point to itself.

pyskip.py

```
#-- 3 --
# [ self.__terminator := a new _SkipItem object whose
#     links all point to itself
#   self.__heads := a new _SkipItem object whose links
#     all point to that new terminator _SkipItem object ]
self.__terminator = _SkipItem ( None, self.__maxLevels )
self.__heads      = _SkipItem ( None, self.__maxLevels )
for i in xrange ( self.__maxLevels ):
    self.__terminator.links[i] = self.__terminator
    self.__heads.links[i]     = self.__terminator
```

4.7. SkipList.__estimateLevels()

Here's the **.__estimateLevels()** function used in the constructor:

pyskip.py

```
# - - - S k i p L i s t . _ _ e s t i m a t e L e v e l s - - -
def __estimateLevels ( self, n ):
    """Estimate how many levels of list we need for n child objects.

    [ n is a positive integer ->
      return ceiling(log base 4(n)) + 1 ]
    """

    #-- 1 --
    result = 1

    #-- 2 --
    # [ result += k, where k is the the number of bits required
    #     to represent n, rounded to the next higher even number ]
    while n > 0:
        result += 1
        n >>= 2

    #-- 3 --
    return result
```

In the **while** loop, we count the number of times we have to shift **n** to the right two bits before it goes to zero, effectively computing the ceiling of the log base 4 of **n**. This gives us a result of 1 for **n=0**, 2 for **n** in the range [1,4), 3 for **n** in the range [4,16), 4 for **n** in the range [16,256), and so on.

4.8. SkipList.insert()

The insertion of a new child element is depicted above; see Section 3.1, “The skip list data structure” (p. 5).

pyskip.py

```
# - - - S k i p L i s t . i n s e r t - - -  
  
def insert ( self, child ) :  
    """Insert a new child element into the skip list."""
```

To link in a new child element, once we have decided how many levels of links it will use, we must insert it into each of those linked lists. So we define the idea of an *insertion cut list*: a list of all the predecessors whose forward links must be updated.

The main complication here is that we must implement stability, that is, objects with equal keys must be ordered by insertion time. For example, if we have five objects all with a key value of 5, the first one must be the first inserted, and the last one must be the last inserted.

So, when computing the insertion cut list, we must find the point in the child sequence after the last child with the same key. Refer to Section 4.2, “Verification functions” (p. 10) for the verification functions used here, and also see Section 4.9, “SkipList.__keyOf()” (p. 20).

pyskip.py

```
#-- 1 --  
# [ key := key-of ( child ) ]  
key = self.__keyOf ( child )
```

The next step is to build the insertion cut list. If there any elements with the same key as the new element, the insertion cut list depends on whether duplicates are allowed or not. We call the point of insertion the *cut point*, and the cut list is the list of links that cross the cut point.

If duplicates aren't allowed, the cut point is positioned *before* any duplicate elements, so that when we check for duplicates, we check the new key against the *successor* of the cut point.

But if duplicates are allowed, due to the stability constraint, the cut point should be *after* any duplicates. See Section 4.8, “SkipList.insert()” (p. 19) for a discussion of stability.

pyskip.py

```
#-- 2 --  
# [ cutList := insertion-cut-list ( key ) ]  
cutList = self.__insertCutList ( key )  
  
#-- 3 -  
# [ prevItem := first item from cutList  
#   nextItem := successor at level 0 of first item from  
#             cutList ]  
prevItem = cutList[0]  
nextItem = prevItem.links[0]
```

Now that we have **nextItem** pointing just after the cut point, we can check for duplicates when they're not allowed. See Section 4.13, “SkipList.__compareItemKey()” (p. 23).

pyskip.py

```
#-- 4 --  
# [ if (not self.allowDups) and  
#   (nextItem is not self.terminator) and  
#   (cmp(key-of(nextItem.child, key)) == 0 ) ->  
#     raise ValueError
```

```

# else -> I ]
if ( ( not self.allowDups ) and
      ( nextItem is not self.__terminator ) and
      ( self.__compareItemKey ( nextItem, key ) == 0 ) ):
    raise ValueError

```

Having eliminated the only error case, now we can proceed with the actual insertion of the new element. See Section 4.14, “**SkipList.__insertItem()**” (p. 24).

pyskip.py

```

#-- 5 --
# [ if cutList is insertion-cut-list ( key ) ->
#   self := self with a new _SkipItem containing (child)
#         inserted after the items pointed at by the
#         the first n elements of cutList, where
#         n is in [1,self.__maxLevels] ]
self.__insertItem ( child, cutList )

```

The next step maintains the invariant on `.__nItems`, that is, we keep track of the number of items in the list.

pyskip.py

```

#-- 6 --
self.__nItems = self.__nItems + 1

```

4.9. SkipList.__keyOf()

This function finds the child's key. If there is no key function, use the child itself as the key.

pyskip.py

```

# - - - S k i p L i s t . _ _ k e y 0 f - - -

def __keyOf ( self, child ):
    """Return the child's key.

    [ child is in the child domain of self ->
      return key-of ( child ) ]
    """
    if self.keyFun:
        return self.keyFun ( child )
    else:
        return child

```

4.10. SkipList.__insertCutList()

This routine builds a list of the predecessors of the cut point for insertion. As Pugh's paper describes, we start searching at the highest-numbered list and proceed down to the level-0 list.

pyskip.py

```

# - - - S k i p L i s t . _ _ i n s e r t C u t L i s t - - -

def __insertCutList ( self, key ):
    """Form the insertion cut list.

    [ key is in self's key domain ->

```

```

        return insertion-cut-list(key) ]
    """

```

The first step is to build a list of pointers to **self.__heads**, one for each possible level.

pyskip.py

```

#-- 1 --
# [ result      := a list of size self.__maxLevels such that
#               each element is self.__heads
#   searchItem := self.__heads ]
result      = [self.__heads] * self.__maxLevels
searchItem  = self.__heads

```

Next, we search all the levels currently in use from the top down. We find the cut point at each level, then for the next level we start at the predecessor of that cut point. The routine to find each cut point is Section 4.11, "**SkipList.__insertPoint()**" (p. 21).

pyskip.py

```

#-- 2 --
# [ if insertion-precedes ( searchItem, key ) ->
#   result      := result modified so that for I in
#                 [0,self.nLevels), result[I] is
#                 insertion-point(I, key)
#   searchItem := <anything> ]
for level in xrange ( self.__nLevels - 1, -1, -1 ):
    #-- 2 body --
    # [ if insertion-precedes(searchItem, key) ->
    #   searchItem := insertion-point-after(level,
    #                                       key, searchItem)
    #   result[level] := <same as previous line> ]
    searchItem = self.__insertPoint ( searchItem, level, key
    )
    result[level] = searchItem

```

Finally, we maintain the invariant on **.nSearches**, since every call to this routine counts as a search.

pyskip.py

```

#-- 3 --
self.nSearches = self.nSearches + 1      # Count as a search
return result

```

4.11. SkipList.__insertPoint()

This routine searches down one level's linked list until it reaches the point where a new key is to be inserted.

pyskip.py

```

# - - -   S k i p L i s t . _ _ i n s e r t P o i n t   - - -
def __insertPoint ( self, searchItem, level, key ):
    """Find the insertion point at a given level.

    [ insertion-precedes(searchItem, key) ->
      return insertion-point-after ( level, key, searchItem) ]
    """

```

First we find the predecessor and successor of the starting point.

pyskip.py

```
#-- 1 --
# [ prevItem := searchItem
#   nextItem := successor of searchItem in (level)th list ]
prevItem = searchItem
nextItem = searchItem.links[level]
```

Next, we move down the n^{th} -level list until we reach the point where the insertion will go.

pyskip.py

```
#-- 2 --
# [ if nextItem is not self.__heads ->
#   if not insertion-precedes(nextItem, key) -> I
#   else ->
#     prevItem := last item E at or after nextItem
#               in the (level)th list such that
#               insertion-precedes(E, key) is true
#     nextItem := <anything> ]
while self.__insertionPrecedes ( nextItem, key ):
    #-- 2 body --
    # [ prevItem := nextItem
    #   nextItem := the successor of nextItem in the
    #               (level)th list ]
    prevItem = nextItem
    nextItem = nextItem.links[level]
```

Finally, we return the predecessor to that location.

pyskip.py

```
#-- 3 --
return prevItem
```

4.12. SkipList.__insertionPrecedes()

This routine tests whether a given **SkipItem** is before a given key, assuming that we're performing an insertion of a child with that key value.

pyskip.py

```
# - - - S k i p L i s t . _ _ i n s e r t i o n P r e c e d e s - - -

def __insertionPrecedes ( self, skipItem, key ):
    """Does this _SkipItem precede this key for insertion?

    [ skipItem is not self.__heads ->
      return keys-are-ordered ( key-of ( skipItem's child ),
                               key ) ]
    """
```

The terminator never precedes anything:

pyskip.py

```
#-- 1 --
if skipItem is self.__terminator:
    return 0
```

Next, we compare the key of the `_SkipItem` to the new key. See Section 4.13, “`SkipList.__compareItemKey()`” (p. 23).

pyskip.py

```
#-- 2 --
# [ comparison := cmp ( key-of ( skipItem's child ), key ) ]
comparison = self.__compareItemKey ( skipItem, key )
```

Whether we want to place the new key before or after duplicates depends on whether the skip list allows duplicates.

pyskip.py

```
#-- 3 --
#
# Note: if duplicates are disallowed, and there is an item that
# duplicates (target), we want to point before the duplicate
# item so that the .insert() method will see it and fail.
# If duplicates are allowed, though, we want to point after
# all matching items so that the list order will reflect
# the insertion order.
#
# [ if self.allowDups ->
#     if comparison <= 0 -> return 1
#     else -> return 0
# else ->
#     if comparison < 0 -> return 1
#     else -> return 0 ]
if self.allowDups:
    return ( comparison <= 0 )
else:
    return ( comparison < 0 )
```

4.13. `SkipList.__compareItemKey()`

This small routine compares two keys, one from a `_SkipItem` and one in the key domain.

pyskip.py

```
# - - - S k i p L i s t . _ _ c o m p a r e I t e m K e y - - -
def __compareItemKey ( self, skipItem, keyB ):
    """Compare the key from a _SkipItem to a key in the key domain.

    [ return cmp ( key-of ( skipItem's child ), keyB ) ]
    """
```

First we get the key from `skipItem`; see Section 4.9, “`SkipList.__keyOf()`” (p. 20).

pyskip.py

```
#-- 1 --
# [ keyA := key-of ( skipItem's child ) ]
keyA = self.__keyOf ( skipItem.child )
```

We increment `self.nCompares`, the count of the number of comparison operations. Then we use Python's built-in `cmp()` function to do the comparison, returning its result as our result.

```
#-- 2 --
self.nCompares = self.nCompares + 1
return cmp ( keyA, keyB )
```

4.14. SkipList.__insertItem()

The responsibility of this method is to build a new **_SkipItem** to hold the newly inserted child, and then link it into one or more of the linked lists.

```
# - - - S k i p L i s t . _ _ i n s e r t I t e m - - -

def __insertItem ( self, child, cutList ):
    """
    [ cutList is insertion-cut-list(key-of(child)) ->
      self := self with a new _SkipItem, with child=(child),
            inserted after the items pointed at by the
            first n levels of cutList, where n is in the
            range [1,self.__maxLevels] ]
    """
```

First, using the random number generator, we decide into how many levels we should link the new item. If number of levels picked exceeds the current number of levels, we must adjust the value of **self.__nLevels** to maintain the invariant that this variable tracks the maximum number of levels actually in use so far. See Section 4.15, “**SkipList.__pickLevel()**” (p. 25).

```
#-- 1 --
# [ levels          := a random integer in [1,self.__maxLevels]
#   self.__nLevels := max ( self.__nLevels,
#                           that random integer ) ]
levels = self.__pickLevel ( )
```

Then we construct the actual **_SkipItem** containing the new child. See Section 4.3, “The **_SkipItem** internal class” (p. 13).

```
#-- 2 --
# [ newItem := a new _SkipItem with child=(child) and
#   (levels) forward links all set to None ]
newItem = _SkipItem ( child, levels )
```

Finally, the **._insertRelink()** method takes care of linking the item into the correct number of linked lists; see Section 4.16, “**SkipList.__insertRelink()**” (p. 25).

```
#-- 3 --
# [ (cutList is insertion-cut-list(key-of(child))) and
#   (newItem is a _SkipItem with at least (levels) links) ->
#   self := self with newItem linked into the first (levels)
#         lists, just after the element pointed at by the
#         corresponding element of cutList ]
self.__insertRelink ( levels, cutList, newItem )
```

4.15. SkipList.__pickLevel()

This method implements the algorithm for probabilistically deciding how many levels to use for linking in a new skip list entry. For a discussion of this algorithm, see Section 3.1, “The skip list data structure” (p. 5).

pyskip.py

```
# - - - S k i p L i s t . _ _ p i c k L e v e l - - -  
  
def __pickLevel ( self ):  
    """Into how many levels should an insertion be linked?  
  
    [ self.__nLevels := max ( self.__nLevels,  
        a randomly chosen integer in [1,self.__maxLevels] )  
    return that same integer ]  
    """
```

Here is what Pugh calls the “dirty hack:” we will never add more than one level to a skip list per insertion. For the details, see Section 3.1, “The skip list data structure” (p. 5).

pyskip.py

```
#-- 1 --  
result = 1  
maxNewLevel = min ( self.__nLevels + 1, self.__maxLevels )
```

Roll the dice, figuratively. The **random.random()** function returns a real in the interval [0,1), and **NEW_LEVEL_PROBABILITY** is what Pugh calls p . The process is limited by the previously computed **maxNewLevel**. For details on the **random** module, see the Python library reference⁴ under “Miscellaneous Services.”

pyskip.py

```
#-- 2 --  
# [ maxNewLevel >= result ->  
#   result := a randomly chosen integer in the range  
#           [result,maxNewLevel] ]  
while ( ( random.random() <= self.NEW_LEVEL_PROBABILITY ) and  
        ( result < maxNewLevel ) ):  
    result = result + 1
```

Maintain the invariant on **self.__nLevels** (a running max of the number of levels ever used in the skip list), and return the generated result.

pyskip.py

```
#-- 3 --  
self.__nLevels = max ( self.__nLevels, result )  
  
#-- 4 --  
return result
```

4.16. SkipList.__insertReLink()

This routine takes care of repairing all the linked lists that must contain the newly created **_SkipItem**. In general, insertion into a linked lists has this form, where **P** is the predecessor, **S** is the successor, and **N** is the new block:

⁴ <http://docs.python.org/lib/lib.html>

```
N.link = S
P.link = N
```

So all we need to do is execute this relinking operation once for each level that contains the new **_SkipItem**. See also the diagram of this relinking in Section 3.1, “The skip list data structure” (p. 5).

pyskip.py

```
# - - - S k i p L i s t . _ _ i n s e r t R e l i n k - - -
def __insertRelink ( self, levels, cutList, newItem ) :
    """Insert the new _SkipItem into all its linked lists.

    [ (cutList is insertion-cut-list(key-of(child))) and
      (newItem is a _SkipItem with child=(child) and
      at least (levels) links) ->
      self := self with newItem linked into the first (levels)
              lists, just after the element pointed at by the
              corresponding element of cutList ]
    """

    #-- 1 --
    for i in xrange(levels):

        #-- 1 loop --
        # [ i is an integer in [0,levels) ->
        #   newItem.links[i] := cutList[i].links[i]
        #   cutList[i].links[i] := newItem ]

        #-- 1.1 --
        # [ i is an integer in [0,levels) ->
        #   prevItem := the item pointed at by cutList[i]
        #   succItem := the (i)th link from the item
        #               pointed at by cutList[i] ]
        prevItem = cutList[i]
        succItem = prevItem.links[i]

        #-- 1.2 --
        # [ i is an integer in [0,levels) ->
        #   newItem := newItem with its (i)th link
        #               pointing to succItem
        #   prevItem := prevItem with its (i)th link
        #               pointing to newItem ]
        newItem.links[i] = succItem
        prevItem.links[i] = newItem
```

4.17. SkipList.delete()

Deletion, like insertion, also uses the idea of a *cut list*, that is, a list of all the predecessors whose links must be repaired to leave out the deleted item.

However, deletion uses a slightly different concept of the cut list, the *search cut list*. The only difference comes when the skip list allows duplicate keys. When we're inserting a child with a duplicate key, we want it to go after the other children with that key, to guarantee stability.

For deletion and searching, however, we want the cut list to point before the first of any values whose keys are equal to the key we're deleting or searching for.

pyskip.py

```
# - - - S k i p L i s t . d e l e t e - - -  
  
def delete ( self, key ):  
    """Delete the first or only child with a given key value.  
    """
```

First we find the search cut list for the given key. We also set local variables **prevItem** and **nextItem** to point to the predecessor and successor of the cut point at level 0. See Section 4.18, "**SkipList.__searchCutList()**" (p. 28).

pyskip.py

```
#-- 1 --  
# [ cutList := search-cut-list ( key )  
#   prevItem := first element of search-cut-list ( key )  
#   nextItem := successor to first element of  
#             search-cut-list ( key ) ]  
cutList = self.__searchCutList ( key )  
prevItem = cutList[0]  
nextItem = prevItem.links[0]
```

If the cut point is before the terminator, or the element after the cut point does not have the key value we're looking for, we're done, and we return **None** to signify failure to delete anything. See Section 4.13, "**SkipList.__compareItemKey()**" (p. 23).

pyskip.py

```
#-- 2 --  
# [ if (nextItem is self.__terminator) or  
#   ( cmp ( key-of ( nextItem's child ), key ) ) > 0 ) ->  
#   return None  
#   else -> I ]  
if ( ( nextItem is self.__terminator ) or  
    ( self.__compareItemKey ( nextItem, key ) > 0 ) ):  
    return None
```

We've found the item to be deleted, and **nextItem** points to it. Relink all the lists that point at that item so that the predecessor at that level in the cut list points at **nextItem**'s successor at that level.

pyskip.py

```
#-- 3 --  
# [ self := self modified so that for all I in  
#           [0,self.__nLevels), if the skipItem pointed to by  
#           cutList[I] has an (I)th link that points to  
#           nextItem, make that link point where nextItem's  
#           (I)th link points ]  
for i in xrange(self.__nLevels):  
    #-- 3 body --  
    # [ if the _SkipItem pointed to by cutList[i] has an  
    #   (i)th link that points to nextItem ->  
    #   that link is made to point where nextItem's (i)th  
    #   link points ]  
    prevItem = cutList[i]  
    if prevItem.links[i] is nextItem:  
        prevItem.links[i] = nextItem.links[i]
```

Finally, we must adjust `.__nItems` to reflect the deletion of one child, and return the deleted child value.

We must also set the level-0 link in the deleted `_SkipItem` to `None`, for reasons discussed in Section 3.4, "Classes in this module" (p. 9).

pyskip.py

```
#-- 4 --
self.__nItems      = self.__nItems - 1
nextItem.links[0] = None
return nextItem.child
```

4.18. SkipList.__searchCutList()

This method returns the "search cut list", a list of the predecessors of a given key at each level of the skip list.

pyskip.py

```
# - - - S k i p L i s t . _ _ s e a r c h C u t L i s t - - -

def __searchCutList ( self, key ):
    """Find predecessors of the item with a given key.

    [ key is in self's key domain ->
      return search-cut-list ( key ) ]
    """
```

We start by building a list called **result** containing pointers to the head element, and also point **searchItem** at the head element.

pyskip.py

```
#-- 1 --
# [ result      := a list of size self.__maxLevels such that
#             each element is self.__heads
#   searchItem := self.__heads ]
result      = [self.__heads] * self.__maxLevels
searchItem  = self.__heads
```

This loop is similar to the one in Section 4.10, "**SkipList.__insertCutList()**" (p. 20). It starts at the highest level currently in use and searches forward to find the predecessor at that level. Then it backs up and goes down a level until it reaches level 0. See Section 4.19, "**SkipList.__searchPoint()**" (p. 29).

pyskip.py

```
#-- 2 --
# [ if search-precedes ( searchItem, key ) ->
#   result      := result modified so that for I in
#                 [0,self.__nLevels),
#                 result[I] := search-point(I, key)
#   searchItem := <anything> ]
for i in xrange ( self.__nLevels-1, -1, -1 ):
    #-- 2 body --
    # [ if search-precedes ( searchItem, key ) ->
    #   result[i] := search-point-after(i, key, searchItem)
    #   searchItem := <same as previous line> ]
```

```

        searchItem = self.__searchPoint ( searchItem, i, key )
        result[i] = searchItem

```

Finally, we increment the total count of searches and return the resulting cut-list.

pyskip.py

```

#-- 3 --
self.nSearches = self.nSearches + 1
return result

```

4.19. SkipList.__searchPoint()

This routine locates the predecessor of the first item after a given point that is before the given key value.

pyskip.py

```

# - - -   S k i p L i s t . _ _ s e a r c h P o i n t   - - -

def __searchPoint ( self, searchItem, level, key ):
    """Search one level of the skip list for a given key.

        [ ( level is in [0,self.__maxLevels ) ) and
          ( search-precedes ( searchItem, key ) ) ->
            return search-point-after ( level, key,
            searchItem ) ]

    """

```

First we set the local variables **prevItem** and **nextItem** to the item where we start, and its successor, respectively.

pyskip.py

```

#-- 1 --
# [ prevItem := searchItem
#   nextItem := successor of searchItem in (level)th list ]
prevItem = searchItem
nextItem = searchItem.links[level]

```

Then we move both these variables down the linked list at the given level so long as **nextItem** still precedes the key we're searching for. See Section 4.20, "**SkipList.__searchPrecedes()**" (p. 30).

pyskip.py

```

#-- 2 --
# [ if nextItem is not self.__heads ->
#   if search-precedes ( nextItem, key ) ->
#     prevItem := last item E in nth-list(nextItem, level)
#               such that search-precedes(E, key) is true
#   nextItem := <anything>
#   else -> I ]
while self.__searchPrecedes ( nextItem, key ):
    prevItem = nextItem
    nextItem = nextItem.links[level]

```

Finally, we return the predecessor item.

pyskip.py

```

#-- 3 --
return prevItem

```

4.20. SkipList.__searchPrecedes()

This is a predicate used to test whether a given `__SkipItem` precedes the item with a given key.

pyskip.py

```
# - - - S k i p L i s t . _ _ s e a r c h P r e c e d e s - - -  
  
def __searchPrecedes ( self, skipItem, key ):  
    """Does this item precede the item with a given key?  
  
    [ ( skipItem is a __SkipItem ) and  
      ( key is in self's key domain) ->  
        if search-precedes ( skipItem, key ) ->  
            return 1  
        else ->  
            return 0 ]  
  
    """
```

Eliminate the case where `skipItem` is the terminator, because that item never precedes anything. Then use the `.__compareItemKey()` method to do the actual comparison.

pyskip.py

```
#-- 1 --  
if skipItem is self.__terminator:  
    return 0  
  
#-- 2 --  
# [ if cmp ( key-of(skipItem's child), key ) < 0 ->  
#     return 1  
#     else ->  
#     return 0 ]  
if self.__compareItemKey ( skipItem, key ) < 0:  
    return 1  
else:  
    return 0
```

4.21. SkipList.match()

The `.match()` method is used to search for a specific, matching item. If there are multiple matching items, by definition it returns the first. This means we use the search cut list to locate the matching item, rather than the insert cut list. See Section 3.2.2, “The search algorithm” (p. 8).

pyskip.py

```
# - - - S k i p L i s t . m a t c h - - -  
  
def match ( self, key ):  
    """Return the first or only child with the given key.  
    """
```

If there is a matching item, the level-0 element of the search cut list will precede it; see Section 4.2.9, “**search-cut-list**” (p. 12). So we could call the `.__searchCutList()` method and use element `[0]` of the returned list.

However, for efficiency reasons, there is a simplified version of `.__searchCutList()` that doesn't build up the entire cut list, but instead just finds the level-0 predecessor: see Section 4.22, “**SkipList.__searchCutItem**” (p. 31).

```

#-- 1 --
# [ searchItem := successor of search-point ( 0, key ) ]
prevItem    = self.__searchCutItem(key)
searchItem  = prevItem.links[0]

```

If the user is looking for a child that isn't there, **searchItem** will point either at the terminator, or at a **_SkipItem** with a different key. In either of those cases, raise the **KeyError** exception. Otherwise, return the child item. See Section 4.13, "**SkipList.__compareItemKey()**" (p. 23).

```

#-- 2 --
# [ (searchItem is a _SkipItem in self) ->
#   if (searchItem is not self.__terminator) and
#   ( cmp ( key-of (searchItem's child), key ) ) == 0 ) ->
#   return searchItem's child
#   else ->
#   raise KeyError ]
if ( ( searchItem is not self.__terminator ) and
    ( self.__compareItemKey ( searchItem, key ) == 0 ) ):
    return searchItem.child
else:
    raise KeyError, "Key not found: %s" % key

```

4.22. SkipList.__searchCutItem

The logic here is basically the same as in **.__searchCutList()**, but the caller only needs to know the level-0 element of the search cut list. So, to avoid a lot of extra storage allocator calls, we don't build the entire search cut list. For more detailed commentary, see Section 4.18, "**SkipList.__searchCutList()**" (p. 28).

```

# - - -   S k i p L i s t . _ _ s e a r c h C u t I t e m   - - -

def __searchCutItem ( self, key ):
    """Find the level-0 predecessor of the given key.

       [ key is in self's key domain ->
         return search-point(0, key).link[0] ]
    """

#-- 1 --
# [ searchItem := self.__heads ]
searchItem = self.__heads

#-- 2 --
# [ if search-precedes ( searchItem, key ) ->
#   searchItem := search-point ( 0, key ) ]
for i in xrange ( self.__nLevels-1, -1, -1 ):
    #-- 2 body --
    # [ if search-precedes ( searchItem, key ) ->
    #   searchItem := search-point-after(i, key, searchItem) ]

    searchItem = self.__searchPoint ( searchItem, i, key )

```

```
#-- 3 --
self.nSearches = self.nSearches + 1
return searchItem
```

4.23. SkipList.find()

Unlike `.match()`, the `.find()` method returns not a child but a generator that visits all children at or after a given key value.

pyskip.py

```
# - - - S k i p L i s t . f i n d - - -

def find ( self, key ):
    """Return an iterator starting at a given position
    """
```

As with the `.match()` method (see Section 4.23, “**SkipList.find()**” (p. 32)), the first step is to find the level-0 predecessor of the desired position. See Section 4.22, “**SkipList.__searchCutItem**” (p. 31).

pyskip.py

```
#-- 1 --
# [ searchItem := search-point ( 0, key ).links[0] ]
prevItem = self.__searchCutItem ( key )
searchItem = prevItem.links[0]
```

At this point, all we need to do is to pass the selected `_SkipItem` to the `_SkipListIterator` constructor, and return the new iterator to the caller. See Section 4.4, “The `_SkipListIterator` class” (p. 14).

pyskip.py

```
#-- 2 --
return _SkipListIterator ( searchItem )
```

4.24. SkipList.__len__()

The special method `.__len__()` is called when the `len()` function is applied to a `_SkipItem`. It returns the value of the internal `.__nItems`.

pyskip.py

```
# - - - S k i p L i s t . _ _ l e n _ _ - - -

def __len__ ( self ):
    """Returns the number of child elements."""
    return self.__nItems
```

4.25. SkipList.__iter__()

The special `.__iter__()` method returns an iterator that walks the entire skip list in order. The logic is trivial: we pass the successor of the level-0 list head to the `_SkipListIterator` constructor. See Section 4.4, “The `_SkipListIterator` class” (p. 14).

```
# - - - S k i p L i s t . _ _ i t e r _ _ - - -
def __iter__( self ):
    """Iterator for the entire list"""
    return _SkipListIterator ( self.__heads.links[0] )
```

4.26. SkipList.__contains__()

This special method is called when the user uses the Python “**in**” or “**not in**” operators to test whether a given key value matches any of the child objects in the skip list.

```
# - - - S k i p L i s t . _ _ c o n t a i n s _ _ - - -
def __contains__( self, key ):
    """Does self contain the given key?"""
```

First we call the **.match()** method to tell us whether there's a matching element. Then we convert the return values of that method to a Boolean.

```
#-- 1 --
try:
    child = self.match ( key )
    return 1
except KeyError:
    return 0
```

4.27. SkipList.__delitem__()

The **.__delitem__()** method is another special method that is called when the user applies the Python **del** statement to one element of a skip list. It's really the same thing as the **.delete()** method, except the caller doesn't want the deleted item.

In the case where there are multiple children with the same key, it's not obvious whether a user would expect to delete all of them. We arbitrarily say it will delete only one. If one wishes to delete them all, one can always write a loop that calls **.delete()** until it returns a **None** to signify that no elements were deleted.

```
# - - - S k i p L i s t . _ _ d e l i t e m _ _ - - -
def __delitem__( self, key ):
    """Delete the first or only item with a given key."""
    self.delete ( key )
```

4.28. SkipList.__getitem__()

This special method name is called when a user retrieves an element from a skiplist using the syntax “**s[k]**”. Note that a **KeyError** exception will be raised if no child objects have a matching key.

```
# - - - S k i p L i s t . _ _ g e t i t e m _ _ - - -  
def __getitem__ ( self, key ):  
    """Get the first or only item with a given key."""  
    return self.match ( key )
```