The Python `SkipList` object TCC

A `SkipList` object is a container object for storing objects (called *child objects*) according to some ordering relation. Objects may be added or deleted at any time. When retrieved, they will be returned in the order defined by the ordering relation.

The ordering relation is defined by the usual Python `cmp()` function, which works on the obvious basic types (numbers, strings and so on) and any object that defines a built-in `__cmp__()` method.

Sometimes it is convenient to talk about a *key* type that defines the ordering relation and is different from the child type. The `SkipList` object allows you to define a function that extracts a key object from a child object. If this is not necessary, though, you can just use the child object itself as the key.

Methods

Use the `SkipList()` constructor to create a `SkipList`. The remaining methods are used to add, delete, and retrieve objects.

`SkipList()`

To create a new `SkipList` object, use this method:

```
SkipList ( keyFunction, allowDups, maxLevels )
```

where the arguments have these definitions:

`keyFunction`	If this optional argument is omitted, the child objects themselves will define the ordering relation. If you prefer to have a separate key object that defines the ordering, you must provide a function with calling sequence `keyFunction(`*c*`)` that operates on any child object *c* and returns the corresponding key object.
`allowDups`	If this optional argument has a true value, the `SkipList` object will allow you to insert more than one object with the same ordering. Otherwise attempts to insert a duplicate object will fail.
`maxLevels`	Provision of this argument is necessary only for maintaining very large sets of values. If you are storing more than a million objects, pass this argument as the smallest power of 4 that exceeds the cardinality of the set you are storing (default is 10, suitable for storing about a million objects).

`.insert()`

To add an object to a `SkipList`, use this method:

`.insert (` *c* `)`

where *c* is the child object to be inserted.

This method raises a `ValueError` exception if the object does not allow duplicates and there is already an object in the list that compares equal to *c*.

`.delete()`

To delete an object from a `SkipList`, use:

    .delete (k)

where k is a key object that is used to locate the object to be deleted. The first or only object that is equal to `value` (according to the `cmp()` function) will be deleted. This function fails if no such objects exist in the list.

`.match()`

To find the first object that has a specific key value:

    .match (k)

The k argument is a key object. If any object in the list has a key equal to k, the first one inserted is returned, and the current position is set to that object. Otherwise the current position becomes undefined and `ValueError` is raised.

`.find()`

To find an object in a `SkipList`:

    .find (k)

The k argument is a key object. If there any any objects in the list with keys $\geq k$, the current position is set to the first such object and that object is returned. If there are no such objects, `ValueError` is raised.

`.first()`

This method returns the first object in the list:

    .first()

If the list is empty, `None` is returned.

`.next()`

To access objects sequentially, use the `.first()` or `.match()` or `.find()` functions to set the current position, and then use this method:

    .next()

This method returns the next object after the current position, if there is one, and advances the current position to that object.

If the current position is on the last object in the list, this method returns `None` and causes the current position to become undefined.

If the current position is undefined, `IndexError` is raised.

Retrieving statistics

There are three members you can use to find out how efficient the searching is:

```
.nItems
.nSearches
.nCompares
```

These members are, respectively: the number of items contained in the list; the number of times you have searched the list to find an item (this includes searching for the point of insertion when adding new elements); and the number of times two items have been compared. If things work correctly, the number of compares per search should be proportional to the log of the number of items in the list.

Implementation

The technique used for storing objects is "skip lists," as described in *Skip lists: a probabilistic alternative to balanced trees*, by William Pugh, *Comm. ACM* 33(6)668-676, June 1990.

It would suffice for correctness just to keep all the objects in an ordinary linked list. However, searching a linked list has a time complexity of $O(n/2)$.

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

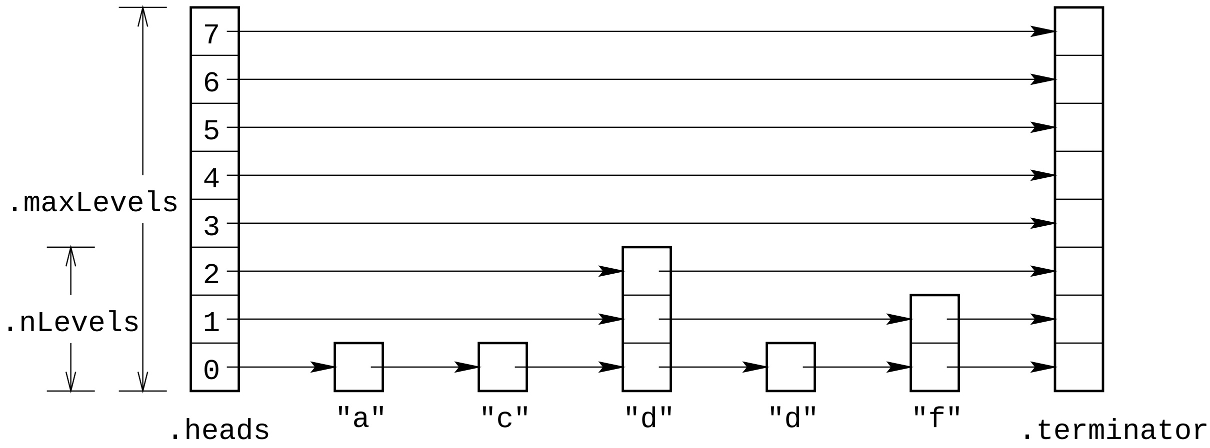
- Every object is in list 0, ordered from lowest to highest key.
- List i visits a subset of the objects in list $(i-1)$, but still in order by key.

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.

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 the highest list that actually contains a value.

Written by John W. Shipman (`john@nmt.edu`). This version printed 1997-07-17.