The SkipList object TCC

A SkipList object is a container object for storing 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 a *comparator method* that you must provide when creating a SkipList. This method must be able to compare any two objects A and B stored in the SkipList and specify whether A comes before B or after B, or whether they are equal in the ordering relation. The SkipList object is stable, that is, if two objects O_1 and O_2 are inserted, and the comparator method ranks them as equal, they will be ordered O_1, O_2 according to the order of their insertion.

Methods

Use the `Skip_List_New()` method to create a SkipList. The remaining methods are used to add, delete, and retrieve objects; in those methods, the `skipList` argument is always the value returned by `Skip_List_New()`.

Skip_List_New()

To create a new SkipList object, use this method:

```
Skip_List_New ( comparator, allowDups, maxLevels )
```

where the arguments have these definitions:

`comparator`	A method with calling sequence `comparator(`a`,`b`)` such that the return value is -1 if a is before b, 0 if they are equal, and 1 if a is after b.
`allowDups`	If this optional argument has a value other than `&null`, 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).

The return value is a SkipList object.

Skip_List_Insert()

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

```
Skip_List_Insert ( skipList, value )
```

where `value` is the object to be inserted. Note that `value` must be in the domain of the `comparator()` function.

This method fails if the object does not allow duplicates and there is already an object in `skipList` with the same value, as determined by the `comparator()` function.

Skip_List_Delete()

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

```
Skip_List_Delete ( skipList, value )
```

where `value` is an object in the domain of the `comparator()` function that is used to locate the object to be deleted. The first or only object that is equal to `value` (according to the `comparator()` function) will be deleted. This function fails if no such objects exist in the `skipList`.

Skip_List_Search()

To find an object in a `SkipList`:

```
Skip_List_Search ( skipList, value )
```

This method generates all objects from the given `skipList` that are equal to `value`, as determined by the `comparator()` function. If there are no such objects, the method fails. If the list allows duplicates, and there are multiple objects that match, the objects will be returned in the same order they were inserted.

Skip_List_Search_Range()

You can request all the objects in a given range by:

```
Skip_List_Search_Range ( skipList, value, stopValue)
```

This method will generated all the objects in the given `skipList` whose ordering is $\geq$`value` and `<stopValue`.

For example, suppose that the values in a list are ordered by a simple string as key, and the `comparator(a,b)` function returns -1 if `a<<b`, 0 if `a==b`, and 1 if `a>>b`. Then to generate all the values whose keys start with the letter C, you would use:

```
every value := Skip_List_Search_Range ( skipList, cObject, dObject ) ...
```

where `cObject` has the key `"C"` and `dObject` has the key `"D"`.

If the `value` argument is `&null` or omitted, the generated values start at the beginning of the sequence. Similarly, if the `stopValue` argument is `&null` or omitted, the generated values will go all the way to the end of the sequence. So this call would generate all the items in the entire list in sequence:

```
every value := Skip_List_Search_Range ( skipList ) ...
```

Retrieving statistics

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

```
Skip_List_N_Items ( skipList )
Skip_List_N_Searches ( skipList )
Skip_List_N_Compares ( skipList )
```

These methods return, 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 1, such that:

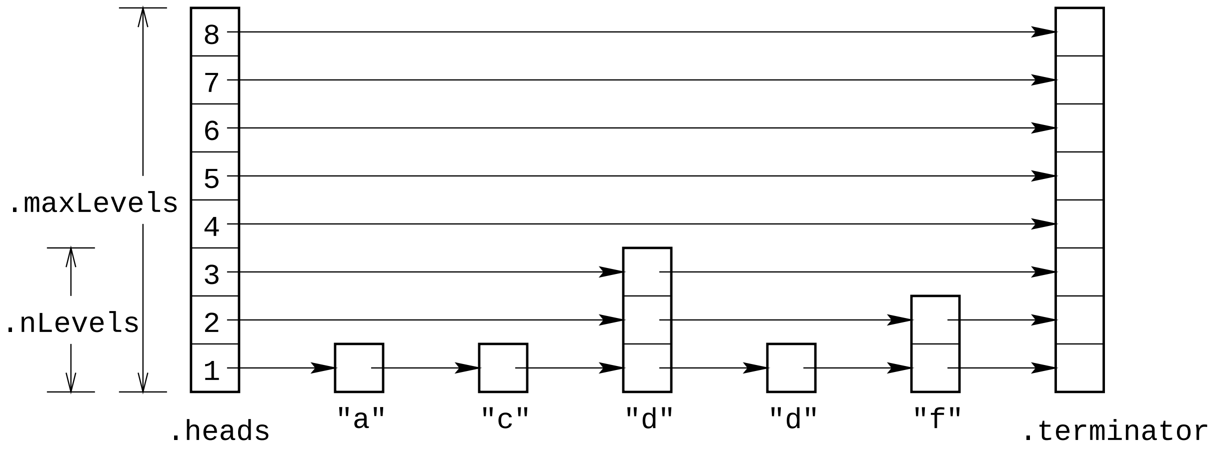
- Every object is in list 1, ordered from lowest to highest key.
- List i visits a subset of the objects in list $(i-1)$, but still in order by key.

In practice, when each new element is inserted, it is always added to list 1, 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 1.

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 the Icon `record` structure for 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 1996-12-13.