1 The Hash Table Library

The hash-table library (‘`hash.el`’) contains a rich set of functions for dynamic hash tables, but the simplest uses only require the *get* and *put* functions.

1.1 What are Hash Tables?

Hash tables, sometimes also known as associative arrays, or scatter tables, are a data structure that offers fast lookup, insertion, and deletion of (key, value) pairs. In a well-designed implementation of hash tables, all of these operations have a time complexity of *O(1)*, rather than the *O(n)* that linked lists, association lists, and vectors would require for some of them.

Hashing was discovered and first published and implemented on computers in the early 1950s, and is based on a simple idea: transform the key to a number (the *hash* process), and then use that number to index a table. In order to keep the table size manageable, reduce the computed number modulo the table size.

As an example, suppose we choose a hash function that is the ASCII value of the first character of the key: then `(hash "aardvark")` is 97, `(hash "bat")` is 98, `(hash "cat")` is 99, and so on. We can store associated data for these strings in an array at positions 97, 98, 99, ..., and clearly, we can also retrieve that data with a simple linear-time array lookup.

However, as soon as we introduce `"antelope"`, `"badger"`, and `"chipmunk"` to our key set, we have a problem: our hash function produces *collisions*. `"aardvark"` and `"antelope"` both hash to the same table location, and similarly for the other new keys.

Collision handling is the price we must pay for the bonus of *O(1)* complexity, and if the mechanism for doing so is poorly designed, performance may well deteriorate to *O(n)*, or worse. Fortunately, the problem has been well studied, and several practical algorithms are now known for dealing with it.

In our simple example above, the hash function is poor. Better ones involve all of the characters in the key: one reasonably fast algorithm multiplies the hash value computed so far by a suitably-chosen constant multiplier, and adds the ASCII value of the next character, carrying out the computation in unsigned arithmetic so that overflow just wraps, and then finally, the computed hash value is reduced modulo the table size to convert it to a table index. This hash function is fast, requiring only one multiply and one add for each key character.

Cryptographic checksums of the key string would also work well, but are often considerably slower than the simple scheme just outlined.

Notice, however, that hash functions do not preserve key order: even if keys are entered in, say, alphabetical order, they will in general be stored in disorder in the table. Thus, the extra work of a sorting step may be needed, if key order is to be restored when the table is traversed.

One of the simplest, yet still highly effective, algorithms is called *linear probing with overflow chains*, and this is the method that GNU Emacs implements in its low-level `obarray` facility; see ⟨undefined⟩ [Creating Symbols], page ⟨undefined⟩. Instead of storing a value directly in the hash table, store a pointer to a list of (key, value) pairs whose keys all hash to that index. Lookup then requires searching that list, comparing keys until the desired one

is found. Provided that list is not long, this is still much better than brute-force comparison against an average of *n/2* keys in a linked list of all (key, value) pairs.

Overflow chains can be kept short by a combination of three actions: choose a good hash function, ensure that the table does not become too full, and select a table size that is a prime number (so that collisions that do occur are spread out through the table).

You might wonder whether it would be possible, for a given set of keys, to find a hash function that produces no collisions at all; lookup would then require neither string comparisons nor overflow chain traversals. The answer is yes, but it is not computationally fast to do so. Such a hash function is called a *perfect hash function*. If, in addition, the table size that it requires is identical to the number of keys, so that no wasted space is present in the table, we have a *minimal perfect hash function*. More details can be found in the literature cited in the next section.

1.1.1 Further reading

Most computer science books on algorithms have a chapter on hashing; the definitive treatise on the subject is Donald E. Knuth's *The Art of Computer Programming, Volume 3, Sorting and Searching*, second edition, 1998, pp. 513–558, ISBN 0-201-89685-0.

An extensive bibliography of research papers on hash algorithms can be found at `http://www.math.utah.edu/pub/tex/bib/index-table-h.html#hash`.

Software for generating perfect hash functions can be found in the GNU `gperf` package, although its algorithms only work well for up to a few hundred keys. Nevertheless, in 1992, practical methods were found for generating minimal perfect hash functions for key sets of up to a million keys, which has important applications in construction of constant databases, such as those provided on CD-ROMs; see the work of Edward A. Fox and coworkers cited in the bibliography file above.

1.2 Hash Library Conventions

All of the public functions and variables in the hash library begin with the prefix *hash-*, to reduce the likelihood of collision with user-defined names, and names from other GNU Emacs libraries.

To use the hash library in your own code, simply include this line near the beginning of your Emacs Lisp file:

```
(require 'hash)
```

The hash library supports *dynamic hash tables*: they grow as needed to support the data stored in them, so you need never worry about how much, or how little data, you will store in them. Lookup, insertion, and deletion all remain of complexity *O(1)*, no matter how many keys are stored.

All functions in the hash library return a value, as described in their docstrings. Also, each of the docstrings ends with a short paragraph that documents the run-time cost of the function, so that you can use that information to compute cost estimates of your own higher-level functions.

All public hash library functions taking a *hashtable* argument allow that argument to be omitted, or equivalently, specified as `nil`: in such a case, a default table, called *hash-*

default-table, will be automatically supplied internally. It is a perfectly ordinary hash table, and can be used with the library functions just like any other hash table.

However, because hash table storage works just like normal assignment to programming language variables (the last value stored replaces previous values), to avoid possible data loss from other packages that also use the hash library and omit the *hashtable* argument, you should make a practice of supplying it, except for short temporary code fragments.

You can have as many hash tables as you like: each provides an independent name space for storing (key, value) pairs.

The *key* passed to the hash library functions may be *any* valid Emacs Lisp object: strings are most efficient, but any other object type will be automatically converted internally to a string by using `prin1-to-string`.

The *value* argument can be *any* valid Emacs Lisp object: it is not further examined or converted in any way, and in particular, large objects are as cheap to handle as small ones.

1.3 Creating and Deleting Tables

With any algorithm for data access, we require at least four basic operations: *create*, *delete*, *get*, and *put*. These are analogous to those for I/O with files: *open*, *close*, *read*, and *write*.

Here are the functions provided in the hash library for the first two operations:

hash-create-table &optional *case-insensitive size rehash-size rehash-threshold* Function

Create and return a new empty hash table.

[cost: *O(n)*]

hash-delete-table &optional *hashtable* Function

Delete *hashtable*, recovering (most of) the storage it used, and return `nil`.

[cost: *O(n)*]

All of the arguments to `hash-create-table` can generally be omitted, but they are available for more precise control if desired. Invalid, or out-of-range, arguments are silently replaced by suitable defaults.

case-insensitive

Frequently, letter case in keys should be ignored. For example, in a BibTeX application, the citation keys '`knuth:1984:tb`', '`KNUTH:1984:TB`', and '`Knuth:1984:Tb`' are equivalent.

Although it would be possible to wrap each *key* argument in function calls to convert keys to a uniform letter case, it would be tedious and error-prone to do so.

When this argument is set non-`nil`, all keys supplied for this table will automatically be converted internally to lowercase prior to use.

size Specify an initial size for the table. In the rare cases where you know in advance approximately, or exactly, how many keys will be stored, you can save a little time by allocating a table somewhat larger than the number of keys, so that the table doesn't have to grow repeatedly until the required size is reached.

rehash-size
When a hash table gets too full, hash collisions become more frequent, and access time deteriorates. This argument controls by how much the table is automatically enlarged. It can be either an integer greater than zero, specifying the number of elements to add when the table becomes full, or a floating-point number greater than one: the ratio of the new size to the old size. Thus, a value of 2.0 will cause the table size to double each time it needs to be grown.

rehash-threshold
You can control the point at which the table is automatically enlarged by providing this argument. It is an integer between one and *size*, inclusive (in which case the table is grown when the number of elements stored exceeds that value), or a floating-point number between zero and one, exclusive, representing the fraction of the table that is allowed to be filled before growing it. If it is an integer value, it is adjusted suitably when the table grows.

1.4 Storing, Retrieving, and Removing Table Entries

The remaining two basic operations for hash tables get and put (key, value) pairs:

hash-put-entry *key value* &optional *hashtable* Function
Store a (key, value) pair in *hashtable*, and return *value*.

[cost: *O(1)*, or *O(n)* if table grows]

hash-get-entry *key* &optional *hashtable* Function
Return the *value* corresponding to *key*. If the key does not exist in the table, return `nil`.

Thus, it is not possible with `hash-get-entry` alone to distinguish between a missing value, and a `nil` one: if this matters to you, you must arrange to store some other magic value, instead of `nil`, when you call `hash-put-entry`, or you must use an existence-test function: see Section 1.6 [Testing for Table and Entry Existence], page 6.

[cost: *O(n ln n)*]

Occasionally, you may need to remove an entry from a hash table: do so like this:

hash-delete-entry *key* &optional *hashtable* Function
Delete the (key, value) pair, and return `t` if the key existed, and `nil` otherwise.

[cost: *O(1)*]

1.5 Iterating over Table Entries

Besides *O(1)* random access to (key, value) pairs, some applications need to process the pairs after storage, without knowing in advance what the keys are.

Two hash library functions make this convenient:

hash-apply *funct* &optional *hashtable arg* Function

Call (*funct key value arg*) for each (key, value) pair in *hashtable*, and return `nil`.

[cost: *O(n)*]

The order of these calls is, in general, unpredictable, but the function will be called exactly once for each pair in the table.

hash-apply-sorted *funct* &optional *predicate hashtable arg* Function

Call (*funct key value arg*) for each (key, value) pair in *hashtable*, where the keys are processed in sorted order, and return `nil`.

The sort order is determined by the `predicate` argument, a function of two arguments that returns `t` if the arguments are in order, and `nil` otherwise. The default value if this argument is omitted, or supplied as `nil`, is `(function string-lessp)`. The arguments passed to the `predicate` function are strings representing the hash table keys.

Because an optimal sorting algorithm using key comparisons has worst-case complexity *O(n lg n)*, this function can be several times more expensive than `hash-apply` for large *n*.

However, the sorting is done behind the scenes, so either function is equally convenient for the Emacs Lisp programmer: simply avoid this second version when you really don't require a particular key order.

[cost: *O(n lg n)*]

These two functions are the *only* way to iterate over the elements of a hash table. No analogue is provided of the conventional loop body

```
while have-an-item
do
    get-the-item
    process-the-item
end while
```

common in other programming languages.

Although this might have been feasible with a different choice of primitives in the Emacs Lisp kernel, it is not efficiently practical to do so in the current Emacs implementation. Instead, think of `hash-apply` and `hash-apply-sorted` as similar to `mapcar` for Lisp sequences (lists, vectors, bool-vectors, and strings), and `mapatoms` for obarrays.

1.6 Testing for Table and Entry Existence

Many applications will probably require only the four basic functions described in earlier sections.

Nevertheless, since Lisp is a dynamically-typed language, it is essential that programs can determine the type of Lisp objects at run time. This is traditionally done with Boolean predicate functions, conventionally named with the data type followed by a terminal `p` or `-p`: the functions return non-`nil` if the object has that type, and `nil` otherwise.

The hash library therefore provides these two functions:

hash-entry-p *key* &optional *hashtable* Function

Return *key* if it is present in the table, and otherwise, return `nil`.

[cost: *O(1)*]

hash-table-p *object* Function

Return *object* if it looks like a hash table, and otherwise, return `nil`.

[cost: *O(1)*]

1.7 Querying Table Attributes

The functions described in the preceding sections provide ways to create and delete hash tables, and handle (key, value) pairs in them.

The functions described here provide information about the hash table itself, based on the design principle that anything that you can store in a program library, you must also be able to retrieve.

hash-get-case-insensitive &optional *hashtable* Function

Return the *case-insensitive* flag from the hash table.

[cost: *O(1)*]

hash-get-cursize &optional *hashtable* Function

Return the current size of the hash table, that is, the number of (key, value) pairs actually stored in it.

[cost: *O(1)*]

hash-get-maxsize &optional *hashtable* Function

Return the maximum size of the hash table; this is the value used in the final modulo operation in the hash function. This is *not* the maximum number of elements that could be stored in it, because the overflow chains can be arbitrarily long. It may differ from the value passed to `hash-create-table`, because it is always adjusted to the nearest prime number at least as large as the requested size when the table is created, and it is similarly adjusted when the table grows.

[cost: *O(1)*]

hash-get-rehash-size &optional *hashtable* Function
Return the *rehash-size* value of the hash table. Like the maximum size, this may differ from the value set in the original call to `hash-create-table`.

[cost: *O(1)*]

hash-get-rehash-threshold &optional *hashtable* Function
Return the *rehash-threshold* value of the hash table. This too may differ from that set in the original call to `hash-create-table`.

[cost: *O(1)*]

1.8 Recovering Keys and (Key, Value) Pairs

The final set of functions provides a convenient way to recover the keys, or both keys and values, from a hash table.

hash-get-key-list &optional *hashtable* Function
Return a list of keys, in arbitrary order.

[cost: *O(n)*]

hash-get-key-list-sorted &optional *predicate hashtable* Function
Return a list of keys, in sorted order. `predicate` is a key string order test function, as with `hash-apply-sorted`.

[cost: *O(n lg n)*]

hash-get-key-value-list &optional *hashtable* Function
Return a list of (key, value) pairs, in arbitrary order.

[cost: *O(n)*]

hash-get-key-value-list-sorted &optional *predicate hashtable* Function
Return a list of (key, value) pairs, in sorted order. `predicate` is a key string order test function, as with `hash-apply-sorted`.

[cost: *O(n ln n)*]

1.9 Testing and Profiling the Hash Library

The hash library file, '`hash.el`', is accompanied by a test package, '`test-hash.el`', a practice that we hope other GNU Emacs Lisp package writers will follow.

The test package contains a test function for each public function in the hash library, plus two driver programs that the (human) tester can invoke interactively with the usual `M-x` prefix, and two interfaces to those functions, to be used in batch mode as part of an automated package validation test:

test-hash Function

Run all of the validation tests. The test output log is stored in a buffer named '`*test-hash*`' (or whatever you have set *test-primes-buffer* to); an existing buffer of that name is made unique by addition of a numeric suffix. If all of the tests are successful, the buffer just contains a list of the test names, something like this:

```
There should be no output here other than the test names

test of hash-create-delete-table ...
test of hash-entry-p ...
test of hash-delete-entry ...
test of hash-apply ...
...
```

The tests are hierarchically ordered, since, for example, you cannot test whether an entry can be properly deleted until you know that the hash table can be created in the first place.

test-hash-with-profile Function

Run `test-hash` with function profiling turned on. This produces the normal test log in the '`*test-hash*`' buffer, and in addition, produces a second temporary buffer, '`*profile*`' (or whatever you have set *profile-buffer* to), to hold the run-time profile showing counts and execution times for each function profiled. A fragment of it looks like this (slightly reformatted to reduce line width):

```
Function                 Calls  Total time (sec)  Avg time
                                                  per call
======================   =====  ================  ========
hash-apply                  41          0.144627  0.003527
hash-apply-sorted            6          1.129990  0.188332
hash-create-table          115          0.215469  0.001874
hash-delete-entry            7          0.001155  0.000165
...
Profile by decreasing average time
Function                 Calls  Total time (sec)  Avg time
                                                  per call
======================== =====  ================  ========
...
hash-apply-sorted            6          1.129990  0.188332
hash-get-key-value-list     14          0.919643  0.065689
hash-get-key-list-sorted    13          0.114579  0.008814
```

The first page of the buffer contains the function names in alphabetical order. In the second page, the results are sorted by descending cost.

test-hash-and-kill-emacs Function

This function is a wrapper for `test-hash`, except that it saves the test results in a file, and exits Emacs with a status code indicating the number of test failures.

test-hash-with-profile-and-kill-emacs Function

This function is a wrapper for `test-hash-with-profile`, except that it saves the test results and profile in files, and exits Emacs with a status code indicating the number of test failures.

For the latter two functions, the filenames chosen are of the form

```
test-hash.results.HOSTNAME.YYYY-MM-DD-hh-mm-ss
test-hash.profile.HOSTNAME.YYYY-MM-DD-hh-mm-ss
```

so that tests can be run on multiple machines without filename collisions, and the test results can readily be distinguished by the filenames.

Not only does profiling reveal hot spots in the code, but non-zero function invocation counts also verify that each function has been exercised by the tests.

The exact results of a profile clearly depend on test data, on the compiler and optimization level used to build Emacs, on algorithms in the Emacs kernel, on the operating system, on the timer granularity, and on the host architecture.

Nevertheless, this table of relative performance (larger is slower), sorted by function names, may be a useful guide. It was produced on a late 1995-vintage Sun UltraSPARC 170 workstation with Sun Solaris 2.6 running GNU Emacs 20.3.6 at the package author's site, and all Emacs code was byte-compiled:

```
hash-apply                        6.94
hash-apply-sorted               306.62
hash-create-table                 4.09
hash-delete-entry                 0.36
hash-delete-table                 5.78
hash-entry-p                      0.31
hash-get-case-insensitive         0.23
hash-get-cursize                  0.23
hash-get-entry                    1.00
hash-get-key-list                 2.16
hash-get-key-list-sorted         16.57
hash-get-key-value-list         144.73
hash-get-key-value-list-sorted  153.25
hash-get-maxsize                  0.22
hash-get-rehash-size              0.22
hash-get-rehash-threshold         0.22
hash-put-entry                    1.87
hash-table-p                      0.31
next-prime                        6.84
prime-p                           1.41
```

Here is the same data, sorted by descending relative cost:

```
hash-apply-sorted               306.62
hash-get-key-value-list-sorted  153.25
hash-get-key-value-list         144.73
hash-get-key-list-sorted         16.57
hash-apply                        6.94
next-prime                        6.84
hash-delete-table                 5.78
```

```
hash-create-table               4.09
hash-get-key-list               2.16
hash-put-entry                  1.87
prime-p                         1.41
hash-get-entry                  1.00
hash-delete-entry               0.36
hash-entry-p                    0.31
hash-table-p                    0.31
hash-get-case-insensitive       0.23
hash-get-cursize                0.23
hash-get-maxsize                0.22
hash-get-rehash-size            0.22
hash-get-rehash-threshold       0.22
```

As predicted above, *hash-apply-sorted* is the most expensive function.

Notice that a *put* operation is about twice as expensive as a *get*, because it requires additional time for storage allocation, and occasionally, automatically enlarging the hash table.

The cheapest functions at the end of the display above involve little more than the overhead of an inlined function call, and an indexed vector lookup, so they are almost minimal Emacs Lisp functions. It is gratifying that a hash function *get* operation is only about five times as expensive as these simplest functions.

Table creation and deletion have only very modest costs, so in practice, the cost of processing the data is likely to overwhelm that for storage and retrieval in dynamic hash tables, demonstrating the great efficiency of hashing.