To improve the performance of zencoding-mode, caching/memoization of previously entered input needed to be done. At first, a naive hash-table implementation was chosen and this greatly cut down the speed with which zencoding generated HTML. Then it was realized that a trie data structure would be more efficient as the hash-table was storing partially entered input as well. A trie could store the partially entered data when needed and use less space than a hash-table.

A trie is a tree where each node represents a character in a string. When you reach a leaf node, a word or sentence is spelled out. Each node can contain a value.

Important: the key is not stored in the node, but in the parent of that node.

The only operations we require for our trie are *retrieve value* and *insert value.*

A node is a sequence of two elements. The first element is some value, in our case the cached version of zencoding's AST (Abstract Syntax Tree) output. If this is `NIL` then there is no value stored in this node. The second element is a sequence of branches (more nodes).

⟨*Node creation function*⟩≡
```
(defun make-zencoding-trie-node ()
  "Creates a vector of two elements. The first element is some value,
the second element is a character table for storing more branches of the
trie. The value is initially NIL."
  (vector nil (make-char-table 'trie-table)))
```

We need some functions for setting and getting the values of our node type. All of the functions accept a node as an argument which must be a sequence type with at least two elements. The *value* argument can be anything, typically something non-`NIL`. The $branch - key$ argument must be a character.

⟨*Node accessors*⟩≡
```
(defun zencoding-trie-node-value (node)
  (aref node 0))

(defun zencoding-trie-node-set-value (node value)
  (aset node 0 value))

(defun zencoding-trie-node-branches (node)
  (aref node 1))

(defun zencoding-trie-node-branch (node branch-key)
  (aref (zencoding-trie-node-branches node) branch-key))
```

We need a function to access a particular branch in a node. If the branch does not exist, it will be created.

⟨*Node accessors*⟩+≡
```
(defun zencoding-trie-node-create-branch (node branch-key)
  (aset (zencoding-trie-node-branches node)
        branch-key
        (make-zencoding-trie-node)))

(defun zencoding-trie-node-brancher (node branch-key)
  (let ((branch (aref (zencoding-trie-node-branches node) branch-key)))
    (if branch
        branch
      ;; branch doesn't exist, so create it and then return it.
      (zencoding-trie-node-create-branch node branch-key))))
```

The retrieval function is given a string s to search for and the *trie* to search within. Where n is the length of the string s, the argument i is bounded like so: $0 < i \leq n$.

We are moving forward through a string till we reach its end. The current character is s_{i-1}. If the current node we are visting in the trie contains the current character as one of its branches, then we have to visit it.

When the end of s has been reached, $i = n$, then we have reached the correct node and can return it (it will be created if it doesn't already exist). Before this, we will have to keep on searching for the right node.

If we reach the end of the trie before the correct node has been found, we return `NIL`.

The traverse function is a generalized version which includes a function argument, f, telling us how to select the next branch to visit.

⟨*Retrieve node operation*⟩≡
```
(defun zencoding-trie-traverse (s trie i f)
  (if (null trie)
      nil
    (let ((branch (funcall f trie (aref s (1- i)))))
      (if (or (null branch) (= i (length s)))
          branch
        (zencoding-trie-traverse s branch (1+ i) f)))))

(defun zencoding-trie-retrieve (s trie)
  (zencoding-trie-traverse s trie 1 'zencoding-trie-node-branch))
```

Next we have a function for inserting a string and its value into a trie. The pre-conditions are that *s* is a type of string, *value* and *trie* are non-NIL.

We use the previously defined retrieve operation to find the final node and then insert the *value* there. Fortunately for us, the retrieve operation takes care of creating new branches if necessary.

⟨*Insert node operation*⟩≡
```
(defun zencoding-trie-insert (s value trie)
  "‘s’ is the string used to navigate the trie (each character is a node
in the trie). ‘value’ is the value we want to insert. ‘trie’ is the node
where we start the insertion."
   (zencoding-trie-node-set-value
    (zencoding-trie-traverse s trie 1 'zencoding-trie-node-brancher)
    value))
```

The code is licensed under the GNU GPL version 3 or later.

⟨*Copyright info*⟩≡
```
;;; zencoding-trie.el --- A trie data structure built for zencoding-mode
;;
;; Copyright (C) 2009, Rudolf Olah
;;
;; Author: Rudolf Olah <omouse@gmail.com>
;;
;; This file is free software; you can redistribute it and/or modify
;; it under the terms of the GNU General Public License as published by
;; the Free Software Foundation; either version 3, or (at your option)
;; any later version.
;;
;; This file is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;; GNU General Public License for more details.
;;
;; You should have received a copy of the GNU General Public License
;; along with GNU Emacs; see the file COPYING.  If not, write to
;; the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor,
;; Boston, MA 02110-1301, USA.
;;
```

The final file looks like this:

```
⟨zencoding-trie.el⟩≡
  ⟨Copyright info⟩

  ⟨Node creation function⟩

  ⟨Node accessors⟩

  ⟨Retrieve node operation⟩

  ⟨Insert node operation⟩

  (provide 'zencoding-trie)

  ;; test code
  (let ((x (make-zencoding-trie-node)))
    (zencoding-trie-insert "ha" 1 x)
    (zencoding-trie-insert "hi" 2 x)
    (zencoding-trie-retrieve "ha" x))
```