1 Trie

Motivation for this data structure is to enable fast retrieval of strings and their common prefixes. Operations of interest are: finding, inserting and deleting key.

1.1 Definition

Trie (also known as prefix tree or digital tree) is a tree T defined over alphabet L with the following properties:

1. There is one root $r(T)$ only.
2. Each node x has arbitrary number of children determined by an array $d_x[i]$, where $i \in L$. If x is leaf, then $d_x[i]$ is empty array i.e. it's length $|d_x|$ is zero.
3. For each child $d_x[i]$ there is character $c(x) \in L$ from alphabet L which determines prefix for x.
4. All leaves and some of internal nodes have associated keys $k(x)$ and corresponding values $v(x)$ determined by characters on the path from root to x. Position of x defines a key associated with it by appending all characters on the path from the root to x.

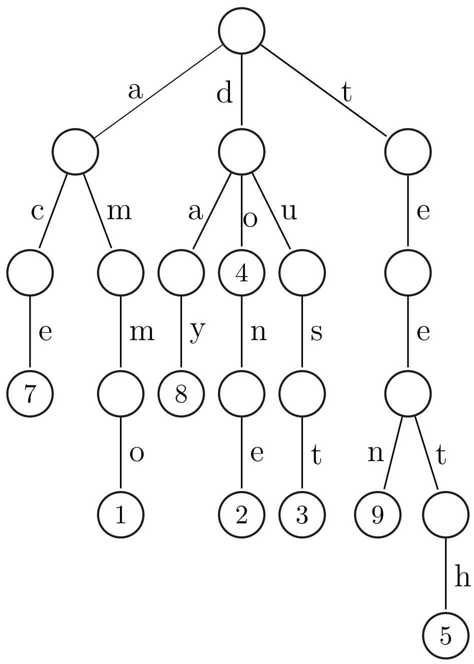

Figure 1: Trie with keys/values: ace/7, ammo/11, day/8, do/4, done/2, dust/3, teen/9, teeth/5

In the pseudo code, root of trie is denoted with $\mathsf{root}(T)$. Children of a node x are accessed via $\mathsf{child}(x, i), i \in L$; if character i is not present, then the child function returns null. Number of children of x is given with $\mathsf{size}(x)$. Value is accessed with $\mathsf{value}(x)$. Keys of x's children are taken with $\mathsf{key}(x, i), i \in L$. Length of a key w is $|w|$.

1.2 Finding key

The operation checks if the given key K exists in a trie T. It goes one by one character of K until the corresponding child exists. If all characters are traversed, then the key K is found; otherwise, the key does not exist.

Input: string K to check for existence in a trie T
Output: key/value pair if exist or null if no such key is present
Worst Case Complexity: $O(|K|)$

```
find(K)
  x = root(T)
  for i = 1 to |K|
    chr = K[i]
    if child(x, chr) ≠ null
      x = child(x, chr)
    else
      return null
  return (key(x), value(x))
```

1.3 Inserting

The operation puts a key/value pair (K, V) into trie T. It goes one by one character of K and checks whether they exist from the root down to leaves. If a character of K is not found on the path, it is added, as well all remaining characters. If all characters of K exist on the path, then the reached node is updated with value V.

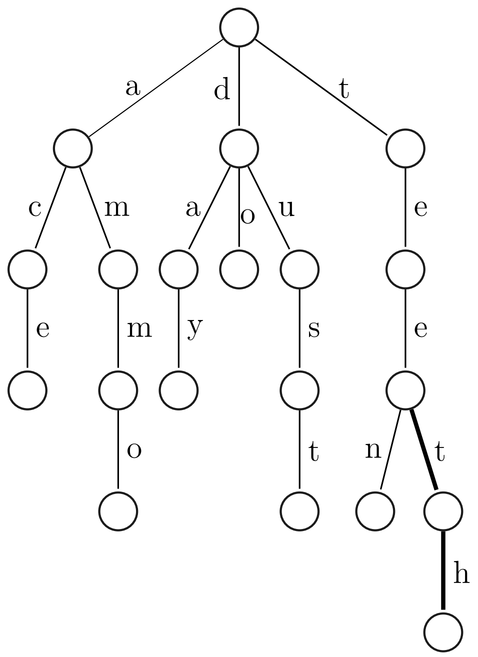

Figure 2: Inserting key *teeth* into trie; bold edges are newly created

Input: key/value (K, V) to insert into trie T
Output: T with (K, V)
Worst Case Complexity: $O(|K|)$

```
insert(K, V)
  if root(T) = null
    new root(T)
  p_x = x = root(T)
  for i = 1 to |K|
    chr = K[i]
    x = child(x, chr)
```

if $x =$ **null**
break
$p_x = x$
while $i <= |K|$
$chr = K[i]$
new x
$\mathsf{child}(p_x, chr) = x$
new $\mathsf{key}(p_x, chr)$
$p_x = x$
$\mathsf{value}(x) = V$

1.4 Deleting

Deleting removes a key K from a trie T by traversing path of K and removing nodes which do not have children anymore. It traverses all characters of K from the root down to leaves. Then it goes back by the same path in bottom-up manner to delete nodes without children. When a non-empty node is reached, the deletion is over.

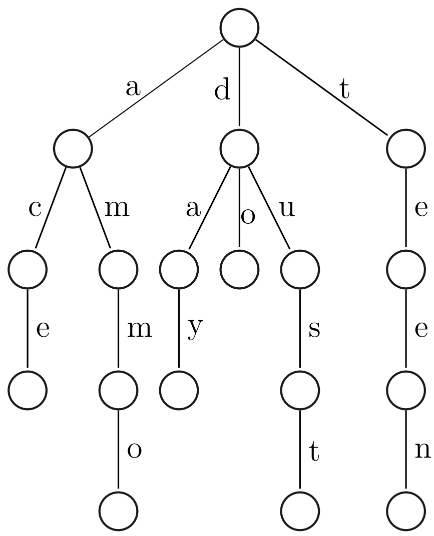

Figure 3: Trie after deleting key *teeth*.

Input: key K to delete in trie T
Output: T without key K and true returned, false if no such key is present
Worst Case Complexity: $O(|K|)$
$\mathsf{delete}(K)$
if $T =$ **null**
return false
$\{$*stack of children traversed on the path of the given key*$\}$
new $children$
$node = \mathsf{root}(T)$
for $i = 1$ **to** $\mathsf{size}(K)$
$chr = K[i]$
if $\mathsf{child}(node, chr) \neq$ **null**
$node = \mathsf{child}(node, chr)$
$\mathsf{push}(children, node)$
else

```
    return false
{go up along the path and delete empty nodes}
i = |K|
chr = K[i]
node = pop(children)
do
  if size(node) ≠ 0
    break
  delete node
  {proceed with node parent}
  node = pop(children)
  delete key(node, chr)
  i = i − 1
  if i = 0
    break
  chr = K[i]
```

1.5 Worst case complexity

Theorem 1.1. For a trie T and a key K, finding, inserting and deleting the key have complexity $O(|K|)$,

Proof All operations have loops of size $|K|$, thus their complexity is $O(|K|)$. **QED**

1.6 Applications

- Associative array.
- Lexicographic sorting.
- Radix sort.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein: Introduction to Algorithms; Second Edition

[2] Miodrag Zivkovic: Algoritmi

[3] Robert Sedgewick: Algorithms

[4] Daniel Dominic Sleator, Robert Endre Tarjan: Self-Adjusting Binary Search Trees