1 Radix tree

Motivation for this data structure is to optimize space usage of nodes, so there are no nodes with only one child.

1.1 Definition

Radix tree is a trie such that each node which is the only child of its parent is merged to its parent.

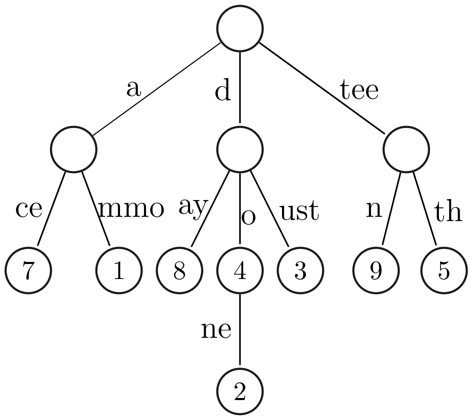

Figure 1: Radix tree with keys/values: ace/7, ammo/1, day/8, do/4, done/2, dust/3, teen/9, teeth/5

In the pseudo code, root of radix tree is denoted with root. For each node $x \in T$, number of children is given with $x.\mathsf{size}$, keys are accessed via $x.\mathsf{key}[i]$ where $i = 1, \ldots, x.\mathsf{size}$, value with $x.\mathsf{value}$, children are found over $x.\mathsf{child}[c]$, where c are corresponding keys over the alphabet L. Length of a string w is $\mathsf{length}(w)$. $\mathsf{substring}(w, i, n)$ returns substring of w of length n starting from i-th character.

1.2 Search

For a given key K, start from the root by finding node x with a key that matches K's prefix. While there is such node, proceed with the procedure on x's children.

Input: string K to find in radix tree T
Output: value if exists, null if no such key is present
Worst Case Complexity: $O(|K|)$

```
find(K)
   x := root
   len := 0
   found := true
   while len ≤ length(K) and found = true
      found := false
      for i := 1 to x.size
         k = substring(K, len + 1, length(x.key[i]))
         if k = x.key[i]
            x := x.child[k]
            len := len + length(k)
            found := true
            break
   if found = true
```

return x.value
else
return null

1.3 Insert

Inserting key/value pair (K, V) into radix tree T goes by finding corresponding nodes which match a prefix of K. The rest of K (if any) is put into T.

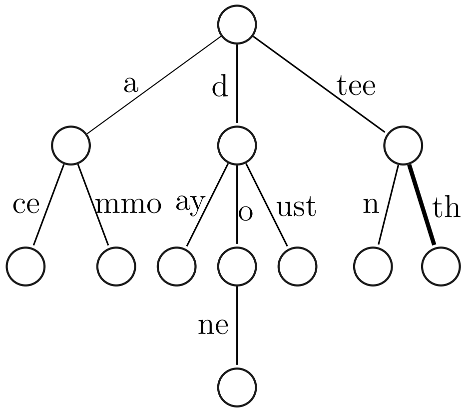

Figure 2: Inserting *teeth* into radix tree; bold edge is created

Input: key and value (K, V) to insert into radix tree T
Output: T with added (K, V)
Worst Case Complexity: $O(|K|)$

```
insert(K, V)
  {find path that matches K's prefix }
  px := x := root
  len := 0
  found := true
  while len ≤ length(K) and found = true
    found := false
    for i := 1 to x.size
      k = substring(K, len + 1, length(x.key[i]))
      if k = x.key[i]
        px := x
        x := x.child[k]
        len := len + length(k)
        found := true
        break
  {if K's suffix which did not match existing keys exists, add it to a new child}
  if len < length(K)
    new x
    k = substring(K, len + 1, length(K) − len)
    px.child[k] = x
    px.key[px.size + 1] = k
    px.size := px.size + 1
  x.value := V
```

1.4 Delete

To delete key K in radix tree T, find a corresponding node x for the key K; let p_x be x's parent. If x is a leaf, then it is deleted. In case that p_x after deletion of x remains with only one child y, then y's key is appended to p_x's.

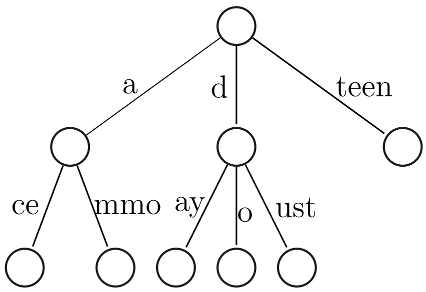

Figure 3: Radix tree after deleting key *teeth*

Input: key K to delete from radix tree T
Output: T with K deleted
Worst Case Complexity: $O(|K|)$

```
delete(K)
   px := x := root
   px_key := x_key := null
   x_index := 0
   len := 0
   found := true
   while len < length(K) and found = true
      found := false
      px_key := x_key
      for i := 1 to x.size
         k := substring(K, len + 1, length(x.key[i]))
         if k = x.key[i]
            px := x
            x := x.child[k]
            x_key = k
            x_index = i
            len := len + length(key)
            found := true
            break
   x.value = null
   {in case the key from a leaf is deleted, remove the leaf}
   if len = length(K)
      delete x
      delete px.child[x_key]
      delete px.child[x_index]
      px.size := px.size − 1
      {in case single child remains, concatenate key with the parent key}
      if px.size = 1
         y_key := px.key[1]
         y = px.child[y_key]
         px_key := px_key + y_key
         delete px.key[1]
         delete px.child[y_key]
```

delete y
$px.\mathsf{size} = 0$

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**

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