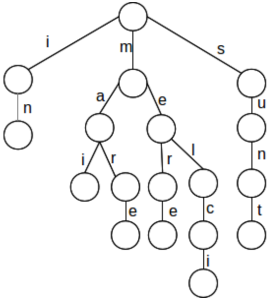
Tries

DS 2018/2019

Content

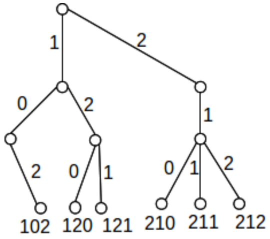
Tries

- *Information re***trie***val*
- A data structure used to work with strings that benefit from their structural properties
- The memory space required to represent a dictionary is reduced: the common root is represented only once
- Economy of memory when there are many common prefixes



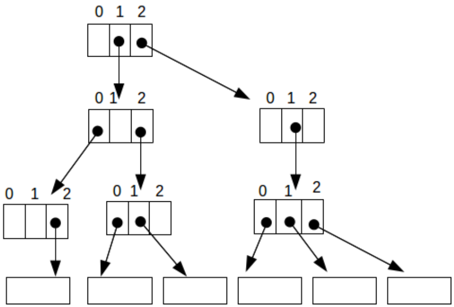
Tries

- A data structure based on the digital representation of the elements
- A tree with a root k-th ordered, where k is the number of digits (letters of an alphabet)
- It is assumed that the elements are represented by sequences of digits (letters) of the same length m ($|U| = m^k$)



Tries - the data structure

- A linked list of nodes where each node has k children
- Suppose the alphabet $\{0, \dots, k-1\}$; the elements from S are keys, and the boundary nodes stores informations associated to these keys



Tries

A trie that stores a collection of words S, $|S| = n$ from a k-sized alphabet has the following properties:

- any internal node has at most k children
- the tree has n external nodes
- the height of the tree is equal to the length of the longest word in S

Tries - The search operation

- Search for an element a in the structure t: follow the path described by the sequence $a[0], \ldots a[m-1]$

```
Function search(a, m, t)
begin
    $i \leftarrow 0$
    $p \leftarrow t$
    while ($p \neq NULL$ AND $i < m$) do
        $p \leftarrow p \rightarrow succ[a[i]]$
        $i \leftarrow i + 1$
    return $p$
end
```

- The worst-case time complexity: $O(m)$

Tries - insertion

- Inserting a word x in the structure t: it follows the path described by the sequence $x[0], \ldots x[m-1]$; add a new node for the components for which there are no nodes in t

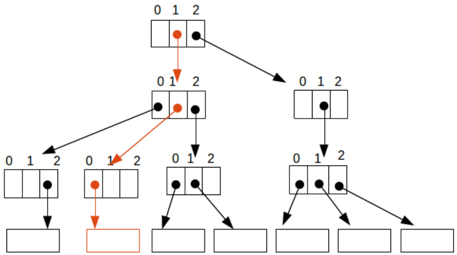


Figure: The insertion of key 110

Tries - deletion

- An element x to be removed is divided into:
  - a common prefix
  - a suffix that no longer belongs to any element
- Go through the path described by x and store the nodes in a stack
- Go back through this path and if all successors are *nil*, then remove the node

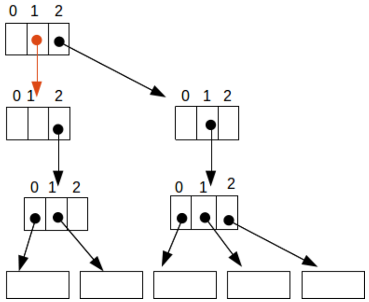
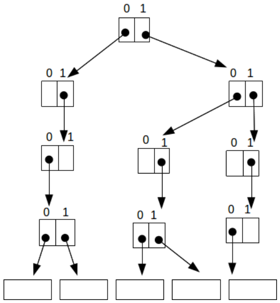


Figure: The deletion of key 102

Content

Compacting the chains

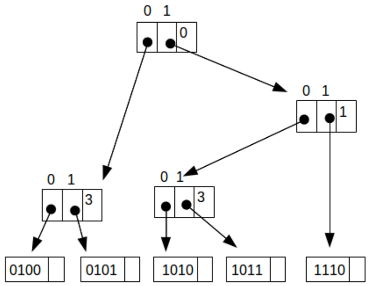
- A rare structure stores few keys; there are many chains of intermediate nodes with only one successor



- Removing these chains would lead to an improved space and time complexity; keep only the intermediate nodes with at least 2 successors

Compacting the chains

- The compact structure stores:
  - in the internal nodes, the position from which the keys with a common prefix differ
  - in the boundary nodes, the key.



Compacting the chains

A digital compacted trie which keeps a collection of words S, $|S| = n$, from an alphabet of size k has the following properties:

- each internal node has at least 2 children and at most k children
- the trie has n external nodes
- the number of nodes is $O(n)$

Compacting the chains - Operations

- **The search operation**: similarly with the uncompacted case, except that for each node the value from the stored position of the node is tested, and when it reaches a boundary node the key is tested
- **The insert operation** involves a search of the key, followed by the insertion of a new node to distinguish by the position on which the last, respectively the new node differs

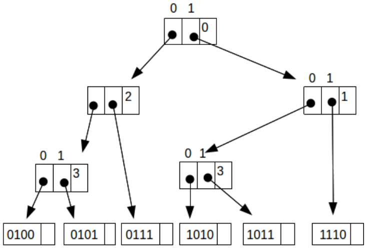


Figure: The insertion of the key 0111

Compacting the chains - Operations

- **The delete operation**: in a similar manner to the uncompacted case;
  - search for the node that stores the key and store the path in a stack;
  - remove the node with that key
  - go back through this path and remove the nodes with one successor

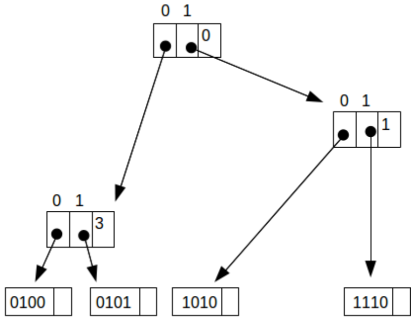


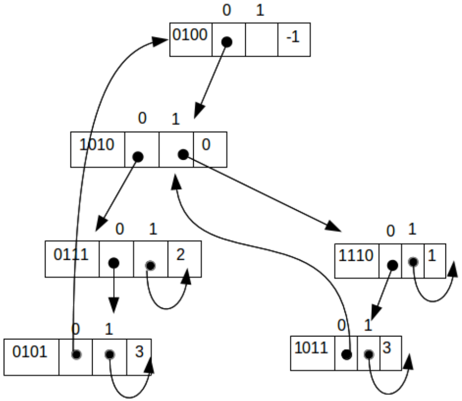
Figure: The deletion of the key 1011

Content

Patricia structures

- *Practical Algorithm to Retrieve Information Coded in Alphanumeric*
- A *digital binary tree* is a trie over the alphabet $\{0, 1\}$.
- The number of the boundary nodes is equal with the number of internal nodes plus one.
- If we add an extra internal node, we could store the keys in the internal nodes. The new node will be the root and will have one (left) child. The position from the root will be -1.

Patricia structures - Example



Patricia structures - Operations

- **The search operation** - may end when the current position is smaller then the last tested position
- **The insert operation**: search for the key in the structure, identify the first position *j* on which *x* and the key differs;

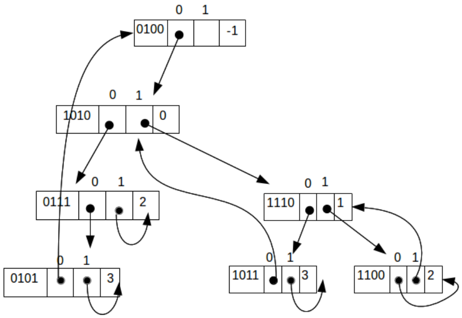


Figure: The insertion of the key 1100

Patricia structures - Operations

- **The delete operation** - example

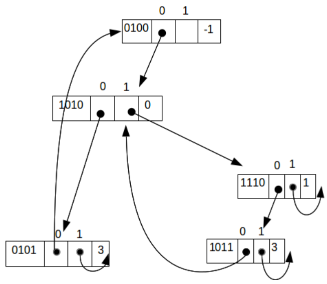


Figure: The deletion of the key 0111

Theorem:
Suppose that we create a Patricia structure starting from an empty structure by n insertions of keys randomly generated. Then the search operation requires $O(\log\ n)$ comparisons on average.

Tries - Applications

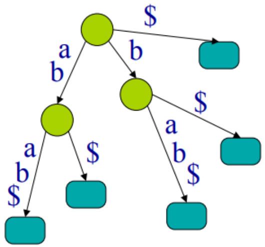
- Find all keys starting with a given prefix (the autocomplete function)
- Find the longest key that is a prefix of query string (example: to send packets, choose the IP address that is the longest prefix)
- Find all words that correspond to a given sequence of numbers
- Database search, P2P network search, querying XML files
- Computational biology

Content

Suffix trees

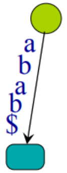
A *suffix tree* for a string s is a compressed trie of the suffixes of s.

Example: $s = abab$; the set of suffixes: $\{\$, b\$, ab\$, bab\$, abab\$\}$.

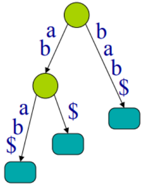


Construction of a suffix tree

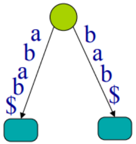
Add the longest suffix:



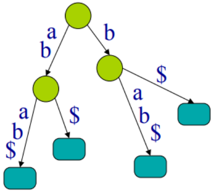
Add the suffix *ab*\$:



Add the suffix *bab*\$:

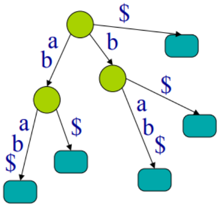


Add the suffix *b*\$:

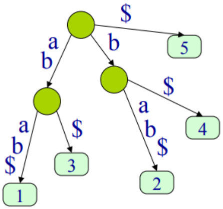


Construction of a suffix tree

Add the suffix $:



Label the leaf nodes (with the corresponding suffix index):



Time complexity: $O(n^2)$. Ukkonen's algorithm: $O(n)$.

Application: string matching

Check if a substring (*pattern*) $P(|P| = m)$ appears in a string $T(|T| = n)$.

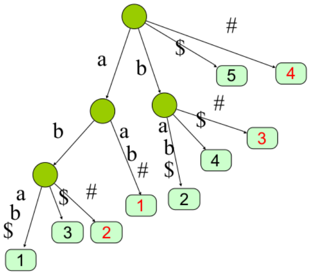
- build a suffix tree for T ($O(n)$)
- cross the tree according to the P substring
- each leaf in the identified subtree corresponds to an appearance (k appearances)

Time complexity: $O(n + k)$.

Application: the longest common substring

A *generalized suffix tree* for a set of strings S is a compressed trie of the suffixes $s \in S$.

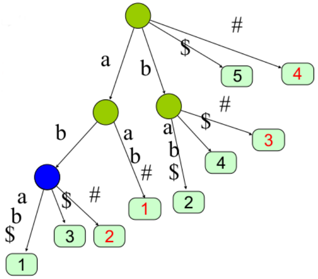
Example: $s_1 = abab$, $s_2 = aab$; $\{\$, b\$, ab\$, bab\$, abab\$\}$, $\{\#, b\#, ab\#, aab\#\}$.



Application: the longest common substring

Example: $s_1 = abab$, $s_2 = aab$; $\{\$, b\$, ab\$, bab\$, abab\$\}$, $\{\#, b\#, ab\#, aab\#\}$.

Any node that has \$ and $\#$ in his subtree corresponds to a common substring for s_1 and s_2. The longest common substring: the node that satisfies the property and has the longest substring.


a
b
$
#
5
4
b
a
b
#
1
a
b
$
#
3
4
2
a
b
$
$
#
1
3
2