Digital Trees

Letícia Rodrigues Bueno

Federal University of ABC (UFABC)

Digital Searching

Digital Searching

- **General problem of searching:** set of keys S and key x to search in S;

Digital Searching

- **General problem of searching:** set of keys S and key x to search in S;
- So far we assumed:

Digital Searching

- **General problem of searching:** set of keys S and key x to search in S;
- So far we assumed:
  - keys are indivisible elements;

Digital Searching

- **General problem of searching:** set of keys S and key x to search in S;
- So far we assumed:
  - keys are indivisible elements;
  - size of keys allows efficient storage in memory;

Digital Searching

- **General problem of searching:** set of keys S and key x to search in S;
- So far we assumed:
  - keys are indivisible elements;
  - size of keys allows efficient storage in memory;
  - keys have the same size;

Digital Searching

- **General problem of searching:** set of keys S and key x to search in S;
- So far we assumed:
  - keys are indivisible elements;
  - size of keys allows efficient storage in memory;
  - keys have the same size;
- **What if the search consists in <u>sentences</u> in a <u>book</u>?**

Digital Searching

- **General problem of searching:** set of keys S and key x to search in S;
- So far we assumed:
  - keys are indivisible elements;
  - size of keys allows efficient storage in memory;
  - keys have the same size;
- **What if the search consists in sentences in a book?**
- Then we can implement a **digital searching**;

Digital Searching

- **General problem of searching:** set of keys S and key x to search in S;
- So far we assumed:
  - keys are indivisible elements;
  - size of keys allows efficient storage in memory;
  - keys have the same size;
- **What if the search consists in <u>sentences</u> in a <u>book</u>?**
- Then we can implement a **digital searching**;
- Data structure for this case: **digital tree**;

Digital Trees or Tries

Digital Trees or Tries

- **Trie:** the term comes from “information re**TRIE**val”;

Digital Trees or Tries

- **Trie:** the term comes from "information re**TRIE**val";
- When comparing keys:

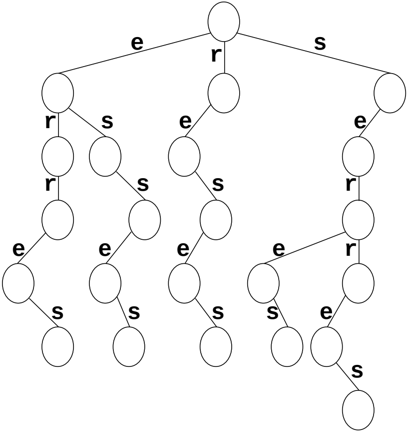
Digital Trees or Tries

- **Trie:** the term comes from “information re**TRIE**val”;
- When comparing keys:
  - it does **NOT** compare the desired key with the keys in the stored set;

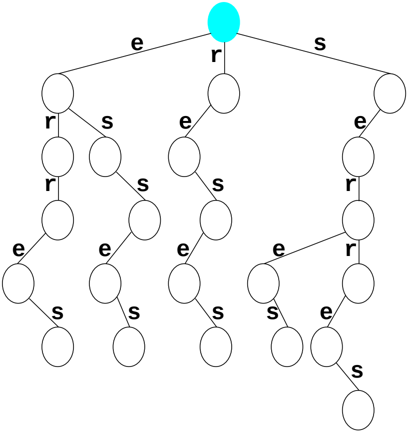
Digital Trees or Tries

- **Trie:** the term comes from “information re**TRIE**val”;
- When comparing keys:
  - it does **NOT** compare the desired key with the keys in the stored set;
  - it **DOES** compare digits of desired key one by one. Number of steps equals to the size of the key;

Example of Digital Tree

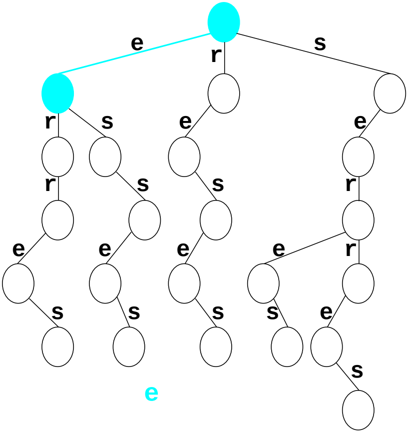


Example of Digital Tree



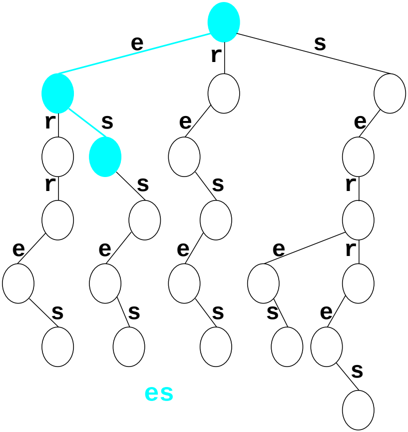
Alphabet:
$\{e, r, s\}$

Example of Digital Tree



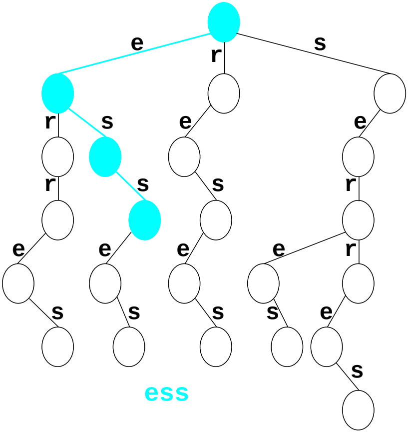
Alphabet:
$\{e, r, s\}$

Example of Digital Tree



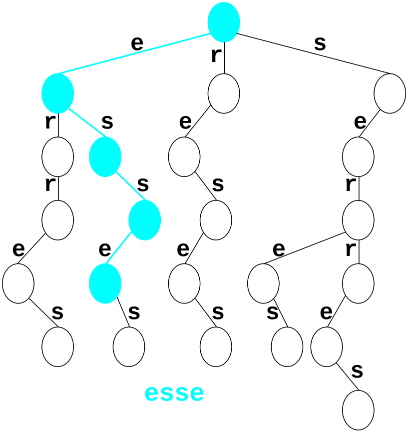
Alphabet:
$\{e, r, s\}$

Example of Digital Tree



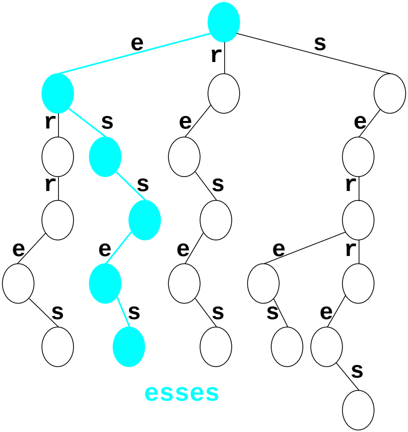
Alphabet:
$\{e, r, s\}$

Example of Digital Tree



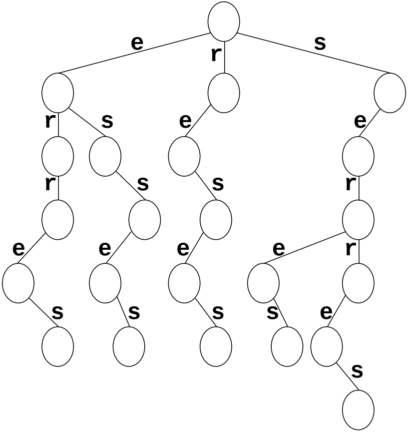
Alphabet:
$\{e, r, s\}$

Example of Digital Tree



Alphabet:
$\{e, r, s\}$

Example of Digital Tree



erre

erres

es

esse

esses

se

ser

serre

re

res

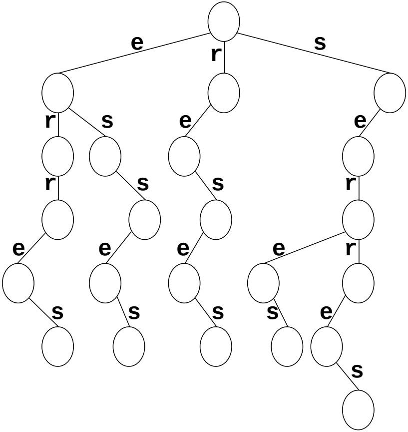
rese

reses

serres

seres

Example of Digital Tree



erre

erres

es

esse

esses

se

ser

serre

re

res

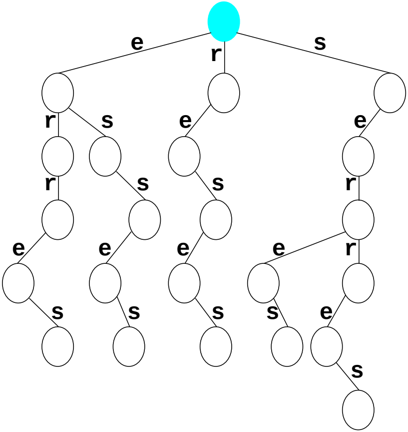
rese

reses

serres

seres

Example of Digital Tree



erre

erres

es

esse

esses

se

ser

serre

re

res

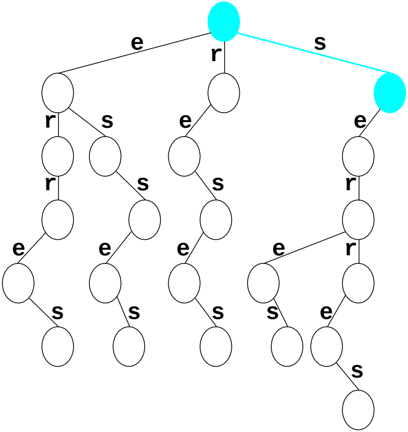
rese

reses

serres

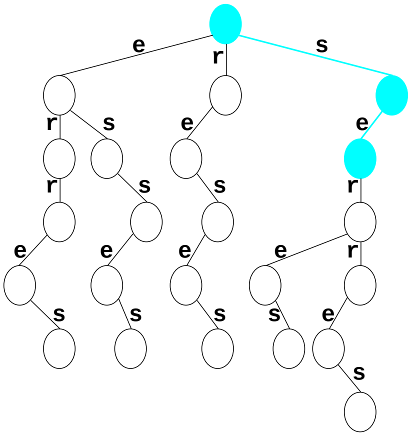
seres

Example of Digital Tree



erre
erres
es
esse
esses
se
ser
serre
re
res
rese
reses
serres
seres

Example of Digital Tree



erre

erres

es

esse

esses

se

ser

serre

re

res

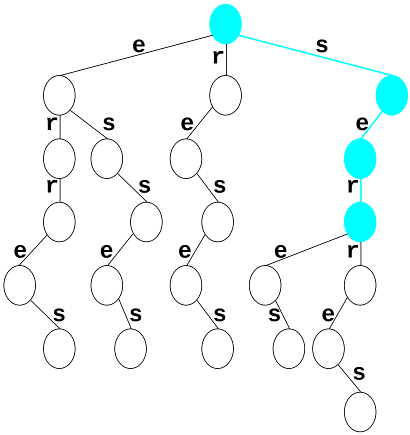
rese

reses

serres

seres

Example of Digital Tree



erre

erres

es

esse

esses

se

ser

serre

re

res

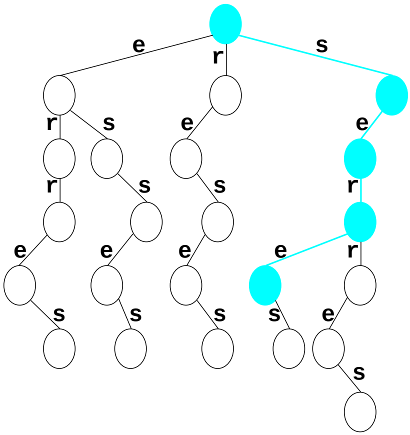
rese

reses

serres

seres

Example of Digital Tree



erre

erres

es

esse

esses

se

ser

serre

re

res

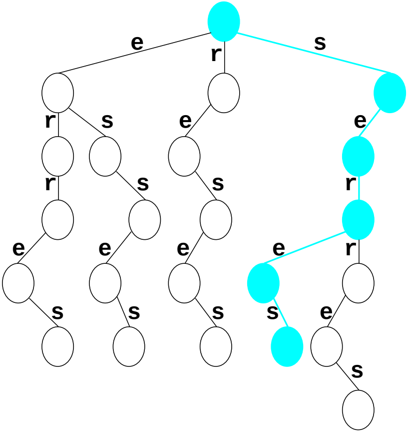
rese

reses

serres

seres

Example of Digital Tree



erre

erres

es

esse

esses

se

ser

serre

re

res

rese

reses

serres

seres

Digital Trees or Tries

Digital Trees or Tries

- Set of keys: $S = \{s_1, s_2, \ldots, s_n\}$;

Digital Trees or Tries

- Set of keys: $S = \{s_1, s_2, \ldots, s_n\}$;
- s_i is sequence of digits d_j;

Digital Trees or Tries

- Set of keys: $S = \{s_1, s_2, \ldots, s_n\}$;
- s_i is sequence of digits d_j;
- Alphabet of S: $d_1 < d_2 < \ldots < d_m$;

Digital Trees or Tries

- Set of keys: $S = \{s_1, s_2, \ldots, s_n\}$;
- s_i is sequence of digits d_j;
- Alphabet of S: $d_1 < d_2 < \ldots < d_m$;
- p first digits of the key: prefix;

Digital Trees or Tries

- Set of keys: $S = \{s_1, s_2, \ldots, s_n\}$;
- s_i is sequence of digits d_j;
- Alphabet of S: $d_1 < d_2 < \ldots < d_m$;
- p first digits of the key: prefix;
- Digital tree is a not empty m-ary tree T where:

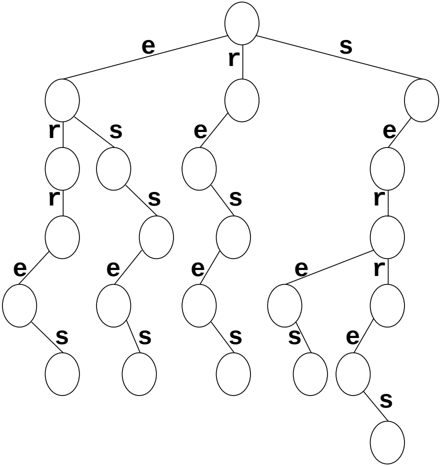
Digital Trees or Tries

- Set of keys: $S = \{s_1, s_2, \ldots, s_n\}$;
- s_i is sequence of digits d_j;
- Alphabet of S: $d_1 < d_2 < \ldots < d_m$;
- p first digits of the key: prefix;
- Digital tree is a not empty m-ary tree T where:
  1. Node v is jth child of its parent $\Rightarrow$ v corresponds digit d_j;

Digital Trees or Tries

- Set of keys: $S = \{s_1, s_2, \ldots, s_n\}$;
- s_i is sequence of digits d_j;
- Alphabet of S: $d_1 < d_2 < \ldots < d_m$;
- p first digits of the key: <u>prefix</u>;
- Digital tree is a not empty m-ary tree T where:
  1. Node v is jth child of its parent $\Rightarrow$ v corresponds digit d_j;
  2. Digits sequence from the root to a node corresponds to the prefix of some key in S.

Example of Digital Tree



Ternary tree;

Alphabet:
$\{e, r, s\} \therefore e < r < s$
$m = 3$

$S = \{erre, erres, es, esse, esses, se, ser, serre, re, res, rese, reses, serres, seres\}$

Digital Trees: search algorithm

digitalSearch(x, pt, l, a):
    if $l < k$ **then**
        let j be the position of $d(l + 1)$
            in the sorted alphabet
        if $pt.pont[j] \neq null$ **then**
            $pt \leftarrow pt.pont[j]$;
            $l \leftarrow l + 1$;
            digitalSearch(x, pt, l, a);
    else if $pt.terminal = true$ **then**
        $a \leftarrow 1$;

Digital Trees: search algorithm

```
digitalSearch(x, pt, l, a):
    if l < k then
        let j be the position of d(l + 1)
            in the sorted alphabet
        if pt.pont[j] ≠ null then
            pt ← pt.pont[j];
            l ← l + 1;
            digitalSearch(x, pt, l, a);
    else if pt.terminal = true then
        a ← 1;
```

Analysis of Complexity:

Digital Trees: search algorithm

1 *digitalSearch*(x, pt, l, a):
2   **if** $l < k$ **then**
3   let j be the position of $d(l+1)$
            in the sorted alphabet
4   **if** $pt.pont[j] \neq null$ **then**
5             $pt \leftarrow pt.pont[j]$;
6             $l \leftarrow l + 1$;
7   *digitalSearch*(x, pt, l, a);
8   **else if** $pt.terminal = true$ **then**
9         $a \leftarrow 1$;

Analysis of Complexity:

- **Line** 3: takes $O(\log m)$ with binary search;

Digital Trees: search algorithm

1 *digitalSearch*(x, pt, l, a):
2   **if** $l < k$ **then**
3   let j be the position of $d(l+1)$ in the sorted alphabet
4   **if** $pt.pont[j] \neq null$ **then**
5             $pt \leftarrow pt.pont[j]$;
6             $l \leftarrow l + 1$;
7   *digitalSearch*(x, pt, l, a);
8   **else if** $pt.terminal = true$ **then**
9         $a \leftarrow 1$;

Analysis of Complexity:

- **Line 3**: takes $O(\log m)$ with binary search;
- **Total complexity**: $O(k \cdot \log m)$;

Digital Trees: search algorithm

1 *digitalSearch*(x, pt, l, a):
2   **if** $l < k$ **then**
3   let j be the position of $d(l+1)$
              in the sorted alphabet
4   **if** $pt.pont[j] \neq null$ **then**
5             $pt \leftarrow pt.pont[j]$;
6             $l \leftarrow l + 1$;
7   *digitalSearch*(x, pt, l, a);
8   **else if** $pt.terminal = true$ **then**
9         $a \leftarrow 1$;

Analysis of Complexity:

- **Line** 3: takes $O(\log m)$ with binary search;
- **Total complexity**: $O(k \cdot \log m)$;
- Binary representation of digits gives complexity: $O(k)$;

x is the searched key with k digits, so k is size of the key.

First call:

$l \leftarrow 0$; $a \leftarrow 0$; *digitalSearch*$(x, root, l, a)$

Digital Trees: insertion algorithm

digitalInsert(x, pt):
    $pt \leftarrow ptroot$; $l \leftarrow a \leftarrow 0$;
    digitalSearch(x, pt, l, a);
    if $a = 0$ **then**
        for $h = l + 1, \ldots, k$ **do**
            let j be the position of
                $d(h)$ in the alphabet;
            new(ptz);
            for $i = 1, \ldots, m$ **do**
                $ptz.pont[i] \leftarrow null$;
            $pt.pont[j] \leftarrow ptz$;
            $ptz.terminal \leftarrow false$;
            $pt \leftarrow ptz$;
        $pt.terminal \leftarrow true$;
    else “Invalid Insertion”;

Digital Trees: insertion algorithm

```
digitalInsert(x, pt):
    pt ← ptroot; l ← a ← 0;
    digitalSearch(x, pt, l, a);
    if a = 0 then
        for h = l + 1, ..., k do
            let j be the position of
               d(h) in the alphabet;
            new(ptz);
            for i = 1, ..., m do
                ptz.pont[i] ← null;
            pt.pont[j] ← ptz;
            ptz.terminal ← false;
            pt ← ptz;
        pt.terminal ← true;
    else "Invalid Insertion";
```

Analysis of complexity:

Digital Trees: insertion algorithm

```
digitalInsert(x, pt):
    pt ← ptroot; l ← a ← 0;
    digitalSearch(x, pt, l, a);
    if a = 0 then
        for h = l + 1, ..., k do
            let j be the position of
                d(h) in the alphabet;
            new(ptz);
            for i = 1, ..., m do
                ptz.pont[i] ← null;
            pt.pont[j] ← ptz;
            ptz.terminal ← false;
            pt ← ptz;
        pt.terminal ← true;
    else "Invalid Insertion";
```

Analysis of complexity:

- Let $k_1 + k_2 = k$;

Digital Trees: insertion algorithm

1 *digitalInsert*(x, pt):
2     $pt \leftarrow ptroot$; $l \leftarrow a \leftarrow 0$;
3   *digitalSearch*(x, pt, l, a);
4   **if** $a = 0$ **then**
5   **for** $h = l + 1, \ldots, k$ **do**
6   let j be the position of $d(h)$ in the alphabet;
7   *new*(ptz);
8   **for** $i = 1, \ldots, m$ **do**
9                 $ptz.pont[i] \leftarrow null$;
10             $pt.pont[j] \leftarrow ptz$;
11             $ptz.terminal \leftarrow false$;
12             $pt \leftarrow ptz$;
13         $pt.terminal \leftarrow true$;
14   **else** “Invalid Insertion”;

Analysis of complexity:

- Let $k_1 + k_2 = k$;
- **Line 3**: takes $O(k_1 \cdot \log m)$;

Digital Trees: insertion algorithm

```
1  digitalInsert(x, pt):
2     pt ← ptroot; l ← a ← 0;
3     digitalSearch(x, pt, l, a);
4     if a = 0 then
5        for h = l + 1, ..., k do
6           let j be the position of
               d(h) in the alphabet;
7           new(ptz);
8           for i = 1, ..., m do
9              ptz.pont[i] ← null;
10          pt.pont[j] ← ptz;
11          ptz.terminal ← false;
12          pt ← ptz;
13       pt.terminal ← true;
14    else "Invalid Insertion";
```

Analysis of complexity:

- Let $k_1 + k_2 = k$;
- **Line** 3: takes $O(k_1 \cdot \log m)$;
- **Line** 8: executes m times. Therefore, insertion takes $O(k_2 \cdot m)$;

Digital Trees: insertion algorithm

```
1   digitalInsert(x, pt):
2     pt ← ptroot; l ← a ← 0;
3     digitalSearch(x, pt, l, a);
4     if a = 0 then
5       for h = l + 1, ..., k do
6         let j be the position of
              d(h) in the alphabet;
7         new(ptz);
8         for i = 1, ..., m do
9           ptz.pont[i] ← null;
10        pt.pont[j] ← ptz;
11        ptz.terminal ← false;
12        pt ← ptz;
13      pt.terminal ← true;
14    else "Invalid Insertion";
```

Analysis of complexity:

- Let $k_1 + k_2 = k$;
- **Line 3**: takes $O(k_1 \cdot \log m)$;
- **Line 8**: executes m times. Therefore, insertion takes $O(k_2 \cdot m)$;
- **Total complexity**: $O(k_1 \log m + k_2 m)$;

x is key to insert. **First call:** *digitalInsert*$(x, ptroot)$

Digital Trees: complexity of searching

Digital Trees: complexity of searching

- Different from other classic searching methods because:

Digital Trees: complexity of searching

- Different from other classic searching methods because:
  - no dependence on total number of keys or file size;

Digital Trees: complexity of searching

- Different from other classic searching methods because:
  - no dependence on total number of keys or file size;
  - no dependence on size of searched key or alphabet;

Digital Trees: complexity of searching

- Different from other classic searching methods because:
  - no dependence on total number of keys or file size;
  - no dependence on size of searched key or alphabet;
- **Alternative implementation**: circular linked list for pointers to avoid too many null pointers (it optimizes memory);

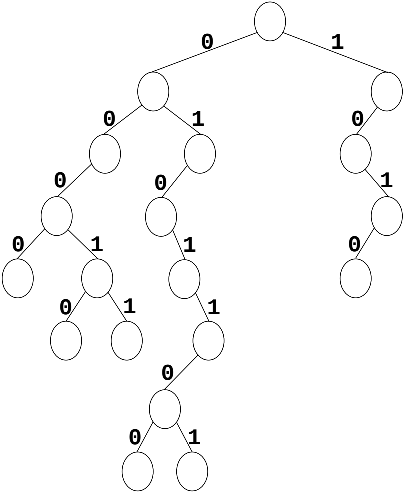
Digital Trees: complexity of searching

- Different from other classic searching methods because:
  - no dependence on total number of keys or file size;
  - no dependence on size of searched key or alphabet;
- **Alternative implementation**: circular linked list for pointers to avoid too many null pointers (it optimizes memory);
- Trie is as efficient as more keys with common prefixes;

Digital Trees: complexity of searching

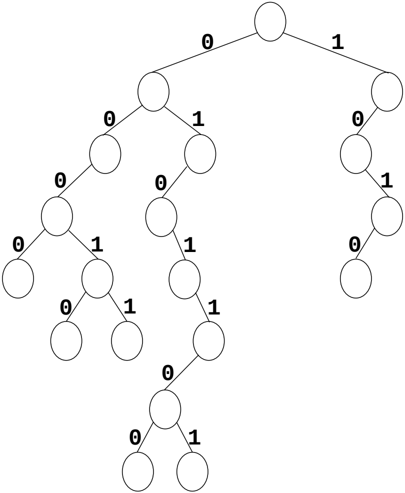
- Different from other classic searching methods because:
  - no dependence on total number of keys or file size;
  - no dependence on size of searched key or alphabet;
- **Alternative implementation**: circular linked list for pointers to avoid too many null pointers (it optimizes memory);
- Trie is as efficient as more keys with common prefixes;
- Trie with many zig-zags is almost always inefficient;

Binary Digital Trees



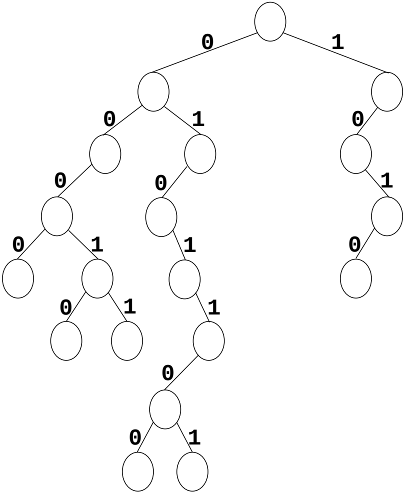
- Binary digital tree with alphabet $\{0, 1\}$;

Binary Digital Trees



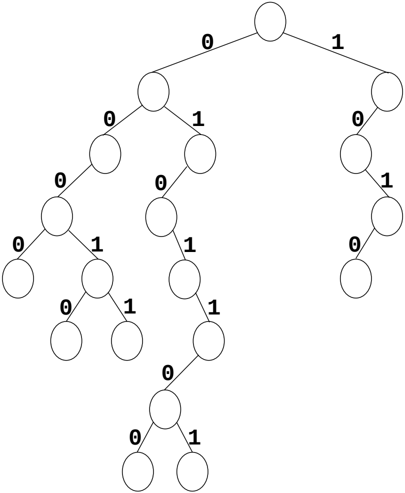
- Binary digital tree with alphabet $\{0, 1\}$;
- Keys are binary sequence;

Binary Digital Trees



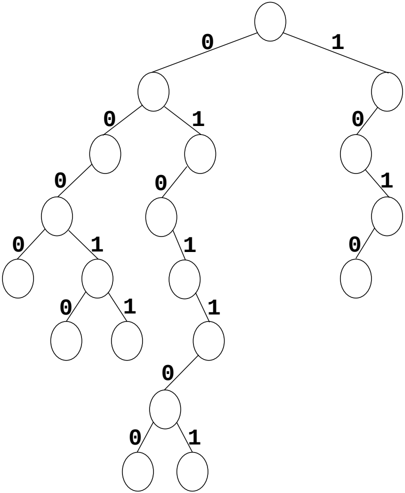
- Binary digital tree with alphabet $\{0, 1\}$;
- Keys are binary sequence;
- Left child: 0;

Binary Digital Trees



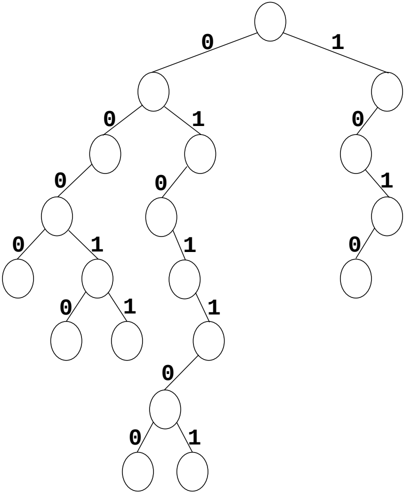
- Binary digital tree with alphabet $\{0, 1\}$;
- Keys are binary sequence;
- Left child: 0;
- Right child: 1;

Binary Digital Trees



- Binary digital tree with alphabet $\{0, 1\}$;
- Keys are binary sequence;
- Left child: 0;
- Right child: 1;
- $S = \{00, 0000, 00010, 00011, 0101100, 0101101, 10, 101, 1010\}$

Binary Digital Trees



- Binary digital tree with alphabet $\{0, 1\}$;
- Keys are binary sequence;
- Left child: 0;
- Right child: 1;
- $S = \{00, 0000, 00010, 00011, 0101100, 0101101, 10, 101, 1010\}$
- Larger use of digital trees is binary case;

Binary Digital Trees

Binary Digital Trees

- Binary keys/codes are more used in computer science;

Binary Digital Trees

- Binary keys/codes are more used in computer science;
- Number of empty pointers is smaller;

Binary Digital Trees

- Binary keys/codes are more used in computer science;
- Number of empty pointers is smaller;
- **Prefix Binary Tree**:

Binary Digital Trees

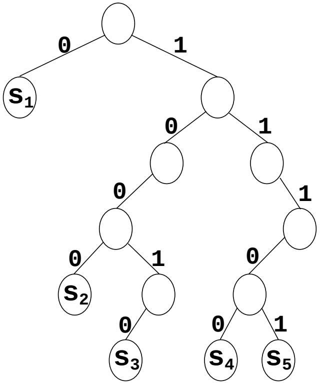
- Binary keys/codes are more used in computer science;
- Number of empty pointers is smaller;
- **Prefix Binary Tree**:
    1. no code is prefix of another one;

Binary Digital Trees

- Binary keys/codes are more used in computer science;
- Number of empty pointers is smaller;
- **Prefix Binary Tree**:
  1. no code is prefix of another one;
  2. keys are represented by leaves;

Binary Digital Trees

- Binary keys/codes are more used in computer science;
- Number of empty pointers is smaller;
- **Prefix Binary Tree**:
    1. no code is prefix of another one;
    2. keys are represented by leaves;



Patricia Trees: Introduction

- **PATRICIA:** acronym of **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric;

Patricia Trees: Introduction

- **PATRICIA:** acronym of **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric;
- **Creator:** Donald Morrison, in 1968;

Patricia Trees: Introduction

- **PATRICIA:** acronym of **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric;
- **Creator:** Donald Morrison, in 1968;
- **Patricia Tree:** binary prefix digital tree;

Patricia Trees: Introduction

- **PATRICIA:** acronym of **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric;
- **Creator:** Donald Morrison, in 1968;
- **Patricia Tree:** binary prefix digital tree;
- Strictly binary;

Patricia Trees: Introduction

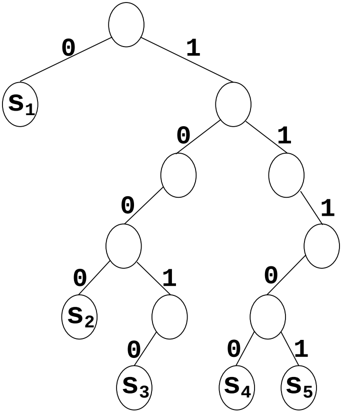
- **PATRICIA:** acronym of **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric;
- **Creator:** Donald Morrison, in 1968;
- **Patricia Tree:** binary prefix digital tree;
- Strictly binary;
- Sequence of nodes with one child only are compressed in one single node;

Patricia Trees: Introduction

- **PATRICIA:** acronym of **P**ractical **A**lgorithm **T**o **R**etrieve **I**nformation **C**oded **I**n **A**lphanumeric;
- **Creator:** Donald Morrison, in 1968;
- **Patricia Tree:** binary prefix digital tree;
- Strictly binary;
- Sequence of nodes with one child only are compressed in one single node;
- No keys are prefix of any other keys.

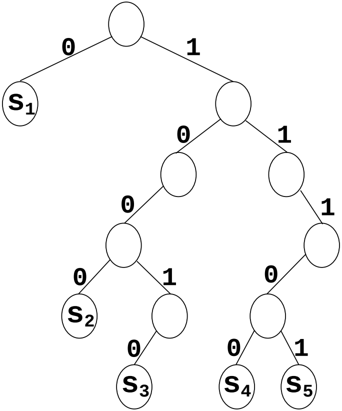
Example of Patricia Tree

Example of Patricia Tree

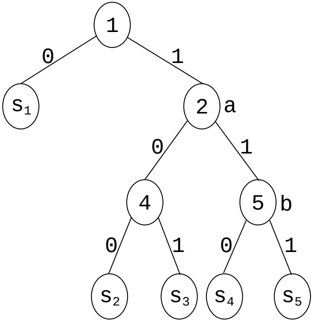


(c) Binary prefix digital tree

Example of Patricia Tree

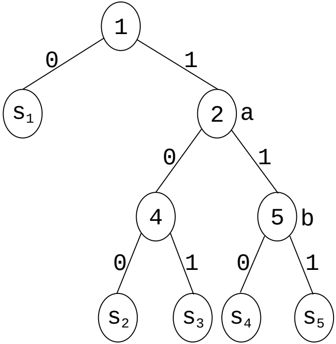


(e) Binary prefix digital tree

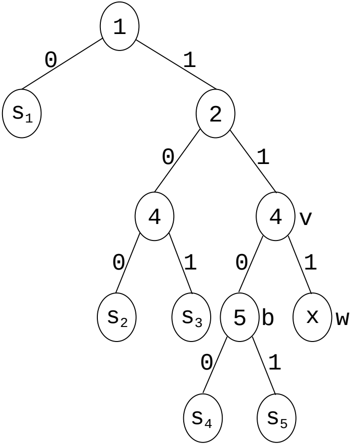


(f) Patricia tree

Patricia Tree: Insertion



Patricia Tree: Insertion



Patricia Tree: insertion algorithm

```
patriciaSearch(x, pt, a):
    if pt.left = null then a ← 1;
    else
        if k < pt.r then a ← 2;
        else
            if d[pt.r] = 0 then
                pt ← pt.left;
                patriciaSearch(x, pt, a);
            else pt ← pt.right
                patriciaSearch(x, pt, a);
```

Exercises

1. Write the procedure for insertion of a key in a Trie.

Exercises

1. Write the procedure for insertion of a key in a Trie.
2. Write the procedure for deletion of a key in a Trie.

Bibliography

SZWARCFITER, J. L. and MARKENZON, L. Estruturas de Dados e seus Algoritmos, LTC, 1994. (in Portuguese)

Questions?