Estruturas de Dados

Árvores Digitais de Busca, Tries e Patricia Tries

Aula 10

Prof. Felipe A. Louza

UFU

Roteiro

Roteiro

Busca Radix

Veremos outro tipo de árvores binárias

- Mas **não** são árvores binárias de busca...
- Usadas quando a chave pode ser representada em poucos bits
- E extrair o bit é uma operação rápida
  - `int`, `unsigned`, `char`, etc...

Busca Radix

Veremos outro tipo de árvores binárias

- Mas **não** são árvores binárias de busca...
- Usadas quando a chave pode ser representada em poucos bits
- E extrair o bit é uma operação rápida
  - `int`, `unsigned`, `char`, etc...
- Cada chave será *inserida* na árvore de acordo com a sua representação binária, ex. $11_2 = 1011$

Busca Radix

Veremos outro tipo de árvores binárias

- Mas **não** são árvores binárias de busca...
- Usadas quando a chave pode ser representada em poucos bits
- E extrair o bit é uma operação rápida
  - `int`, `unsigned`, `char`, etc...
- Cada chave será *inserida* na árvore de acordo com a sua representação binária, ex. $11_2 = 1011$

Se a chave tiver no máximo b bits

- a altura da árvore h será pequena: $\mathrm{O}(b)$ $\leftarrow$ ex: $b = 32$ bits, $b = \lg(n)$
- mas poderá guardar muitas chaves: $n = 2^b$ $\leftarrow n = 2^{32} \approx 4$ bilhões

Uma árvore com altura baixa, mas com muitos nós.

Busca Radix

Veremos outro tipo de árvores binárias

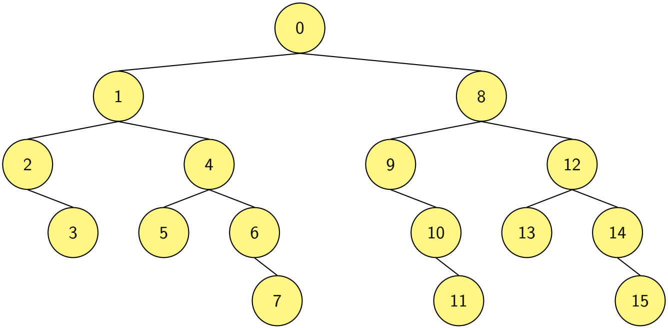
- Mas **não** são árvores binárias de busca...
- Usadas quando a chave pode ser representada em poucos bits
- E extrair o bit é uma operação rápida
  - `int`, `unsigned`, `char`, etc...
- Cada chave será *inserida* na árvore de acordo com a sua representação binária, ex. $11_2 = 1011$

Se a chave tiver no máximo b bits

- a altura da árvore h será pequena: $\mathrm{O}(b)$ $\leftarrow$ ex: $b = 32$ bits, $b = \lg(n)$
- mas poderá guardar muitas chaves: $n = 2^b$ $\leftarrow n = 2^{32} \approx 4$ bilhões

Operações de *busca/inserção/remoção* em tempo $h = \lg(n)$

Árvores Digitais de Busca

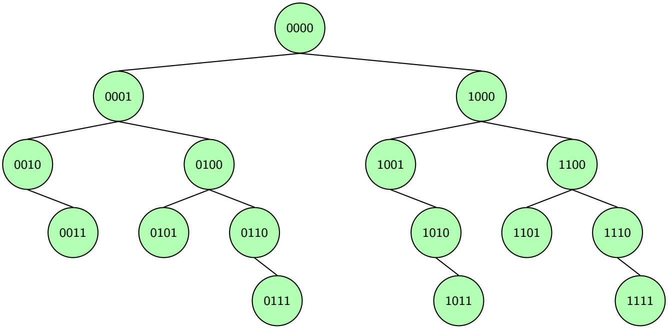


Note que **não** é uma **ABB**!

- Por exemplo, 2 e 3 são maiores do que 1
- Mas estão na subárvore esquerda de 1

Digital ≈ digitos.

Árvores Digitais de Busca

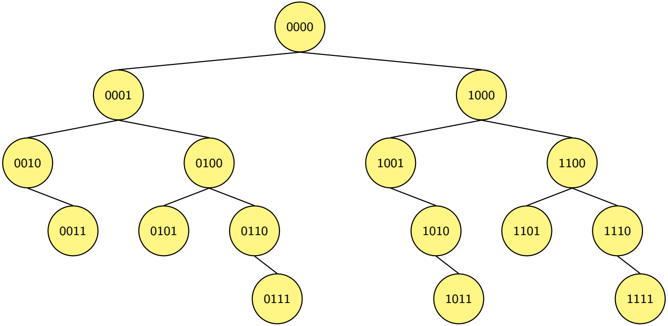


As subárvores de um nó de nível k são tais que:

- chaves com k-ésimo bit zero ficam na subárvore esquerda
- chaves com k-ésimo bit um ficam na subárvore direita

Vamos olhar para a representação binária das chaves.

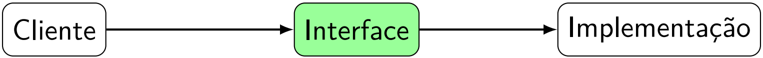
Árvores Digitais de Busca



Vamos assumir que as chaves são distintas (`primary key`)

- as chaves estão em "algum lugar" no caminho especificado pelos seus bits

TAD - Interface



digital.h

```c
#ifndef ADB_H
#define ADB_H

//Dados
typedef struct No {
  unsigned chave;
  struct No *esq, *dir;
} No;

#define bits_na_chave 32 //4
```

```c
//Funções
No*  criar_arvore();
void destruir_arvore(No **p);

No*  inserir(No* p, unsigned chave);
No*  buscar(No* p, unsigned x);

void imprimir(No* p, int level);

#endif
```

- Implementação para chaves do tipo `unsigned`

Árvores Digitais de Busca - Implementação

Precisamos acessar o k-ésimo bit (da esquerda para a direita):

```c
unsigned bit(unsigned chave, int k) {
  return chave >> (bits_na_chave - 1 - k) & 1;
}
```

Exemplo:

- $2434793472_2 = 10010001\ \ 00100000\ \ 00000000\ \ 00000000$

Árvores Digitais de Busca - Implementação

Precisamos acessar o k-ésimo bit (da esquerda para a direita):

```
unsigned bit(unsigned chave, int k) {
  return chave >> (bits_na_chave - 1 - k) & 1;
}
```

Exemplo:

- $2434793472_2 = 10010001\ \ 00100000\ \ 00000000\ \ 00000000$
- $2434793472_2 >> (32 - 1 - 3) = 9_2$

Árvores Digitais de Busca - Implementação

Precisamos acessar o *k*-ésimo bit (da esquerda para a direita):

```
unsigned bit(unsigned chave, int k) {
  return chave >> (bits_na_chave - 1 - k) & 1;
}
```

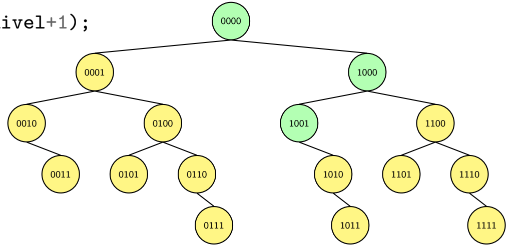
Exemplo:

- $2434793472_2 = 10010001 \;\; 00100000 \;\; 00000000 \;\; 00000000$
- $2434793472_2 >> (32 - 1 - 3) = 9_2$
- $9_2 = 00000000 \;\; 00000000 \;\; 00000000 \;\; 00001001 \;\; \& \;\; 1 = 1$

Árvores Digitais de Busca - Busca

```c
No* buscar_rec(No *p, unsigned x, int nivel) {//precisamos saber o nível
  if (p == NULL)
    return NULL;
  if (x == p->chave)
    return p; //encontrou o valor!
  if (bit(x, nivel) == 0)
    return buscar_rec(p->esq, x, nivel+1);
  else
    return buscar_rec(p->dir, x, nivel+1);
}

No* buscar(No *p, unsigned x) {
  return buscar_rec(p, x, 0);
}
```

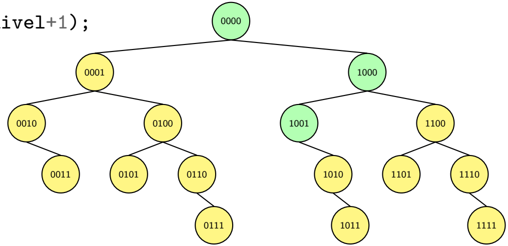


Exemplo: $9_b = 1001$.

Árvores Digitais de Busca - Busca

```c
No* buscar_rec(No *p, unsigned x, int nivel) {//precisamos saber o nível
  if (p == NULL)
    return NULL;
  if (x == p->chave)
    return p; //encontrou o valor!
  if (bit(x, nivel) == 0)
    return buscar_rec(p->esq, x, nivel+1);
  else
    return buscar_rec(p->dir, x, nivel+1);
}

No* buscar(No *p, unsigned x) {
  return buscar_rec(p, x, 0);
}
```



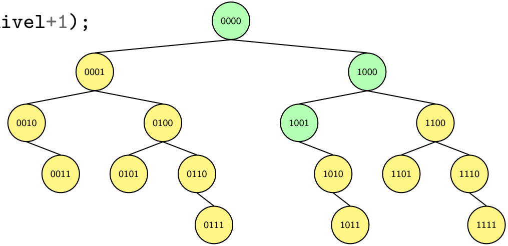
No pior caso, leva tempo $\mathrm{O}(b)$

- b é o número de bits das chaves
- Podemos guardar até $n = 2^b$ números na árvore, $b = \lceil \lg(n) \rceil$

Árvores Digitais de Busca - Busca

```c
No* buscar_rec(No *p, unsigned x, int nivel) {//precisamos saber o nível
  if (p == NULL)
    return NULL;
  if (x == p->chave)
    return p; //encontrou o valor!
  if (bit(x, nivel) == 0)
    return buscar_rec(p->esq, x, nivel+1);
  else
    return buscar_rec(p->dir, x, nivel+1);
}

No* buscar(No *p, unsigned x) {
  return buscar_rec(p, x, 0);
}
```



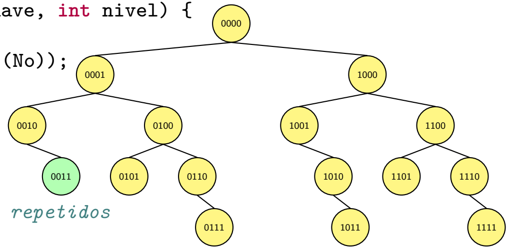
No pior caso, leva tempo $\mathrm{O}(b)$

- b é o número de bits das chaves
- Podemos guardar até $n = 2^b$ números na árvore, $b = \lceil \lg(n) \rceil$

Se $n \ll 2^b$, e as chaves forem aleatórias, tempo esperado também é $\mathrm{O}(\lg n)$.

```c
No* inserir_rec(No *p, unsigned chave, int nivel) {
  if (p == NULL) {
    No* novo = (No*) malloc(sizeof(No));
    novo->esq = novo->dir = NULL;
    novo->chave = chave;
    return novo;
  }
  if (chave == p->chave)
    return p; //não insere valores repetidos
  if (bit(chave, nivel) == 0)
    p->esq = inserir_rec(p->esq, chave, nivel+1);
  else
    p->dir = inserir_rec(p->dir, chave, nivel+1);
  return p;
}

No* inserir(No *p, unsigned chave) {
  return inserir_rec(p, chave, 0);
}
```

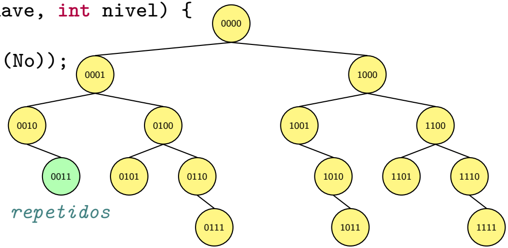


Exemplo: $3_b = 0011$.

Árvores Digitais de Busca - Inserção

```
No* inserir_rec(No *p, unsigned chave, int nivel) {
  if (p == NULL) {
    No* novo = (No*) malloc(sizeof(No));
    novo->esq = novo->dir = NULL;
    novo->chave = chave;
    return novo;
  }
  if (chave == p->chave)
    return p; //não insere valores repetidos
  if (bit(chave, nivel) == 0)
    p->esq = inserir_rec(p->esq, chave, nivel+1);
  else
    p->dir = inserir_rec(p->dir, chave, nivel+1);
  return p;
}

No* inserir(No *p, unsigned chave) {
  return inserir_rec(p, chave, 0);
}
```

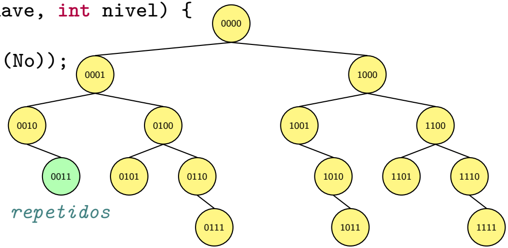


No pior caso, leva tempo $\mathrm{O}(b) = \mathrm{O}(\lg(n))$

Árvores Digitais de Busca - Inserção

```c
No* inserir_rec(No *p, unsigned chave, int nivel) {
  if (p == NULL) {
    No* novo = (No*) malloc(sizeof(No));
    novo->esq = novo->dir = NULL;
    novo->chave = chave;
    return novo;
  }
  if (chave == p->chave)
    return p; //não insere valores repetidos
  if (bit(chave, nivel) == 0)
    p->esq = inserir_rec(p->esq, chave, nivel+1);
  else
    p->dir = inserir_rec(p->dir, chave, nivel+1);
  return p;
}

No* inserir(No *p, unsigned chave) {
  return inserir_rec(p, chave, 0);
}
```



No pior caso, leva tempo $\mathrm{O}(b) = \mathrm{O}(\lg(n))$

A árvore pode ser diferente dependendo da ordem em que inserimos. Ex: 3 e 2.

Cliente

exemplo1.c

```c
#include <stdio.h>
#include "digital.h"
```

```c
int main() { //modificamos bits_na_chave para 4
  int i;
  No *T = criar_arvore();
  for (i = 0; i < 16; i++) {
    if(i==2) continue;
    T = inserir(T, i);
  }
  T = inserir(T, 2);
  imprimir(T, 0);
  destruir_arvore(&T);
  return 0;
}
```

Makefile

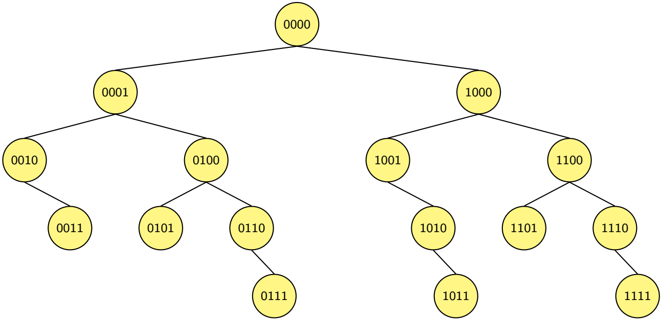
Vamos usar o Makefile para compilar:

```
exemplo1: exemplo1.c digital.o
  gcc $^ -o $@
```

Vamos executar:

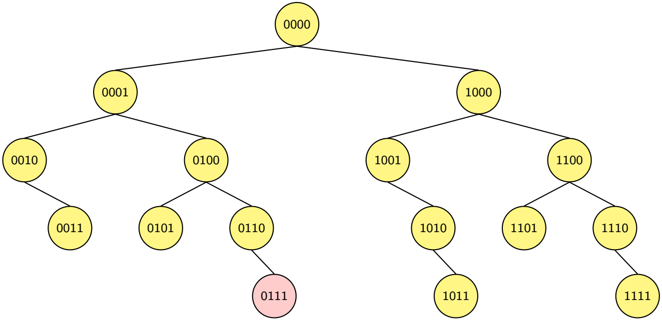
```
$ ./exemplo1
----1111        (15)
---1110         (14)
--1100          (12)
---1101         (13)
-1000           (8)
----1011        (11)
---1010         (10)
--1001          (9)
0000            (0)
----0111        (7)
---0110         (6)
--0100          (4)
---0101         (5)
-0001           (1)
---0010         (2)
--0011          (3)
```

Árvores Digitais de Busca - Remoção



Para remover:

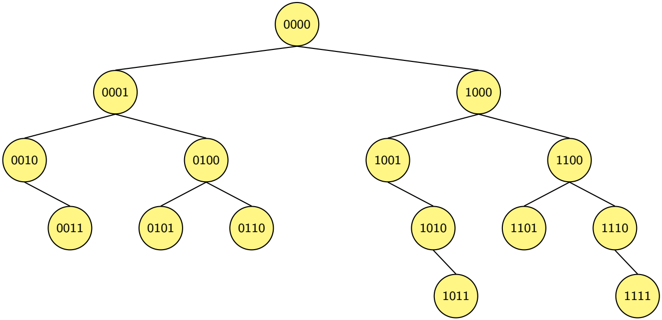
Árvores Digitais de Busca - Remoção



Para remover:

- uma folha, basta apagá-la

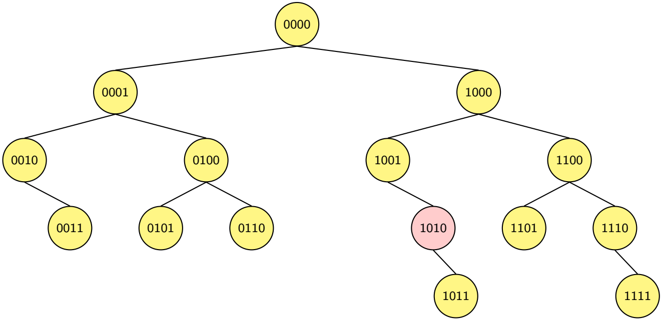
Árvores Digitais de Busca - Remoção



Para remover:

- uma folha, basta apagá-la

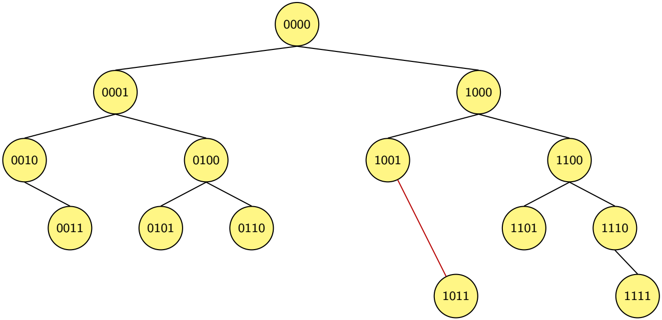
Árvores Digitais de Busca - Remoção



Para remover:

- uma folha, basta apagá-la
- um nó com um filho, basta fazer o pai apontar para o neto

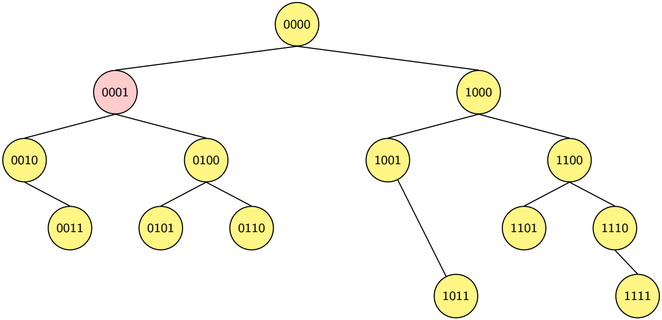
Árvores Digitais de Busca - Remoção



Para remover:

- uma folha, basta apagá-la
- um nó com um filho, basta fazer o pai apontar para o neto

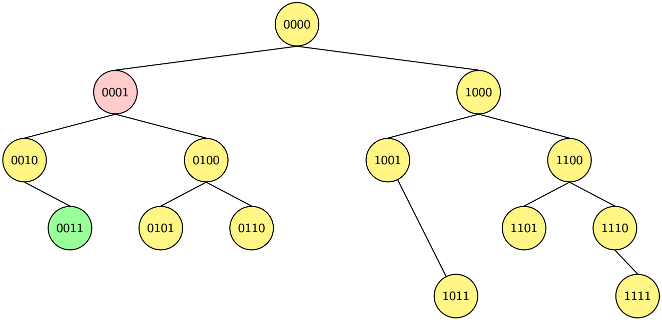
Árvores Digitais de Busca - Remoção



Para remover:

- uma folha, basta apagá-la
- um nó com um filho, basta fazer o pai apontar para o neto
- um nó com dois filhos

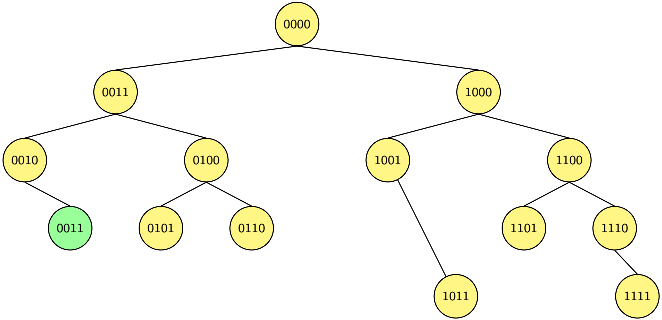
Árvores Digitais de Busca - Remoção



Para remover:

- uma folha, basta apagá-la
- um nó com um filho, basta fazer o pai apontar para o neto
- um nó com dois filhos
  - copie um descendente qualquer por cima do nó (compartilha *k* bits)

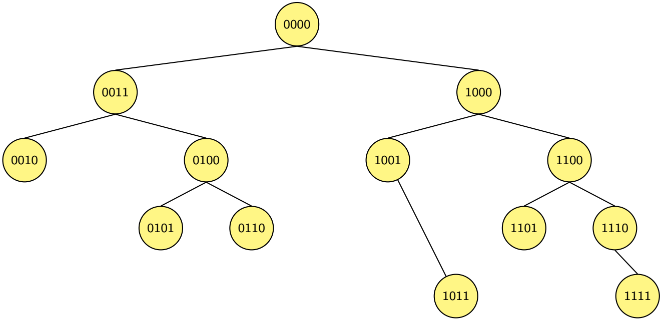
Árvores Digitais de Busca - Remoção



Para remover:

- uma folha, basta apagá-la
- um nó com um filho, basta fazer o pai apontar para o neto
- um nó com dois filhos
  - copie um descendente qualquer por cima do nó (compartilha k bits)
  - apague o descendente

Árvores Digitais de Busca - Remoção



Para remover:

- uma folha, basta apagá-la
- um nó com um filho, basta fazer o pai apontar para o neto
- um nó com dois filhos
  - copie um descendente qualquer por cima do nó (compartilha *k* bits)
  - apague o descendente

Árvores Digitais de Busca - Conclusões

Árvores Digitais de Busca são **interessantes** em muitas aplicações

Custo computacional próximo do ótimo:

- Para chaves de *32-bits*, teremos no máximo $\lg(2^{32}) = 32$ comparações para *busca/inserção/remoção*.

Com $b = 32$, podemos guardar $n \approx 4$ bilhões e chaves com $h = 32 = \lg(n)$.

Árvores Digitais de Busca - Conclusões

Árvores Digitais de Busca são **interessantes** em muitas aplicações

Custo computacional próximo do ótimo:

- Para chaves de *32-bits*, teremos no máximo $\lg(2^{32}) = 32$ comparações para *busca/inserção/remoção*.
- Implementação muito mais simples do que uma **ABB balanceada**.

Mais simples, por exemplo, do que uma árvore **rubro-negra**.

Árvores Digitais de Busca - Conclusões

Árvores Digitais de Busca são **interessantes** em muitas aplicações

Custo computacional próximo do ótimo:

- Para chaves de *32-bits*, teremos no máximo $\lg(2^{32}) = 32$ comparações para *busca/inserção/remoção*.
- Implementação muito mais simples do que uma **ABB balanceada**.

Principal limitação:

- As chaves **não** estão ordenadas
- Algumas operações que envolvem a ordem ficam caras
  - imprimir ordenado, sucessor e antecessor, por exemplo

Roteiro

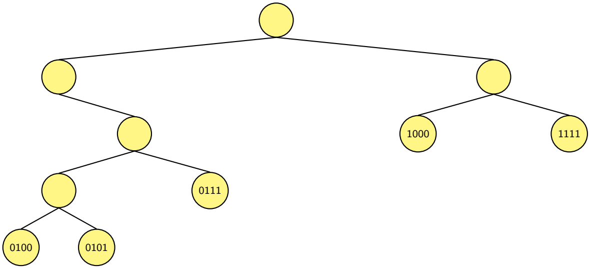
Tries

Tries: utilizam uma ideia parecida

- Mas conseguimos usar a informação da ordem das chaves
- Vem da palavra retrieval
  - Pronunciamos "trái" ao invés de tree

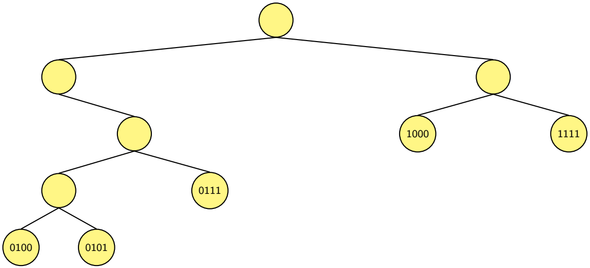
Vamos assumir que as chaves tem tamanho fixo de bits, mas Tries também funcionam para chaves com tamanho variável (contanto que não seja prefixo de outra).

Tries - Busca



Guardamos a informação apenas nas folhas:

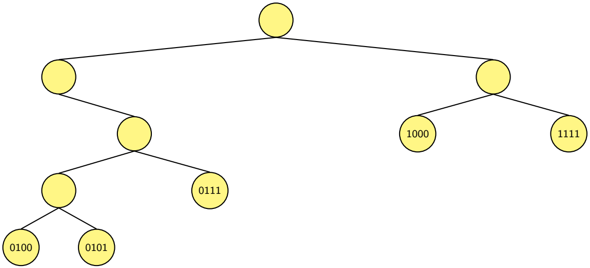
Tries - Busca



Guardamos a informação apenas nas folhas:

- para buscar basta ir para a esquerda ou para a direita
  - dependendo do valor do k-ésimo bit

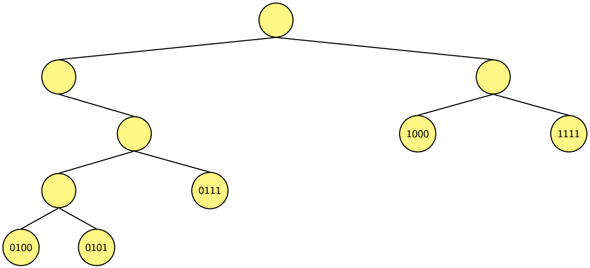
Tries - Busca



Guardamos a informação apenas nas folhas:

- para buscar basta ir para a esquerda ou para a direita
  - dependendo do valor do *k*-ésimo bit
- Ao chegar em `NULL`, a chave **não** está na árvore

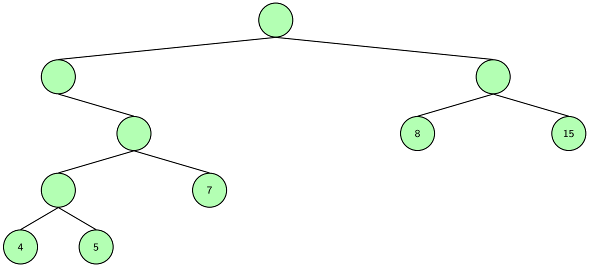
Tries - Busca



Guardamos a informação apenas nas folhas:

- para buscar basta ir para a esquerda ou para a direita
  - dependendo do valor do k-ésimo bit
- Ao chegar em `NULL`, a chave **não** está na árvore
- Ao chegar em uma folha, comparamos com o elemento

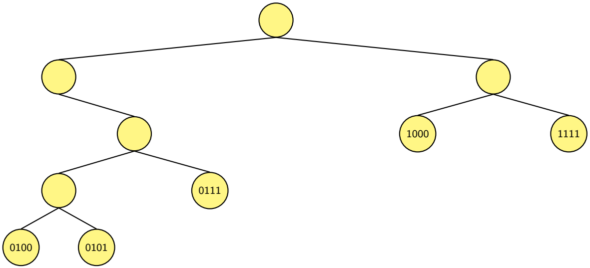
Tries - Busca



Guardamos a informação apenas nas folhas:

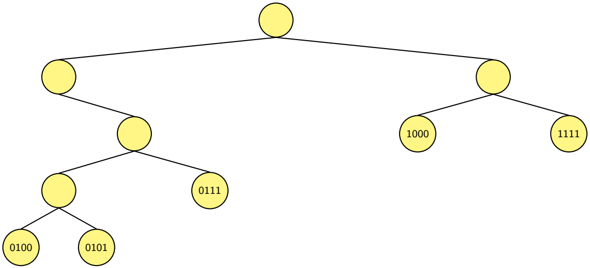
- para buscar basta ir para a esquerda ou para a direita
  - dependendo do valor do k-ésimo bit
- Ao chegar em `NULL`, a chave **não** está na árvore
- Ao chegar em uma folha, comparamos com o elemento

Tries - Inserção



Para inserir um elemento, realizamos uma busca pela chave:

Tries - Inserção

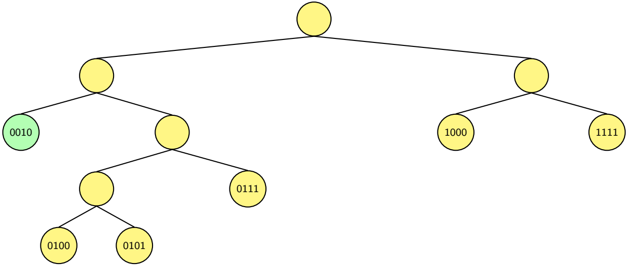


Para inserir um elemento, realizamos uma busca pela chave:

- Se a busca terminar em um nó `NULL`

Exemplo: $2_2 = 0010$

Tries - Inserção

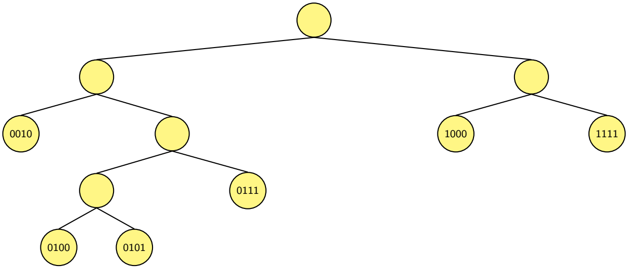


Para inserir um elemento, realizamos uma busca pela chave:

- Se a busca terminar em um nó `NULL`
  - inserimos um novo nó com a chave

Exemplo: $2_2 = 0010$

Tries - Inserção

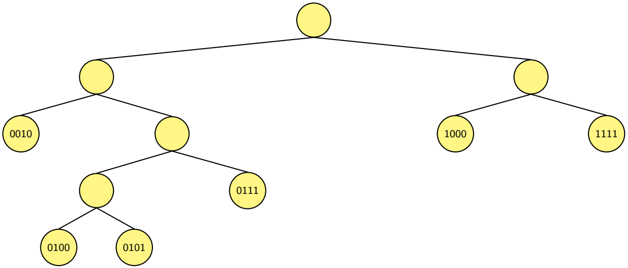


Para inserir um elemento, realizamos uma busca pela chave:

- Se a busca terminar em um nó NULL
  - inserimos um novo nó com a chave
- Se a busca terminar em uma folha v

Exemplo: $9_2 = 1001$

Tries - Inserção

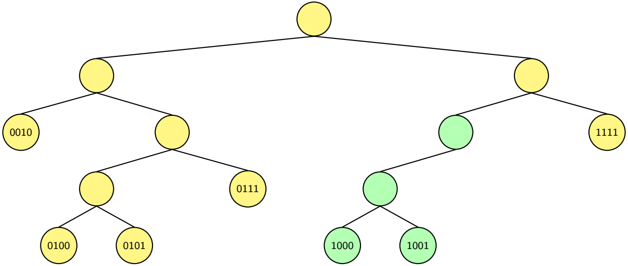


Para inserir um elemento, realizamos uma busca pela chave:

- Se a busca terminar em um nó `NULL`
  - inserimos um novo nó com a chave
- Se a busca terminar em uma folha *v*
  - Precisamos descer na árvore até diferenciar a chave com *v*

Exemplo: $9_2 = 1001$

Tries - Inserção



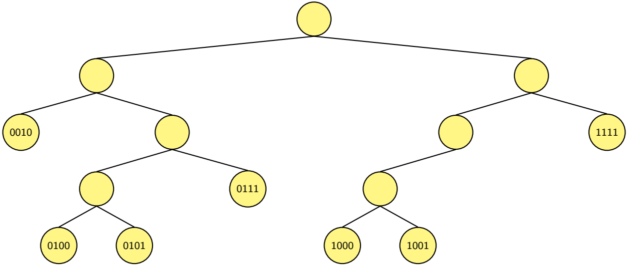
Para inserir um elemento, realizamos uma busca pela chave:

- Se a busca terminar em um nó `NULL`
  - inserimos um novo nó com a chave
- Se a busca terminar em uma folha v
  - Precisamos descer na árvore até diferenciar a chave com v

---

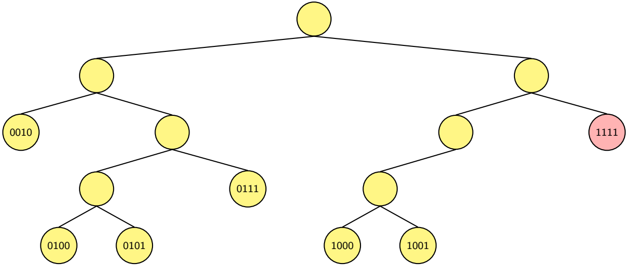
Exemplo: $9_2 = 1001$

Tries - Remoção



Para remover um elemento, realizamos uma busca pela chave:

Tries - Remoção

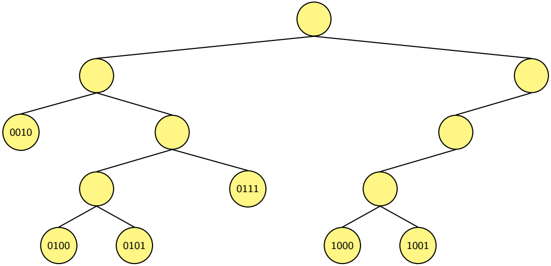


Para remover um elemento, realizamos uma busca pela chave:

- A busca deve terminar em um nó folha
  1. removemos a folha

Exemplo: $15_2 = 1111$

Tries - Remoção

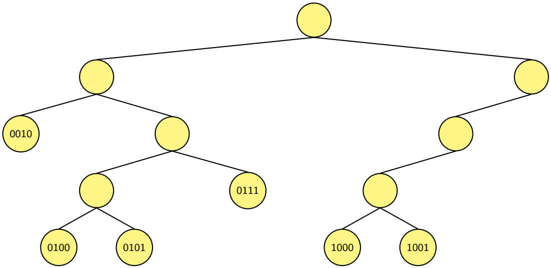


Para remover um elemento, realizamos uma busca pela chave:

- A busca deve terminar em um nó folha
    1. removemos a folha
    2. vamos para o nó pai v

Exemplo: $15_2 = 1111$

Tries - Remoção



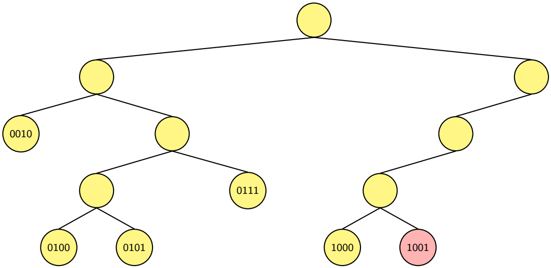
Para remover um elemento, realizamos uma busca pela chave:

- A busca deve terminar em um nó folha
  1. removemos a folha
  2. vamos para o nó pai v
  3. enquanto o pai de v tiver um único filho, "colapsamos" o caminho

---

Exemplo: $15_2 = 1111$

Tries - Remoção



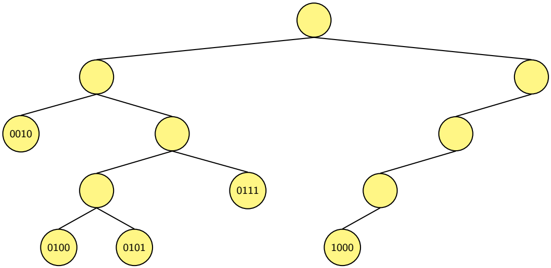
Para remover um elemento, realizamos uma busca pela chave:

- A busca deve terminar em um nó folha
  1. removemos a folha
  2. vamos para o nó pai v
  3. enquanto o pai de v tiver um único filho, “colapsamos” o caminho

---

Exemplo: $9_2 = 1001$

Tries - Remoção



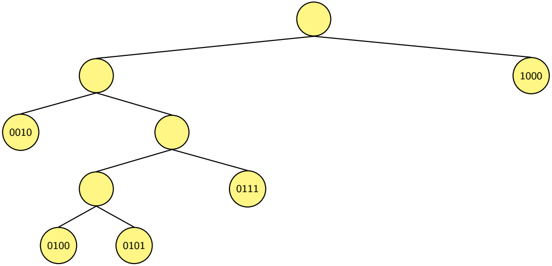
Para remover um elemento, realizamos uma busca pela chave:

- A busca deve terminar em um nó folha
  1. removemos a folha
  2. vamos para o nó pai v
  3. enquanto o pai de v tiver um único filho, "colapsamos" o caminho

---

Exemplo: $9_2 = 1001$

Tries - Remoção



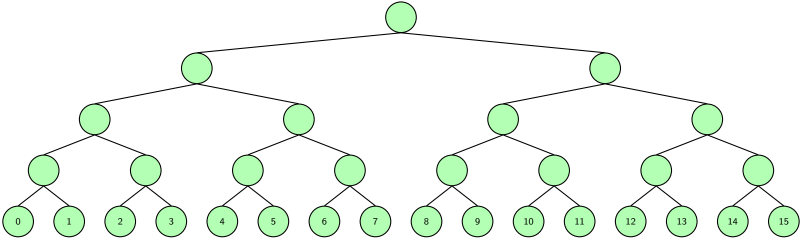
Para remover um elemento, realizamos uma busca pela chave:

- A busca deve terminar em um nó folha
  1. removemos a folha
  2. vamos para o nó pai v
  3. enquanto o pai de v tiver um único filho, "colapsamos" o caminho

Exemplo: $9_2 = 1001$

Tries - Exemplo

Trie para os valores $\{0, 1, 2, \ldots, 15\}$:

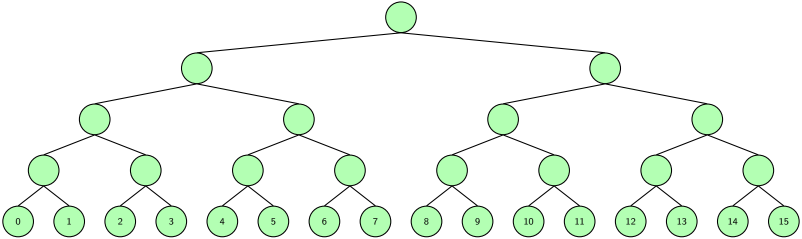


Altura h da árvore é $\lceil \lg(n) \rceil$.

Tries - Exemplo

Trie para os valores $\{0, 1, 2, \ldots, 15\}$:

- As folhas estão ordenadas.

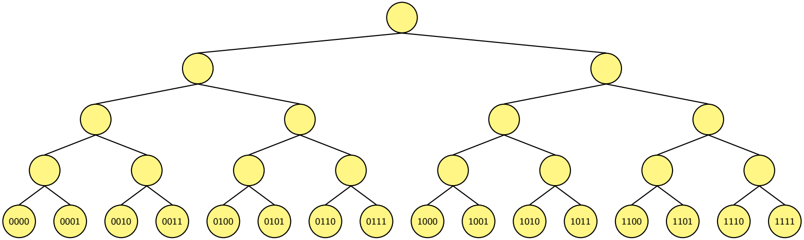


A estrutura de uma Trie é independente da ordem de inserção.

Tries - Problemas

Problema 1:

- Armazenar os dados apenas nas folhas desperdiça memória

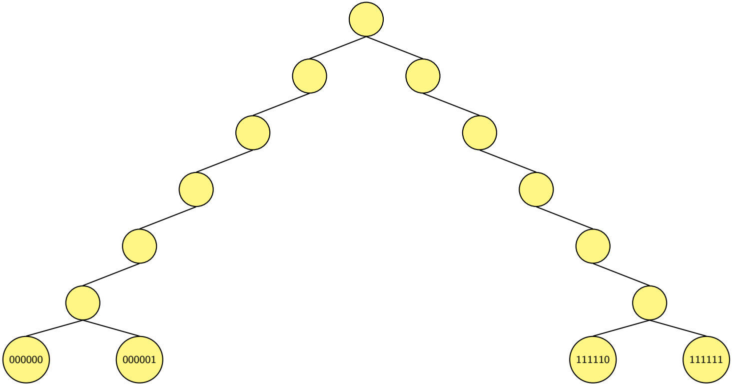


Número de nós internos é igual à $n - 1$, para n folhas (metade do espaço é disperdiçado).

Tries - Problemas

Problema 2:

- Para diferenciar duas chaves podem surgir longos caminhos



Roteiro

Patricia Tries

Practical algorithm to retreive informaton coded in alphanumeric

Proposta para `indexação de strings`, é igualmente efetiva para chaves de bits.

Patricia Tries

Practical algorithm to retreive informaton coded in alphanumeric

1. Cada nó armazena o índice do bit que deve ser testado para decidir o caminho.
   - para diferenciar 000000 de 000001, checamos o sexto bit
   - não precisamos olhar os outros bits

Proposta para indexação de `strings`, é igualmente efetiva para chaves de bits.

Patricia Tries

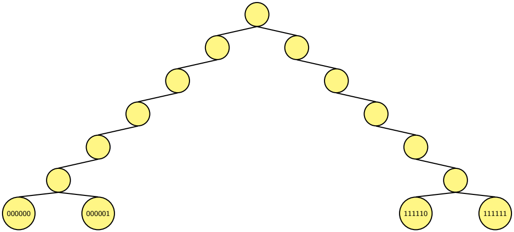
Practical algorithm to retreive informaton coded in alphanumeric

1. Cada nó armazena o índice do bit que deve ser testado para decidir o caminho.
   - para diferenciar 000000 de 000001, checamos o sexto bit
   - não precisamos olhar os outros bits
2. As chaves são armazenadas também nos nós internos
   - Utilizamos links "para cima" para percorrer a árvore

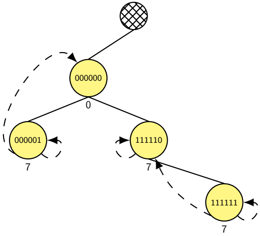
Proposta para indexação de `strings`, é igualmente efetiva para chaves de bits.

Patricia Trie - Comparação com Trie

Trie

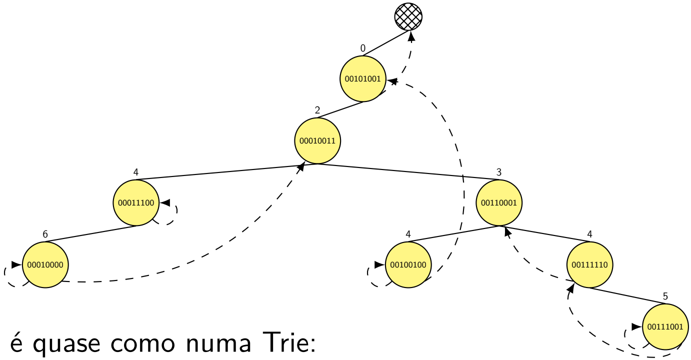


Patricia Trie



No lugar de um ponteiro para NULL, apontamos para um nó da árvore.

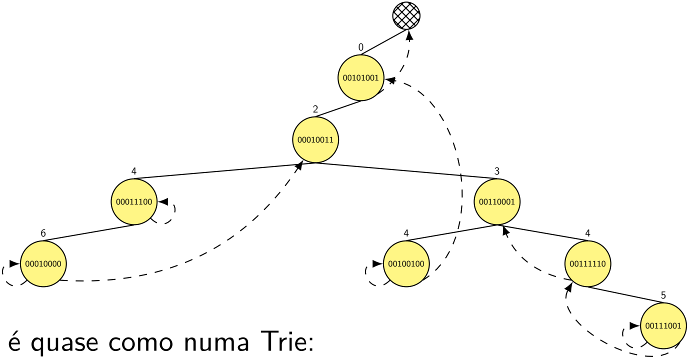
Patricia Trie



A busca é quase como numa Trie:

- descemos na árvore indo para a esquerda ou para a direita
- dependendo se o bit é zero ou um

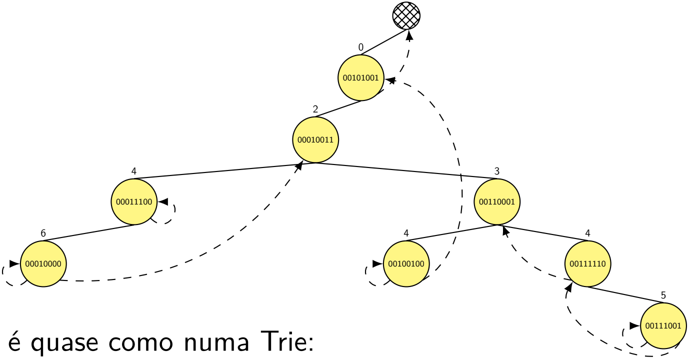
Patricia Trie



A busca é quase como numa Trie:

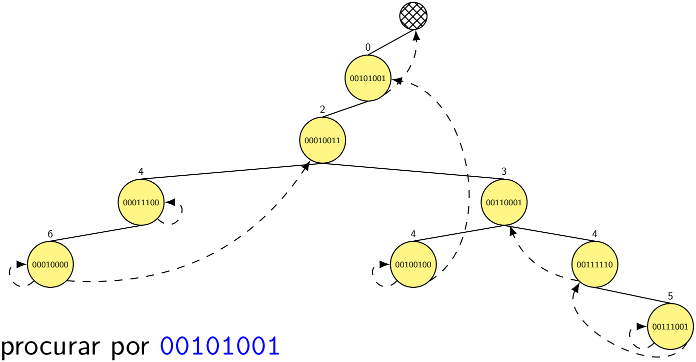
- descemos na árvore indo para a esquerda ou para a direita
- dependendo se o bit é zero ou um
  - mas podemos pular alguns bits
  - cada nó sabe qual bit que deve ser usado (evita caminhos desnecessários)

Patricia Trie



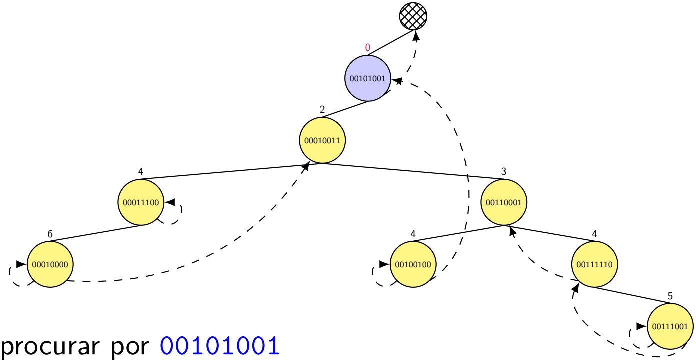
A busca é quase como numa Trie:

- descemos na árvore indo para a esquerda ou para a direita
- dependendo se o bit é zero ou um
  - mas podemos pular alguns bits
  - cada nó sabe qual bit que deve ser usado (evita caminhos desnecessários)
- eventualmente, subiremos na árvore
  - basta comparar com a chave e encerrar a busca
  - é como se tivéssemos chegado na folha da Trie



Vamos procurar por 00101001

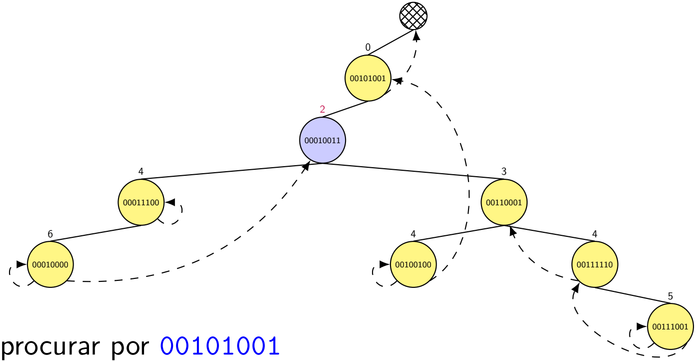
Patricia Trie - Simulação de Busca



Vamos procurar por 00101001

- O bit 0 de 00101001 é 0 - vá para a esquerda

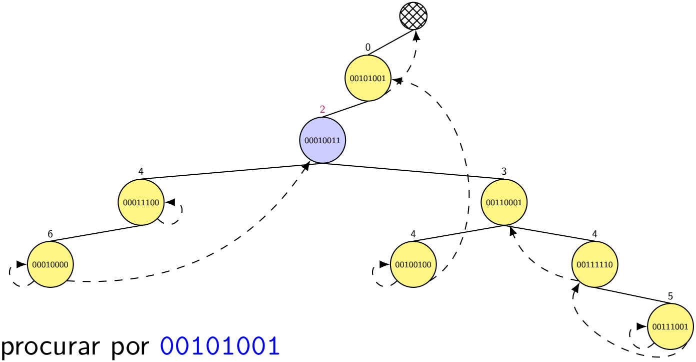
Patricia Trie - Simulação de Busca



Vamos procurar por 00101001

- O bit 0 de 00101001 é 0 - vá para a esquerda

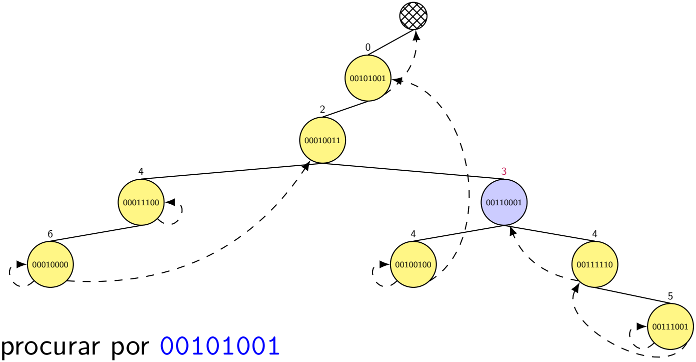
Patricia Trie - Simulação de Busca



Vamos procurar por 00101001

- O bit 0 de 00101001 é 0 - vá para a esquerda
- O bit 2 de 00101001 é 1 - vá para a direita

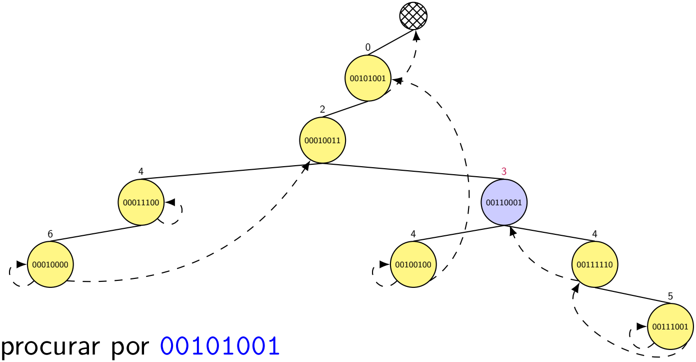
Patricia Trie - Simulação de Busca



Vamos procurar por 00101001

- O bit 0 de 00101001 é 0 - vá para a esquerda
- O bit 2 de 00101001 é 1 - vá para a direita

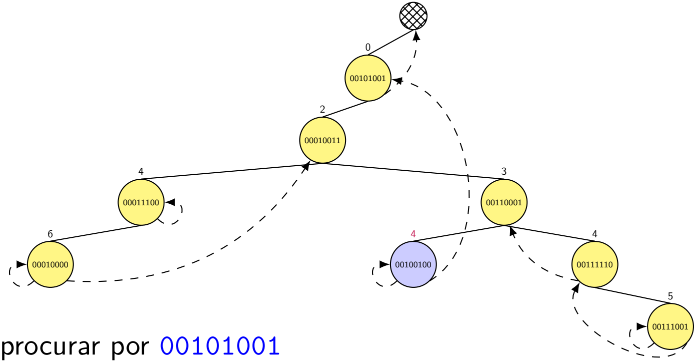
Patricia Trie - Simulação de Busca



Vamos procurar por 00101001

- O bit 0 de 00101001 é 0 - vá para a esquerda
- O bit 2 de 00101001 é 1 - vá para a direita
- O bit 3 de 00101001 é 0 - vá para a esquerda

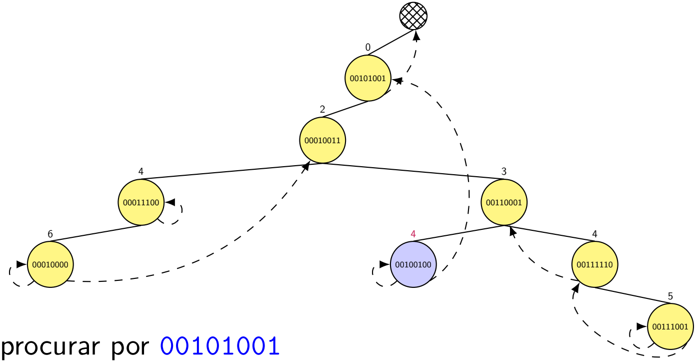
Patricia Trie - Simulação de Busca



Vamos procurar por 00101001

- O bit 0 de 00101001 é 0 - vá para a esquerda
- O bit 2 de 00101001 é 1 - vá para a direita
- O bit 3 de 00101001 é 0 - vá para a esquerda

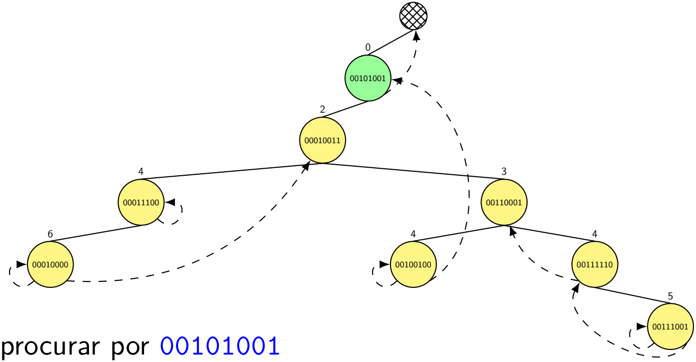
Patricia Trie - Simulação de Busca



Vamos procurar por 00101001

- O bit 0 de 00101001 é 0 - vá para a esquerda
- O bit 2 de 00101001 é 1 - vá para a direita
- O bit 3 de 00101001 é 0 - vá para a esquerda
- O bit 4 de 00101001 é 1 - vá para a direita

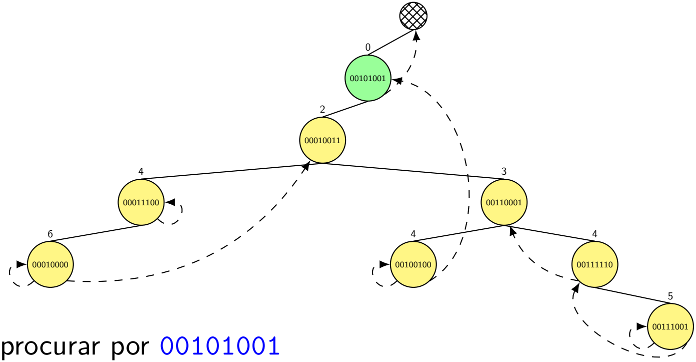
Patricia Trie - Simulação de Busca



Vamos procurar por 00101001

- O bit 0 de 00101001 é 0 - vá para a esquerda
- O bit 2 de 00101001 é 1 - vá para a direita
- O bit 3 de 00101001 é 0 - vá para a esquerda
- O bit 4 de 00101001 é 1 - vá para a direita

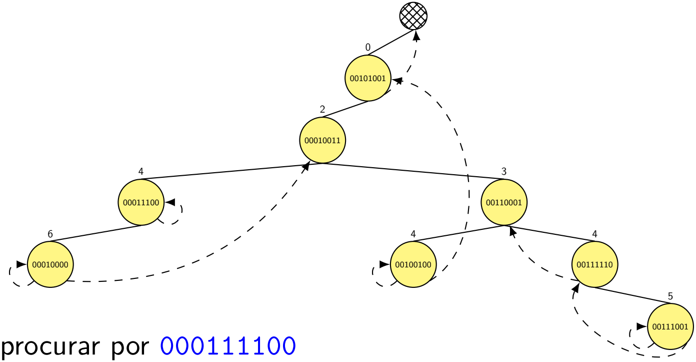
Patricia Trie - Simulação de Busca



Vamos procurar por 00101001

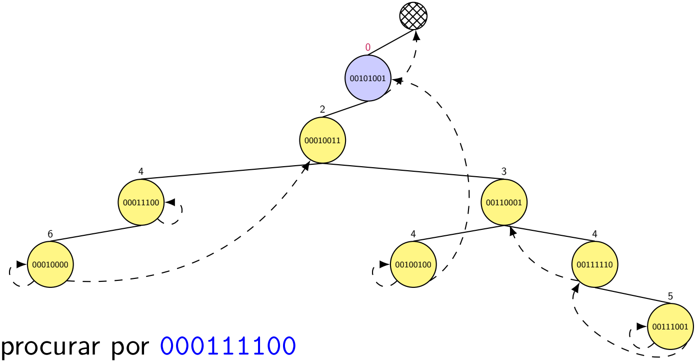
- O bit 0 de 00101001 é 0 - vá para a esquerda
- O bit 2 de 00101001 é 1 - vá para a direita
- O bit 3 de 00101001 é 0 - vá para a esquerda
- O bit 4 de 00101001 é 1 - vá para a direita
- Subiu na árvore - compare com a chave do nó atual (como uma folha na Trie)
  - encontrou 00101001

Patricia Trie - Simulação de Busca



Vamos procurar por 000111100

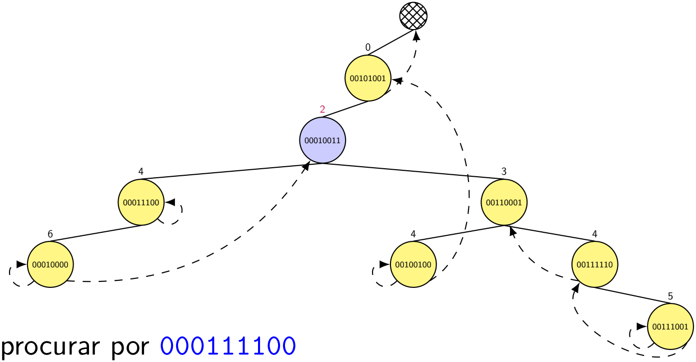
Patricia Trie - Simulação de Busca



Vamos procurar por 000111100

- O bit 0 de 000111100 é 0 - vá para a esquerda

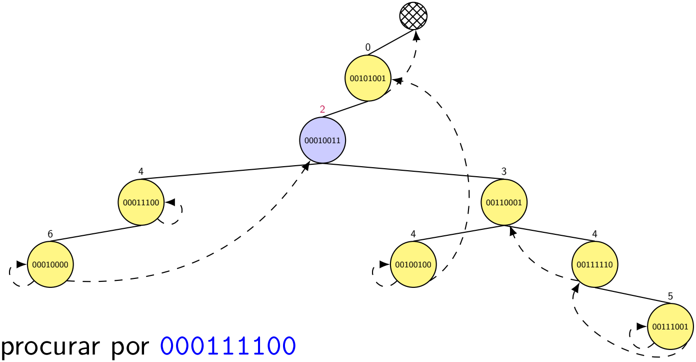
Patricia Trie - Simulação de Busca



Vamos procurar por 000111100

- O bit 0 de 000111100 é 0 - vá para a esquerda

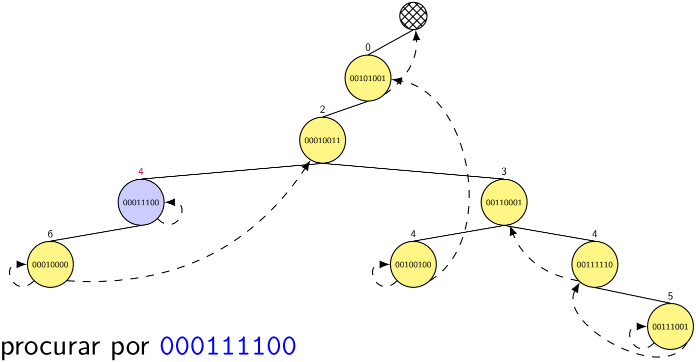
Patricia Trie - Simulação de Busca



Vamos procurar por 000111100

- O bit 0 de 000111100 é 0 - vá para a esquerda
- O bit 2 de 000111100 é 0 - vá para a esquerda

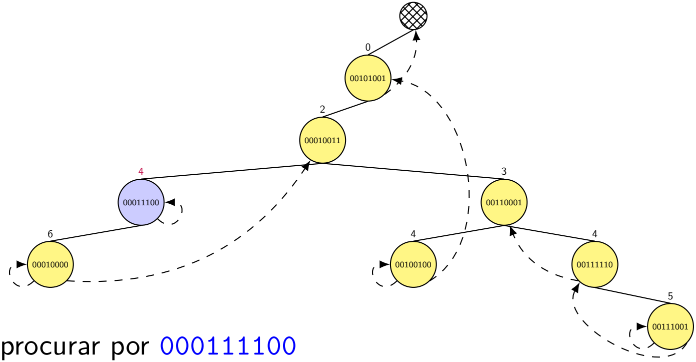
Patricia Trie - Simulação de Busca



Vamos procurar por 000111100

- O bit 0 de 000111100 é 0 - vá para a esquerda
- O bit 2 de 000111100 é 0 - vá para a esquerda

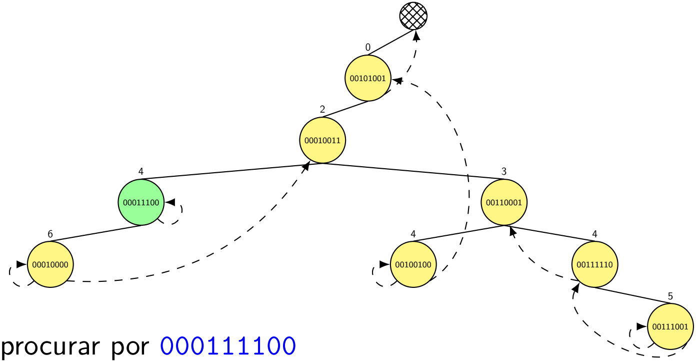
Patricia Trie - Simulação de Busca



Vamos procurar por 000111100

- O bit 0 de 000111100 é 0 - vá para a esquerda
- O bit 2 de 000111100 é 0 - vá para a esquerda
- O bit 4 de 000111100 é 1 - vá para a direita

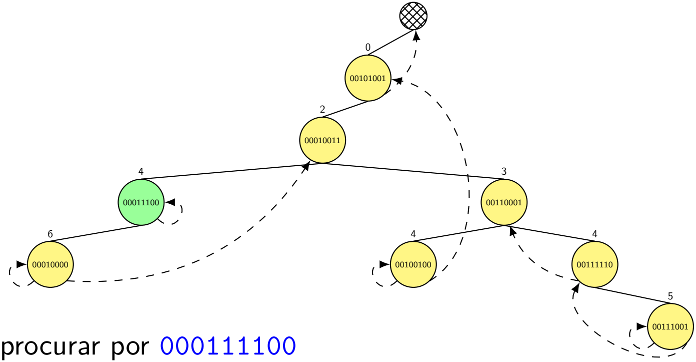
Patricia Trie - Simulação de Busca



Vamos procurar por 000111100

- O bit 0 de 000111100 é 0 - vá para a esquerda
- O bit 2 de 000111100 é 0 - vá para a esquerda
- O bit 4 de 000111100 é 1 - vá para a direita

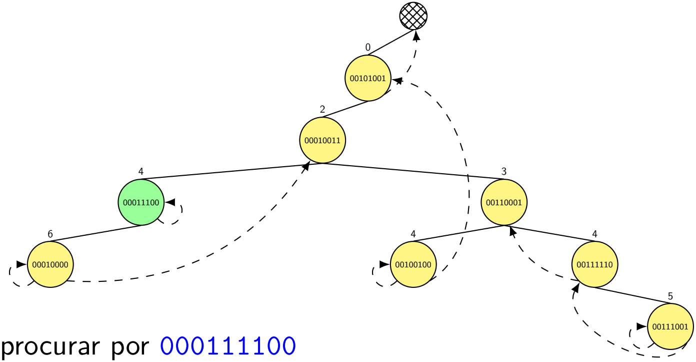
Patricia Trie - Simulação de Busca



Vamos procurar por 000111100

- O bit 0 de 000111100 é 0 - vá para a esquerda
- O bit 2 de 000111100 é 0 - vá para a esquerda
- O bit 4 de 000111100 é 1 - vá para a direita
- Subiu na árvore - compare com a chave do nó atual
  - não encontrou 000111100

Patricia Trie - Simulação de Busca



Vamos procurar por 000111100

- O bit 0 de 000111100 é 0 - vá para a esquerda
- O bit 2 de 000111100 é 0 - vá para a esquerda
- O bit 4 de 000111100 é 1 - vá para a direita
- Subiu na árvore - compare com a chave do nó atual
  - não encontrou 000111100

Sabemos que a busca subiu quando o bit for menor ou igual ao atual.

Patricia Trie - Inserção e Remoção

Não vamos ver as operações de **Inserção** e **Remoção**:

1. nos dois casos precisamos buscar a chave na árvore
2. inserir/remover corrigindo a estrutura

R. Sedgewick, "*Algorithms in C - Parts 1-4 - Third Edition*" (Capítulo 15)

Conclusão

Árvores Digitais de Busca:

- Altura $\mathrm{O}(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $\mathrm{O}(k) = \mathrm{O}(\lg n)$
- Mas não tem uma ordenação das chaves...

Conclusão

Árvores Digitais de Busca:

- Altura $\mathrm{O}(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $\mathrm{O}(k) = \mathrm{O}(\lg n)$
- Mas não tem uma ordenação das chaves...

Tries:

- Resolvem o problema das árvores digitais de busca
- Mas gastam **muito espaço** e tempo para diferenciar chaves

Conclusão

Árvores Digitais de Busca:

- Altura $O(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $O(k) = O(\lg n)$
- Mas não tem uma ordenação das chaves...

Tries:

- Resolvem o problema das árvores digitais de busca
- Mas gastam **muito espaço** e tempo para diferenciar chaves

Patricia Tries:

- Resolvem o problema das Tries (**economizam espaço**)
- Podem ser usadas para indexar chaves de tamanho variável
- Podem ser usadas em Radix maiores que 2
  - Por exemplo, 256 para indexar palavras
  - Nó precisa ter até 256 filhos...

Fim

Dúvidas?

Roteiro

Referências

1. Materiais adaptados dos slides do Prof. Rafael C. S. Schouery, da Universidade Estadual de Campinas.
2. R. Sedgewick, "*Algorithms in C - Parts 1-4 - Third Edition*" (Capítulo 15)