

MC-202 — Unidade 17

Busca Radix, Árvores Digitais e Tries

Rafael C. S. Schouery
rafael@ic.unicamp.br

Universidade Estadual de Campinas

1º semestre/2017

Busca Radix

Veremos outro tipo de **árvores binárias**:

Busca Radix

Veremos outro tipo de **árvores binárias**:

- Mas não são árvores binárias **de busca**...

Busca Radix

Veremos outro tipo de **árvores binárias**:

- Mas não são árvores binárias **de busca**...
- Usadas quando a chave de busca tem poucos bits

Busca Radix

Veremos outro tipo de **árvores binárias**:

- Mas não são árvores binárias **de busca**...
- Usadas quando a chave de busca tem poucos bits
- E extrair o bit é uma operação rápida

Busca Radix

Veremos outro tipo de **árvores binárias**:

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

Busca Radix

Veremos outro tipo de **árvores binárias**:

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

Se a chave de busca tiver no máximo k bits

Busca Radix

Veremos outro tipo de **árvores binárias**:

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

Se a chave de busca tiver no máximo k bits

- a altura da árvore será pequena: $O(k)$

Busca Radix

Veremos outro tipo de **árvores binárias**:

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

Se a chave de busca tiver no máximo k bits

- a altura da árvore será pequena: $O(k)$
- mas poderá guardar muitas chaves: 2^k

Busca Radix

Veremos outro tipo de **árvores binárias**:

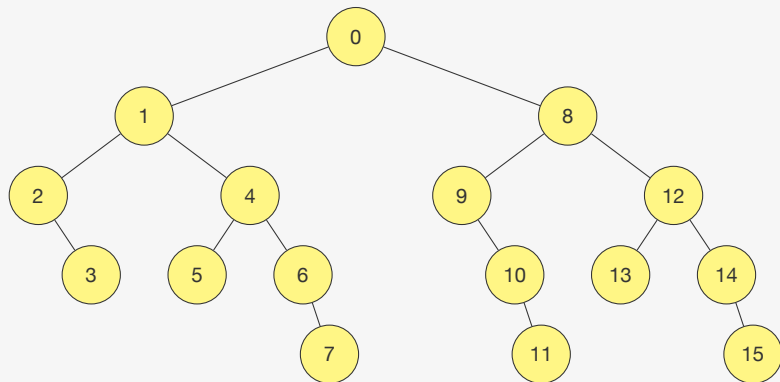
- Mas não são árvores binárias **de busca**...
- Usadas quando a chave de busca tem poucos bits
- E extrair o bit é uma operação rápida
 - `int`, `char`, `char *`, etc...

Se a chave de busca tiver no máximo k bits

- a altura da árvore será pequena: $O(k)$
- mas poderá guardar muitas chaves: 2^k

Usando códigos mais simples e com menos overhead do que uma árvore **rubro-negra**

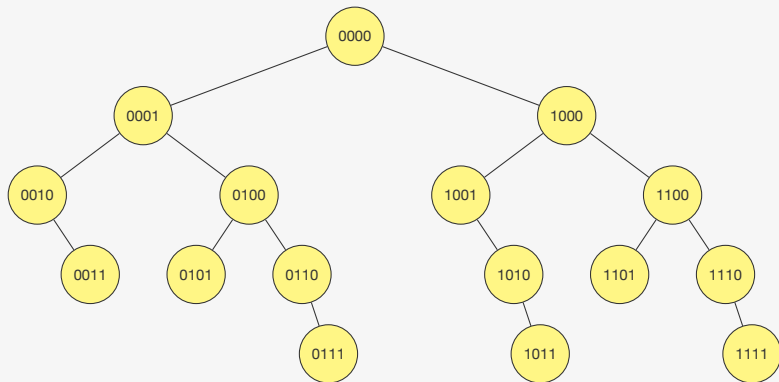
Árvores Digitais de Busca



Note que não é uma árvore de busca binária

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

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**

Implementação

Veremos a implementação para chaves do tipo `unsigned`

Implementação

Veremos a implementação para chaves do tipo `unsigned`

```
1 typedef struct No {  
2     unsigned chave;  
3     struct No *esq, *dir;  
4 } No;
```

Implementação

Veremos a implementação para chaves do tipo `unsigned`

```
1 typedef struct No {  
2     unsigned chave;  
3     struct No *esq, *dir;  
4 } No;
```

Usualmente, temos uma estrutura mais complicada com uma chave de busca

Implementação

Veremos a implementação para chaves do tipo `unsigned`

```
1 typedef struct No {  
2     unsigned chave;  
3     struct No *esq, *dir;  
4 } No;
```

Usualmente, temos uma estrutura mais complicada com uma chave de busca

- Basta trocar a chave por essa estrutura

Implementação

Veremos a implementação para chaves do tipo `unsigned`

```
1 typedef struct No {  
2     unsigned chave;  
3     struct No *esq, *dir;  
4 } No;
```

Usualmente, temos uma estrutura mais complicada com uma chave de busca

- Basta trocar a chave por essa estrutura
- e modificar os códigos para acessar a chave da estrutura

Implementação

Veremos a implementação para chaves do tipo `unsigned`

```
1 typedef struct No {  
2     unsigned chave;  
3     struct No *esq, *dir;  
4 } No;
```

Usualmente, temos uma estrutura mais complicada com uma chave de busca

- Basta trocar a chave por essa estrutura
- e modificar os códigos para acessar a chave da estrutura

Precisaremos do k -ésimo bit (da esquerda para a direita):

Implementação

Veremos a implementação para chaves do tipo `unsigned`

```
1 typedef struct No {
2     unsigned chave;
3     struct No *esq, *dir;
4 } No;
```

Usualmente, temos uma estrutura mais complicada com uma chave de busca

- Basta trocar a chave por essa estrutura
- e modificar os códigos para acessar a chave da estrutura

Precisaremos do k -ésimo bit (da esquerda para a direita):

```
1 #define bits_na_chave 32
2
3 unsigned bit(unsigned chave, int k) {
4     return chave >> (bits_na_chave - 1 - k) & 1;
5 }
```

Busca

```
1 No * busca_rec(No *arvore, unsigned x, int nivel) {  
2     if (arvore == NULL)  
3         return NULL;
```

Busca

```
1 No * busca_rec(No *arvore, unsigned x, int nivel) {
2     if (arvore == NULL)
3         return NULL;
4     if (x == arvore->chave)
5         return arvore;
```

Busca

```
1 No * busca_rec(No *arvore, unsigned x, int nivel) {
2     if (arvore == NULL)
3         return NULL;
4     if (x == arvore->chave)
5         return arvore;
6     if (bit(x, nivel) == 0)
7         return busca_rec(arvore->esq, x, nivel+1);
```

Busca

```
1 No * busca_rec(No *arvore, unsigned x, int nivel) {
2     if (arvore == NULL)
3         return NULL;
4     if (x == arvore->chave)
5         return arvore;
6     if (bit(x, nivel) == 0)
7         return busca_rec(arvore->esq, x, nivel+1);
8     else
9         return busca_rec(arvore->dir, x, nivel+1);
10 }
```

Busca

```
1 No * busca_rec(No *arvore, unsigned x, int nivel) {
2     if (arvore == NULL)
3         return NULL;
4     if (x == arvore->chave)
5         return arvore;
6     if (bit(x, nivel) == 0)
7         return busca_rec(arvore->esq, x, nivel+1);
8     else
9         return busca_rec(arvore->dir, x, nivel+1);
10 }
11
12 No * busca(No *arvore, unsigned x) {
13     return busca_rec(arvore, x, 0);
14 }
```


Busca

```
1 No * busca_rec(No *arvore, unsigned x, int nivel) {
2     if (arvore == NULL)
3         return NULL;
4     if (x == arvore->chave)
5         return arvore;
6     if (bit(x, nivel) == 0)
7         return busca_rec(arvore->esq, x, nivel+1);
8     else
9         return busca_rec(arvore->dir, x, nivel+1);
10 }
11
12 No * busca(No *arvore, unsigned x) {
13     return busca_rec(arvore, x, 0);
14 }
```

No pior caso, leva tempo $O(b)$

- b é o número de bits das chaves

Busca

```
1 No * busca_rec(No *arvore, unsigned x, int nivel) {
2     if (arvore == NULL)
3         return NULL;
4     if (x == arvore->chave)
5         return arvore;
6     if (bit(x, nivel) == 0)
7         return busca_rec(arvore->esq, x, nivel+1);
8     else
9         return busca_rec(arvore->dir, x, nivel+1);
10 }
11
12 No * busca(No *arvore, unsigned x) {
13     return busca_rec(arvore, x, 0);
14 }
```

No pior caso, leva tempo $O(b)$

- b é o número de bits das chaves

Se as chaves forem aleatórias, leva tempo $O(\lg n)$

Inserção

```
1 No *insere_rec(No *arvore, unsigned chave, int nivel) {
2     No *novo;
3     if (arvore == NULL) {
4         novo = malloc(sizeof(No));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         return novo;
8     }
```

Inserção

```
1 No *insere_rec(No *arvore, unsigned chave, int nivel) {
2     No *novo;
3     if (arvore == NULL) {
4         novo = malloc(sizeof(No));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         return novo;
8     }
9     if (chave == arvore->chave)
10        return arvore;
```

Inserção

```
1 No *insere_rec(No *arvore, unsigned chave, int nivel) {
2     No *novo;
3     if (arvore == NULL) {
4         novo = malloc(sizeof(No));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         return novo;
8     }
9     if (chave == arvore->chave)
10        return arvore;
11    if (bit(chave, nivel) == 0)
12        arvore->esq = insere_rec(arvore->esq, chave, nivel+1);
```

Inserção

```
1 No *insere_rec(No *arvore, unsigned chave, int nivel) {
2     No *novo;
3     if (arvore == NULL) {
4         novo = malloc(sizeof(No));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         return novo;
8     }
9     if (chave == arvore->chave)
10        return arvore;
11    if (bit(chave, nivel) == 0)
12        arvore->esq = insere_rec(arvore->esq, chave, nivel+1);
13    else
14        arvore->dir = insere_rec(arvore->dir, chave, nivel+1);
```

Inserção

```
1 No *insere_rec(No *arvore, unsigned chave, int nivel) {
2     No *novo;
3     if (arvore == NULL) {
4         novo = malloc(sizeof(No));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         return novo;
8     }
9     if (chave == arvore->chave)
10        return arvore;
11    if (bit(chave, nivel) == 0)
12        arvore->esq = insere_rec(arvore->esq, chave, nivel+1);
13    else
14        arvore->dir = insere_rec(arvore->dir, chave, nivel+1);
15    return arvore;
16 }
```

Inserção

```
1 No *insere_rec(No *arvore, unsigned chave, int nivel) {
2     No *novo;
3     if (arvore == NULL) {
4         novo = malloc(sizeof(No));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         return novo;
8     }
9     if (chave == arvore->chave)
10        return arvore;
11    if (bit(chave, nivel) == 0)
12        arvore->esq = insere_rec(arvore->esq, chave, nivel+1);
13    else
14        arvore->dir = insere_rec(arvore->dir, chave, nivel+1);
15    return arvore;
16 }
17
18 void insere(No **arvore, unsigned chave) {
19     *arvore = insere_rec(*arvore, chave, 0);
20 }
```


Inserção

```
1 No *insere_rec(No *arvore, unsigned chave, int nivel) {
2     No *novo;
3     if (arvore == NULL) {
4         novo = malloc(sizeof(No));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         return novo;
8     }
9     if (chave == arvore->chave)
10        return arvore;
11    if (bit(chave, nivel) == 0)
12        arvore->esq = insere_rec(arvore->esq, chave, nivel+1);
13    else
14        arvore->dir = insere_rec(arvore->dir, chave, nivel+1);
15    return arvore;
16 }
17
18 void insere(No **arvore, unsigned chave) {
19     *arvore = insere_rec(*arvore, chave, 0);
20 }
```

No pior caso, leva tempo $O(b)$

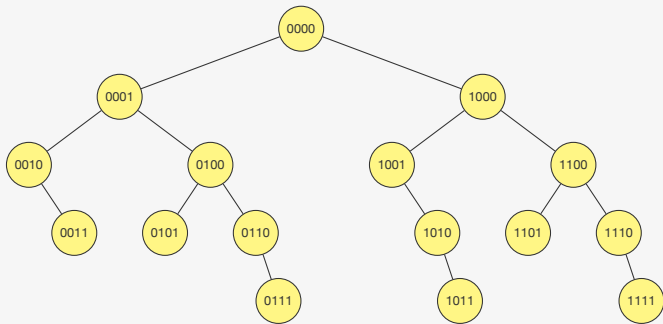
Inserção

```
1 No *insere_rec(No *arvore, unsigned chave, int nivel) {
2     No *novo;
3     if (arvore == NULL) {
4         novo = malloc(sizeof(No));
5         novo->esq = novo->dir = NULL;
6         novo->chave = chave;
7         return novo;
8     }
9     if (chave == arvore->chave)
10        return arvore;
11    if (bit(chave, nivel) == 0)
12        arvore->esq = insere_rec(arvore->esq, chave, nivel+1);
13    else
14        arvore->dir = insere_rec(arvore->dir, chave, nivel+1);
15    return arvore;
16 }
17
18 void insere(No **arvore, unsigned chave) {
19     *arvore = insere_rec(*arvore, chave, 0);
20 }
```

No pior caso, leva tempo $O(b)$

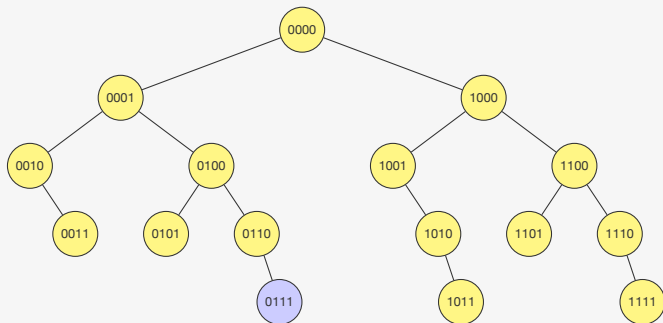
Se as chaves forem aleatórias, leva tempo $O(\lg n)$

Remoção



Para remover:

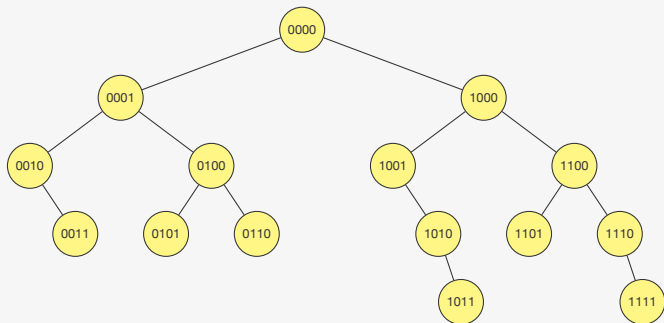
Remoção



Para remover:

- uma folha, basta apagá-la

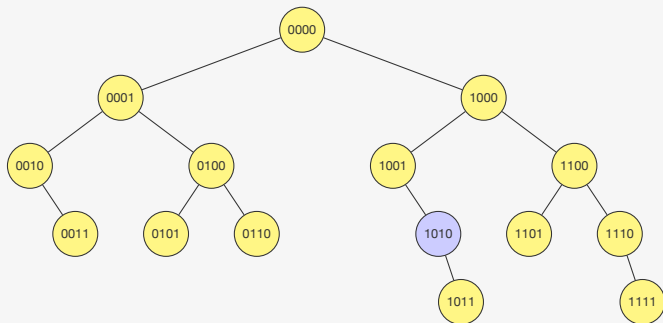
Remoção



Para remover:

- uma folha, basta apagá-la

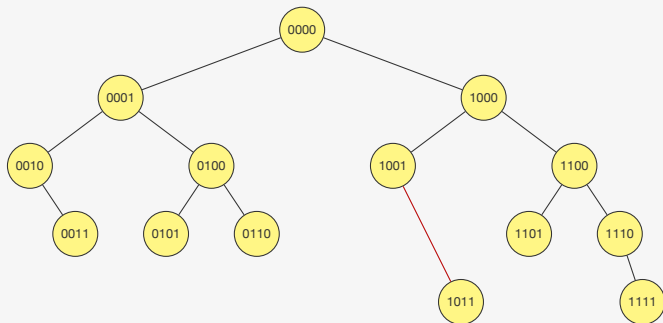
Remoção



Para remover:

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

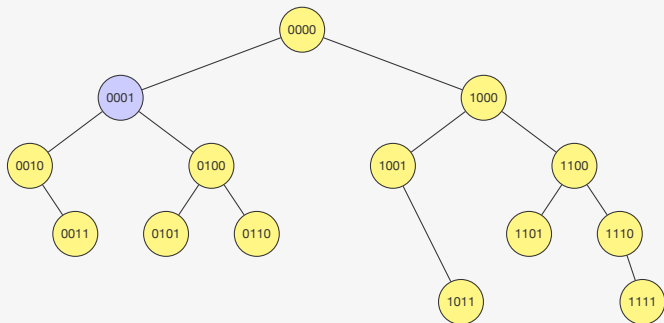
Remoção



Para remover:

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

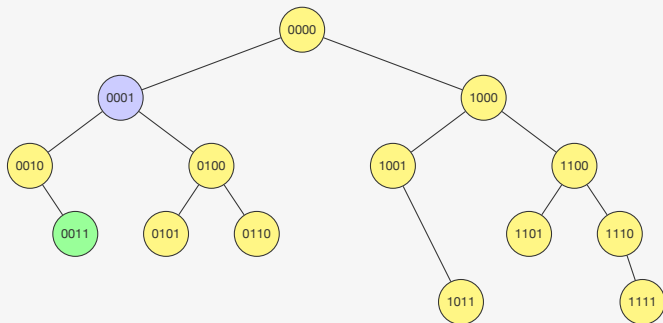
Remoção



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

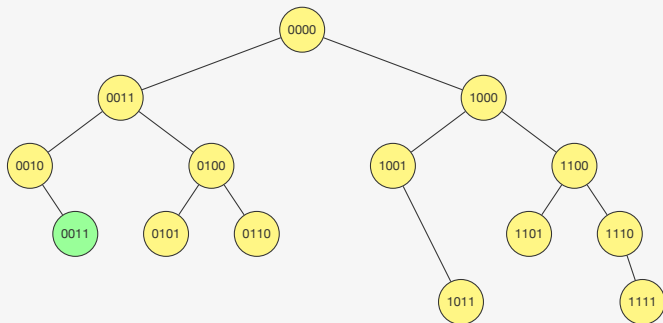
Remoção



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

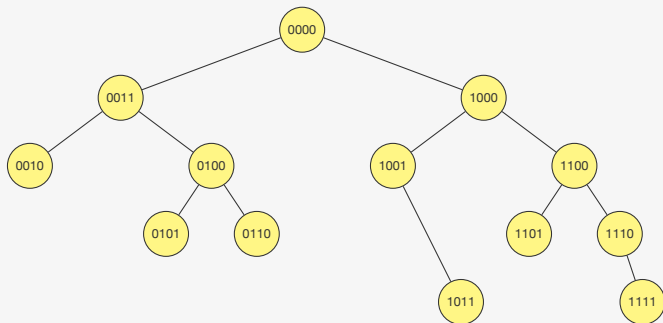
Remoção



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
 - apague o descendente

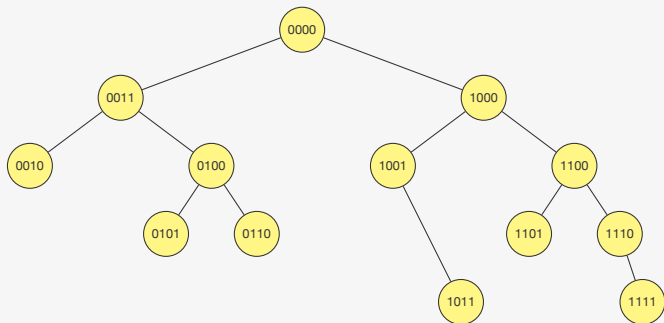
Remoção



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
 - apague o descendente

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
 - apague o descendente

Exercício: Faça a implementação da remoção

Tries

As árvores digitais de busca tem um problema:

Tries

As árvores digitais de busca tem um problema:

- As chaves não estão ordenadas

Tries

As árvores digitais de busca tem um problema:

- As chaves não estão ordenadas
- Algumas operações que envolvem a ordem ficam caras

Tries

As árvores digitais de busca tem um problema:

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

Tries

As árvores digitais de busca tem um problema:

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

Tries: utilizam uma ideia parecida

Tries

As árvores digitais de busca tem um problema:

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

Tries: utilizam uma ideia parecida

- Mas conseguimos usar a informação da ordem das chaves

Tries

As árvores digitais de busca tem um problema:

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

Tries: utilizam uma ideia parecida

- Mas conseguimos usar a informação da ordem das chaves
- Vem da palavra *retrieval*

Tries

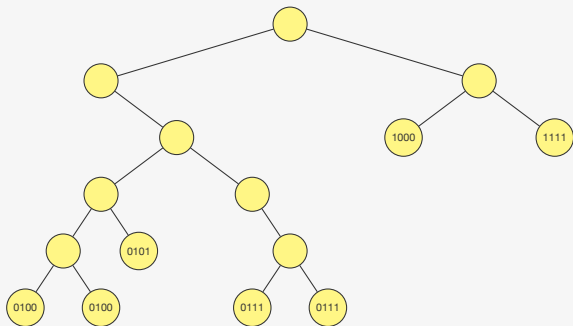
As árvores digitais de busca tem um problema:

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

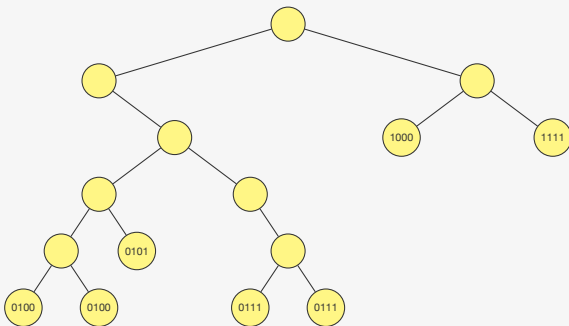
Tries: utilizam uma ideia parecida

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

Tries - Ideia

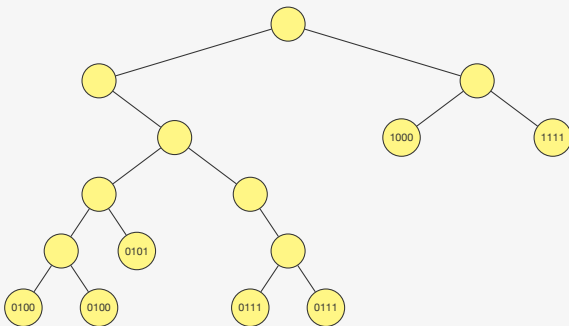


Tries - Ideia



Guardamos a informação nas apenas folhas:

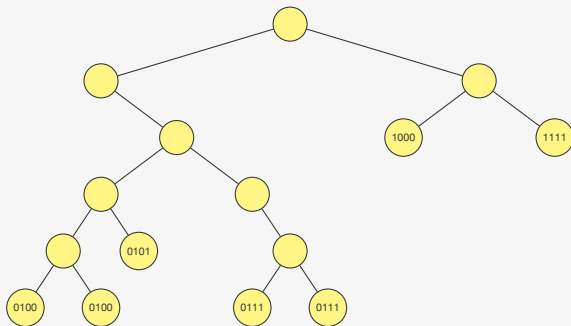
Tries - Ideia



Guardamos a informação nas apenas folhas:

- para buscar basta ir para a esquerda ou para a direita

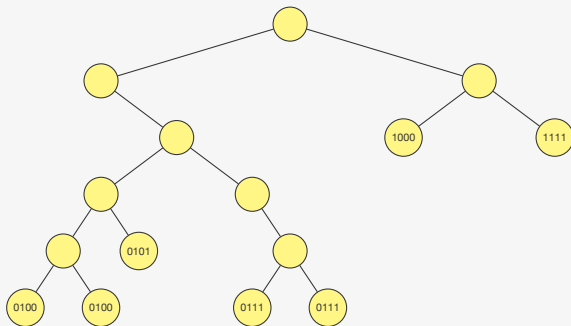
Tries - Ideia



Guardamos a informação nas apenas folhas:

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

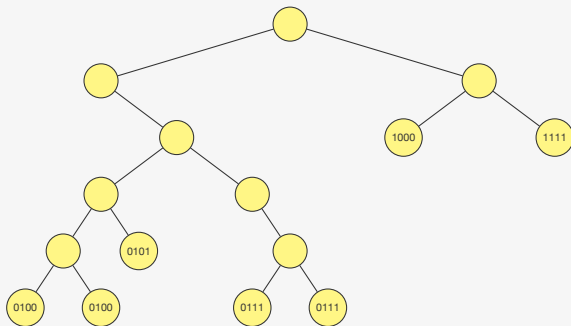
Tries - Ideia



Guardamos a informação nas apenas 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 - Ideia



Guardamos a informação nas apenas folhas:

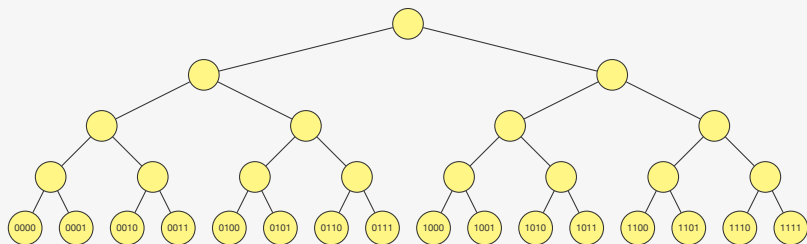
- para buscar basta ir para a esquerda ou para a direita
 - dependendo do valor do k -ésimo bit
- Ao chegar em **NULL**, a chave não está na árvore
- Ao chegar em uma folha, comparamos com o elemento

Tries - Problemas

Tries - Problemas

Problema 1:

- Armazenar os dados nas folhas desperdiça memória

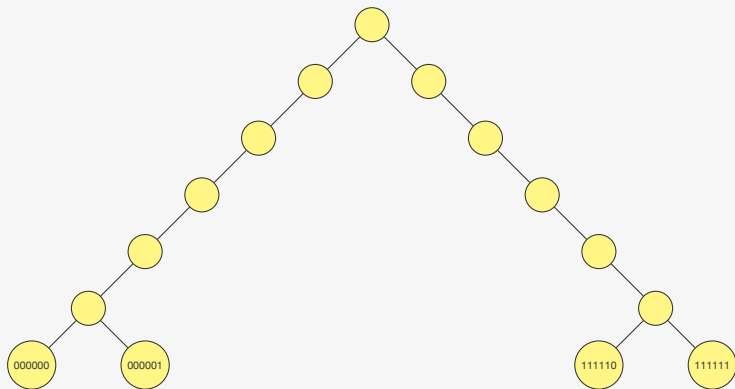


Tries - Problemas

Tries - Problemas

Problema 2:

- Para diferenciar duas chaves podem surgir longos caminhos



Patricia Tries

Practical algorithm to retrieve information coded in alphanumeric

Patricia Tries

Practical algorithm to retrieve information coded in alphanumeric

- As chaves são armazenadas nos nós internos

Patricia Tries

Practical algorithm to retrieve information coded in alphanumeric

- As chaves são armazenadas nos nós internos
- Evita longos caminhos olhando para o bit que importa

Patricia Tries

Practical algorithm to retrieve information coded in alphanumeric

- As chaves são armazenadas nos nós internos
- Evita longos caminhos olhando para o bit que importa
 - para diferenciar 000000 de 000001, checamos o sexto bit

Patricia Tries

Practical algorithm to retrieve information coded in alphanumeric

- As chaves são armazenadas nos nós internos
- Evita longos caminhos olhando para o bit que importa
 - para diferenciar 000000 de 000001, checamos o sexto bit
 - não precisa olhar os outros bits

Inicialização e estrutura

```
1 typedef struct No {  
2     unsigned chave;  
3     int bit;  
4     struct No *esq, *dir;  
5 } No;
```

Inicialização e estrutura

```
1 typedef struct No {  
2     unsigned chave;  
3     int bit;  
4     struct No *esq, *dir;  
5 } No;
```

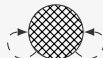
```
1 void inicializa(No **arvore) {  
2     *arvore = malloc(sizeof(No));  
3     (*arvore)->chave = UINT_MAX;  
4     (*arvore)->esq = (*arvore)->dir = *arvore;  
5     (*arvore)->bit = -1;  
6 }
```



Inicialização e estrutura

```
1 typedef struct No {  
2     unsigned chave;  
3     int bit;  
4     struct No *esq, *dir;  
5 } No;
```

```
1 void inicializa(No **arvore) {  
2     *arvore = malloc(sizeof(No));  
3     (*arvore)->chave = UINT_MAX;  
4     (*arvore)->esq = (*arvore)->dir = *arvore;  
5     (*arvore)->bit = -1;  
6 }
```

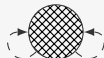


É uma árvore **com cabeça**: a raiz é um nó *dummy*

Inicialização e estrutura

```
1 typedef struct No {
2     unsigned chave;
3     int bit;
4     struct No *esq, *dir;
5 } No;
```

```
1 void inicializa(No **arvore) {
2     *arvore = malloc(sizeof(No));
3     (*arvore)->chave = UINT_MAX;
4     (*arvore)->esq = (*arvore)->dir = *arvore;
5     (*arvore)->bit = -1;
6 }
```



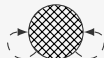
É uma árvore **com cabeça**: a raiz é um nó *dummy*

UINT_MAX será uma chave proibida, diferente de qualquer outra chave da árvore

Inicialização e estrutura

```
1 typedef struct No {  
2     unsigned chave;  
3     int bit;  
4     struct No *esq, *dir;  
5 } No;
```

```
1 void inicializa(No **arvore) {  
2     *arvore = malloc(sizeof(No));  
3     (*arvore)->chave = UINT_MAX;  
4     (*arvore)->esq = (*arvore)->dir = *arvore;  
5     (*arvore)->bit = -1;  
6 }
```



É uma árvore **com cabeça**: a raiz é um nó *dummy*

UINT_MAX será uma chave proibida, diferente de qualquer outra chave da árvore

- É o valor máximo de um **unsigned**

Inicialização e estrutura

```
1 typedef struct No {
2     unsigned chave;
3     int bit;
4     struct No *esq, *dir;
5 } No;
```

```
1 void inicializa(No **arvore) {
2     *arvore = malloc(sizeof(No));
3     (*arvore)->chave = UINT_MAX;
4     (*arvore)->esq = (*arvore)->dir = *arvore;
5     (*arvore)->bit = -1;
6 }
```

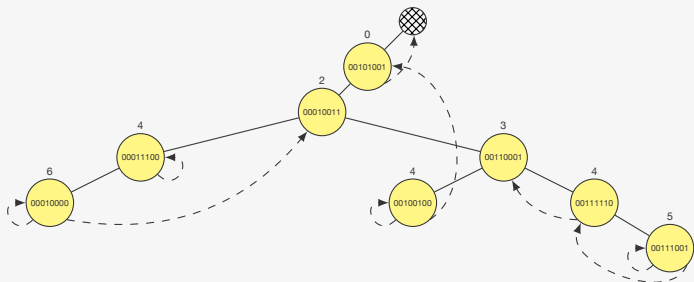


É uma árvore **com cabeça**: a raiz é um nó *dummy*

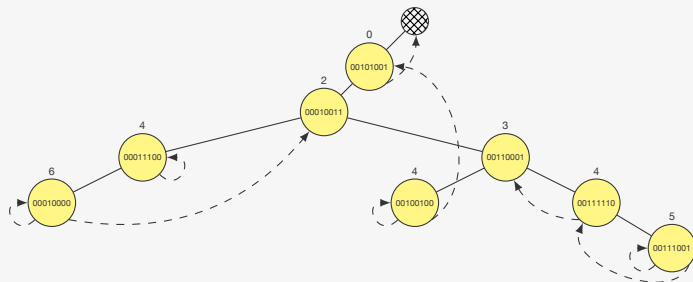
UINT_MAX será uma chave proibida, diferente de qualquer outra chave da árvore

- É o valor máximo de um **unsigned**
- Definido em `<limits.h>`

Patricia Trie

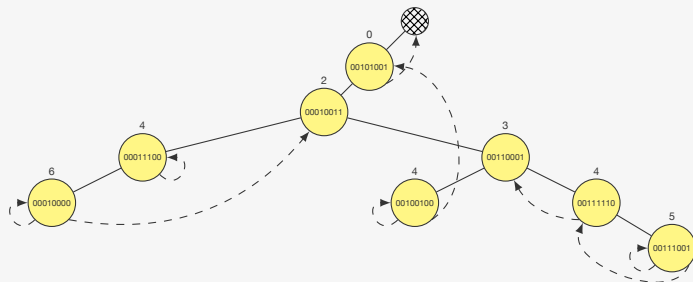


Patricia Trie



A busca é quase como numa Trie:

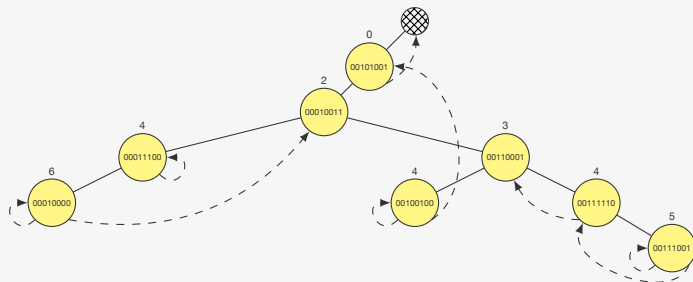
Patricia Trie



A busca é quase como numa Trie:

- descemos na árvore indo para a esquerda ou para a direita

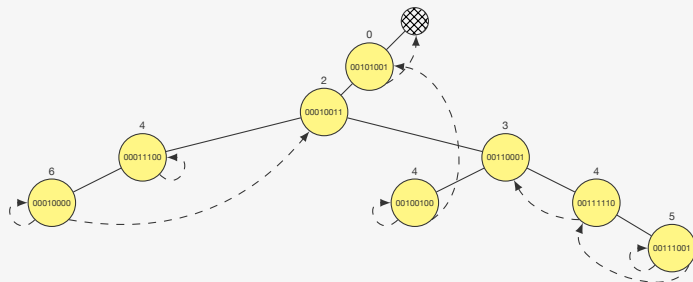
Patricia Trie



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

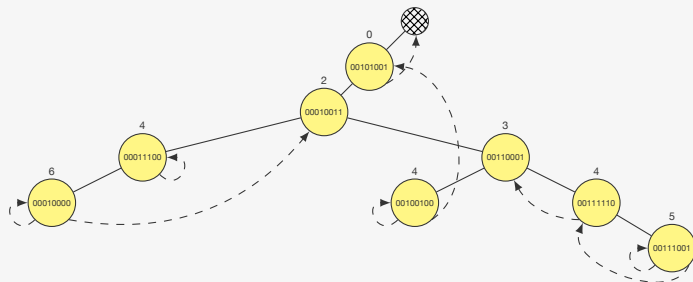
Patricia Trie



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

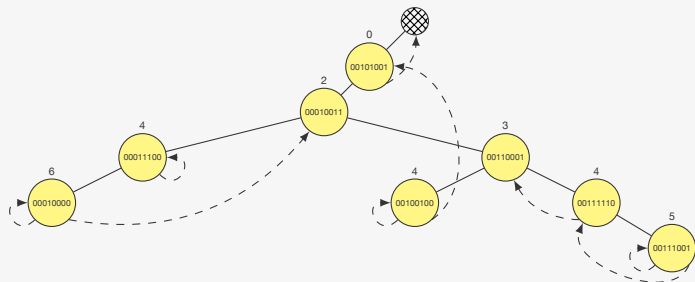
Patricia Trie



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 é o bit que deve ser usado

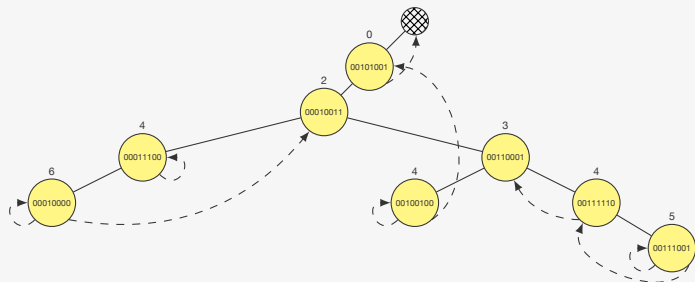
Patricia Trie



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 é o bit que deve ser usado
 - isso evita caminhos longos desnecessários

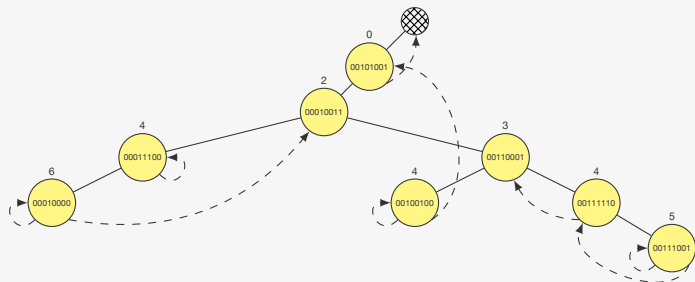
Patricia Trie



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 é o bit que deve ser usado
 - isso evita caminhos longos desnecessários
- eventualmente, subiremos na árvore

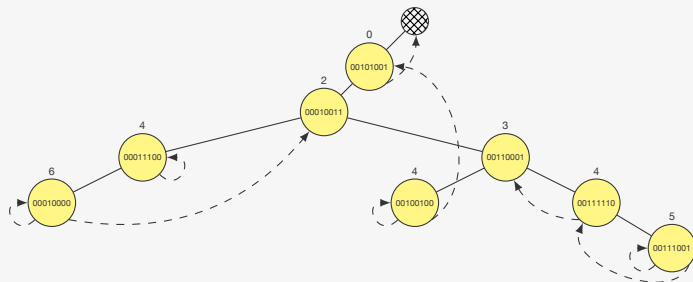
Patricia Trie



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
 - mas podemos pular alguns bits
 - cada nó sabe qual é o bit que deve ser usado
 - isso evita caminhos longos desnecessários
- eventualmente, subiremos na árvore
 - basta comparar com a chave e encerrar a busca

Patricia Trie

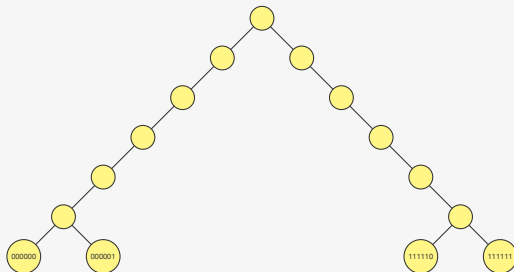


A busca é quase como numa Trie:

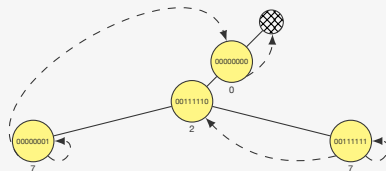
- descemos na árvore indo para a esquerda ou para a direita
- dependendo se o bit é zero ou um
 - mas podemos pular alguns bits
 - cada nó sabe qual é o bit que deve ser usado
 - isso evita caminhos longos desnecessários
- eventualmente, subiremos na árvore
 - basta comparar com a chave e encerrar a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

Patricia Trie - Comparação com Trie

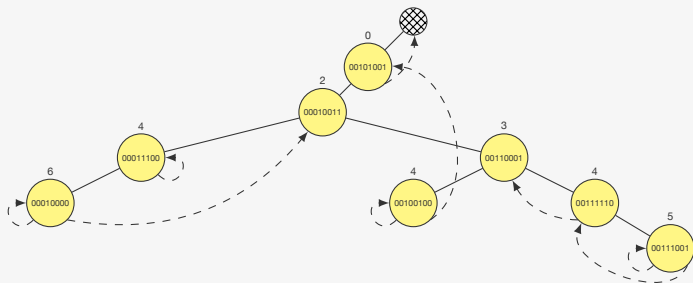
Trie



Patricia Trie

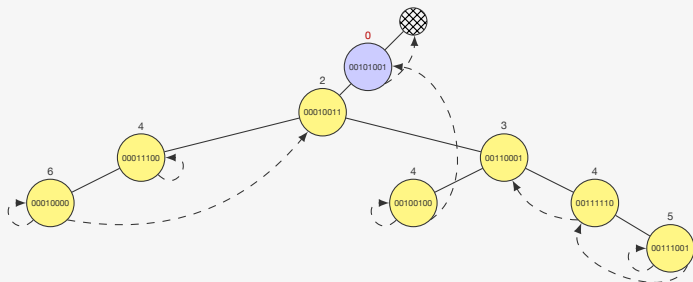


Patricia Trie - Simulação de Busca



Vamos procurar por **00101001**

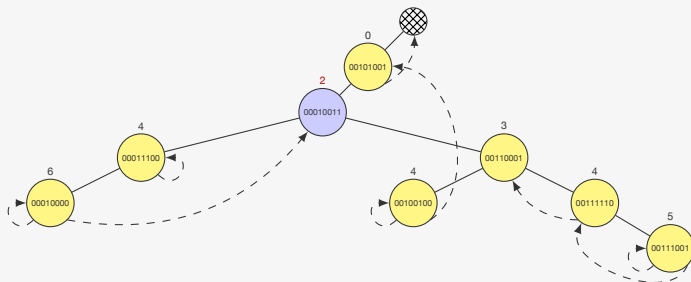
Patricia Trie - Simulação de Busca



Vamos procurar por **00101001**

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

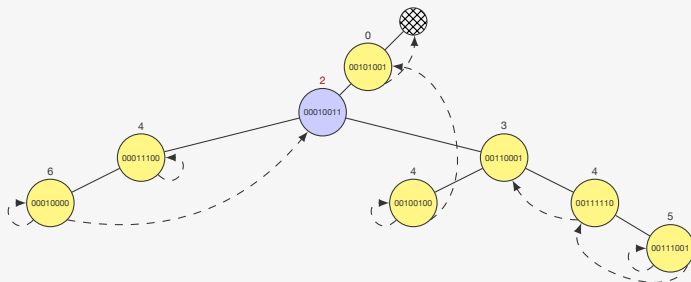
Patricia Trie - Simulação de Busca



Vamos procurar por **00101001**

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

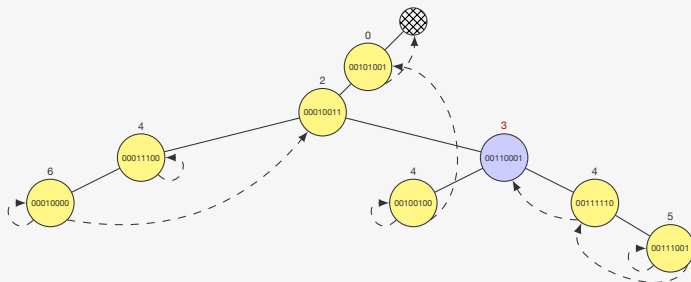
Patricia Trie - Simulação de Busca



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

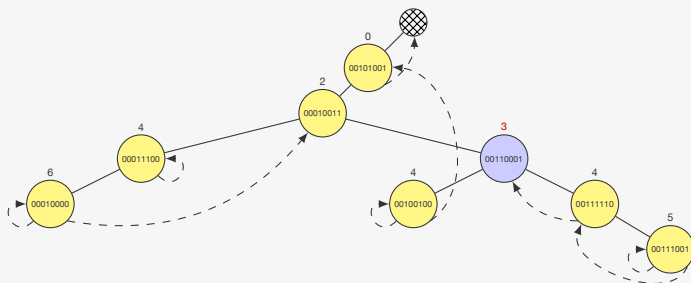
Patricia Trie - Simulação de Busca



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

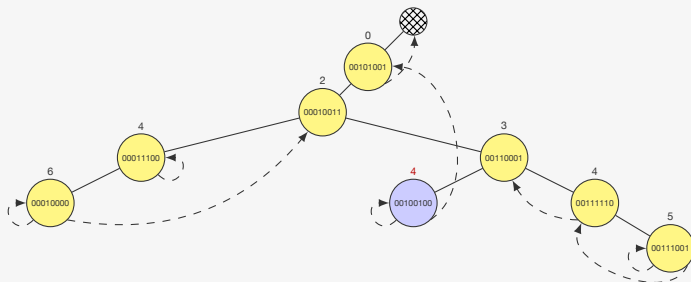
Patricia Trie - Simulação de Busca



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

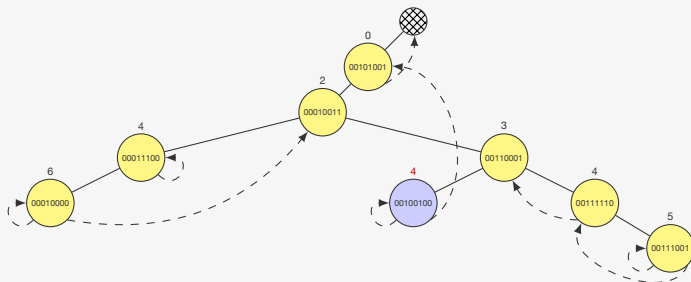
Patricia Trie - Simulação de Busca



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

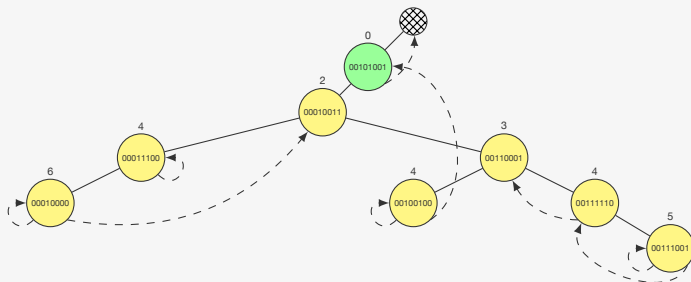
Patricia Trie - Simulação de Busca



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

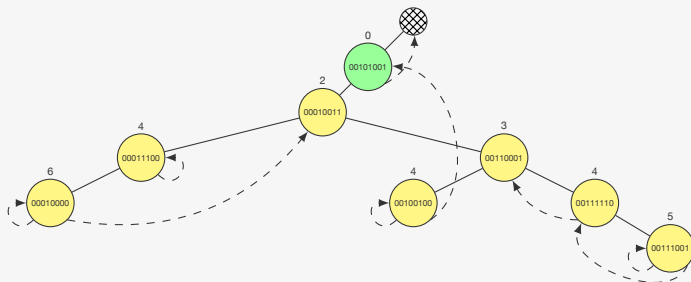
Patricia Trie - Simulação de Busca



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

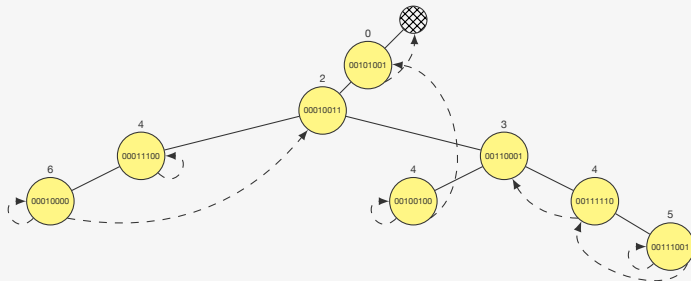
Patricia Trie - Simulação de Busca



Vamos procurar por **00101001**

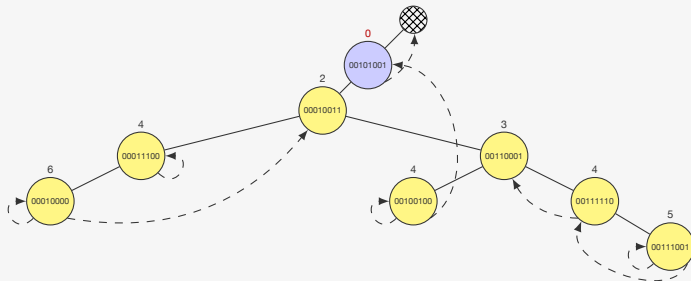
- O bit **0** de **00101001** é **0** - vá para a esquerda
- O bit **2** de **00101001** é **1** - vá para a direita
- O bit **3** de **00101001** é **0** - vá para a esquerda
- O bit **4** de **00101001** é **1** - vá para a direita
- Subiu na árvore - compare com a chave do nó atual
 - encontrou **00101001**

Patricia Trie - Simulação de Busca



Vamos procurar por **000111100**

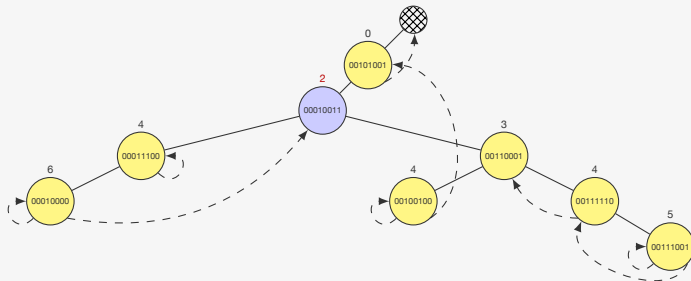
Patricia Trie - Simulação de Busca



Vamos procurar por **000111100**

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

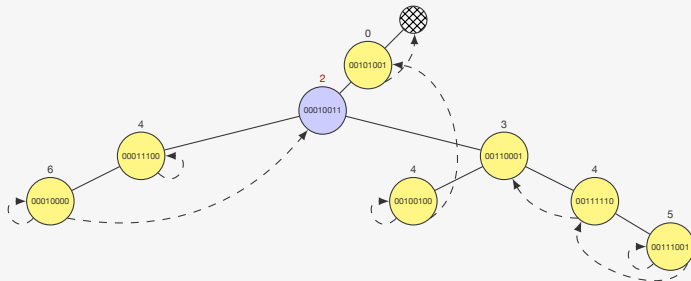
Patricia Trie - Simulação de Busca



Vamos procurar por **000111100**

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

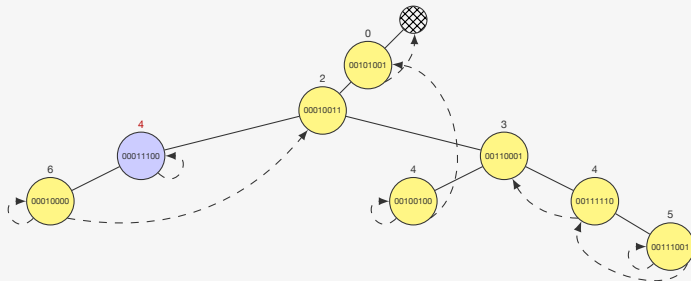
Patricia Trie - Simulação de Busca



Vamos procurar por **000111100**

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

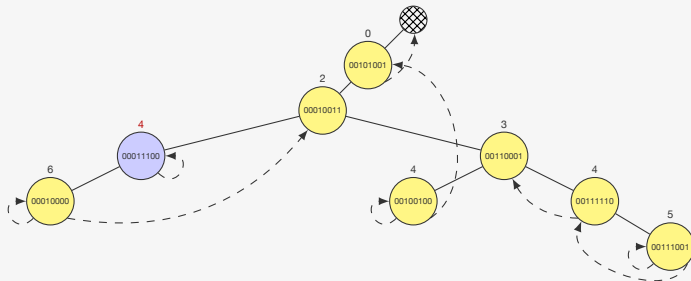
Patricia Trie - Simulação de Busca



Vamos procurar por **000111100**

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

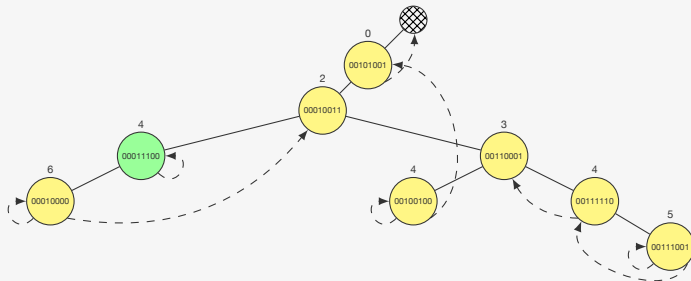
Patricia Trie - Simulação de Busca



Vamos procurar por **000111100**

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

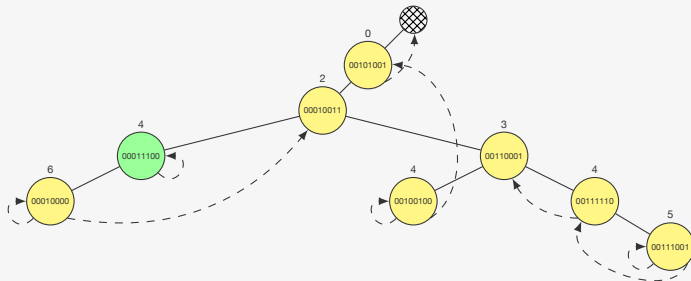
Patricia Trie - Simulação de Busca



Vamos procurar por **000111100**

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

Patricia Trie - Simulação de Busca



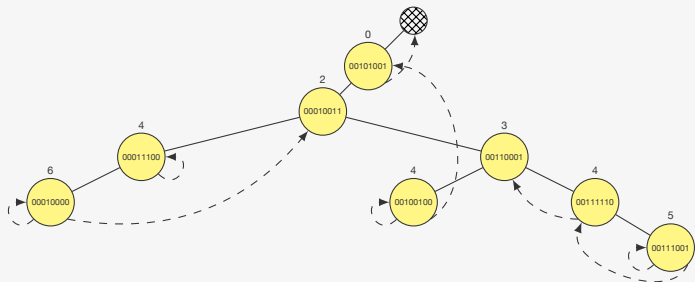
Vamos procurar por **000111100**

- O bit **0** de **00101001** é **0** - vá para a esquerda
- O bit **2** 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
 - não encontrou **000111100**

Busca - Implementação

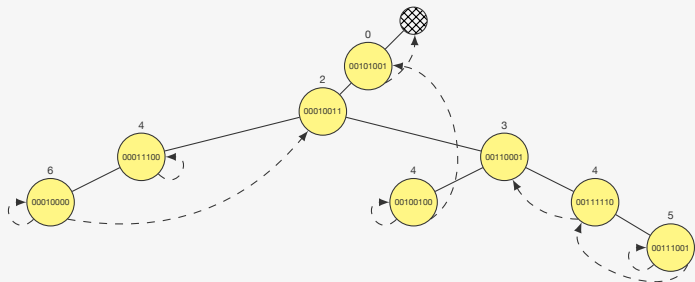
```
1 No * busca_rec(No *arvore, unsigned x, int w) {
2     if (arvore->bit <= w)
3         return arvore;
4     if (bit(x, arvore->bit) == 0)
5         return busca_rec(arvore->esq, x, arvore->bit);
6     else
7         return busca_rec(arvore->dir, x, arvore->bit);
8 }
9
10 No * busca(No *arvore, unsigned x) {
11     No *t = busca_rec(arvore->esq, x, -1);
12     return t->chave == x ? t : NULL;
13 }
```

Patricia Trie - Inserção



Para inserir 00111011:

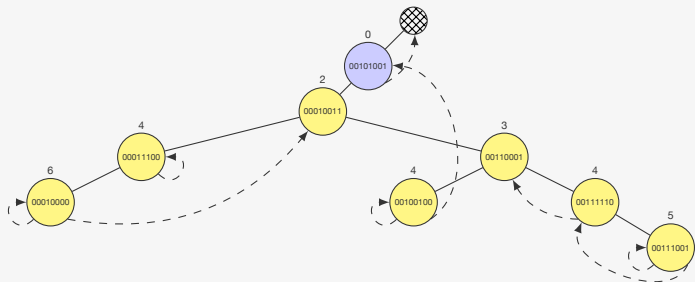
Patricia Trie - Inserção



Para inserir 00111011:

- Procuramos por 00111011

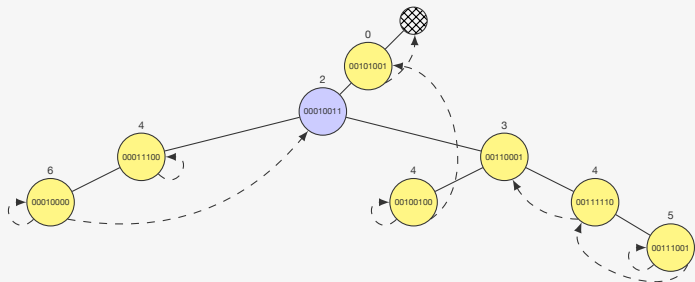
Patricia Trie - Inserção



Para inserir **00111011**:

- Procuramos por **00111011**

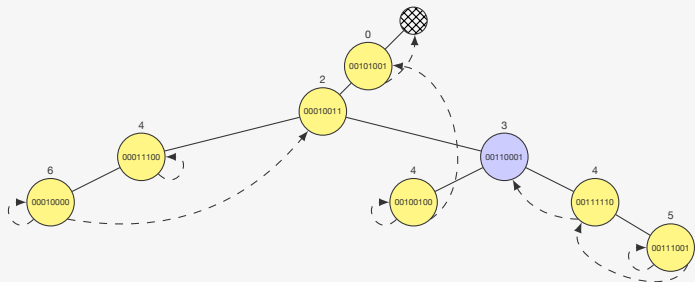
Patricia Trie - Inserção



Para inserir **00111011**:

- Procuramos por **00111011**

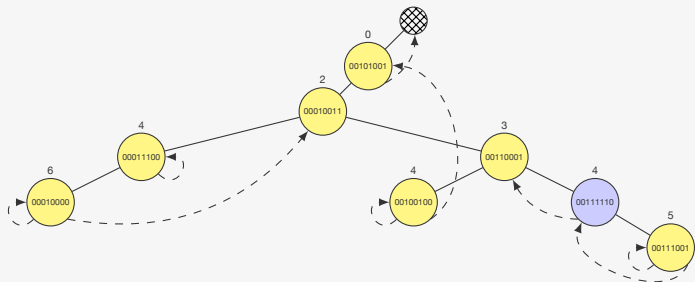
Patricia Trie - Inserção



Para inserir 00111011:

- Procuramos por 00111011

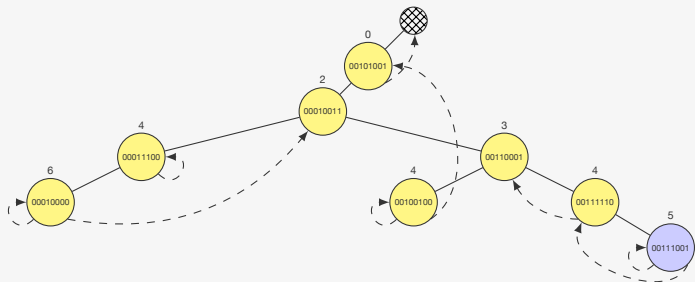
Patricia Trie - Inserção



Para inserir 00111011:

- Procuramos por 00111011

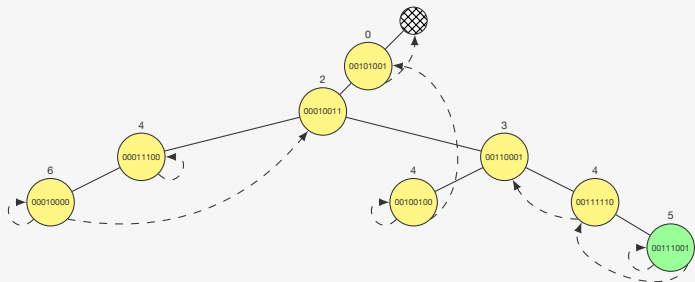
Patricia Trie - Inserção



Para inserir 00111011:

- Procuramos por 00111011

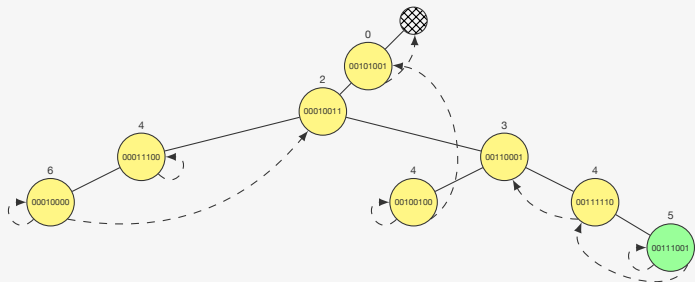
Patricia Trie - Inserção



Para inserir 00111011:

- Procuramos por 00111011
- Encontramos 00111001

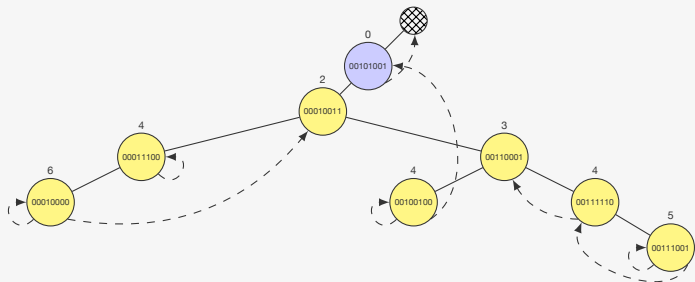
Patricia Trie - Inserção



Para inserir **00111011**:

- Procuramos por **00111011**
- Encontramos **00111001**
- Eles têm o mesmo prefixo até a posição **5**

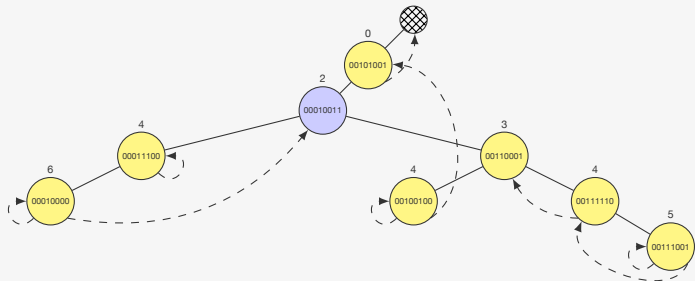
Patricia Trie - Inserção



Para inserir **00111011**:

- Procuramos por **00111011**
- Encontramos **00111001**
- Eles têm o mesmo prefixo até a posição **5**
- Fazemos uma nova busca verificando onde dividir a árvore

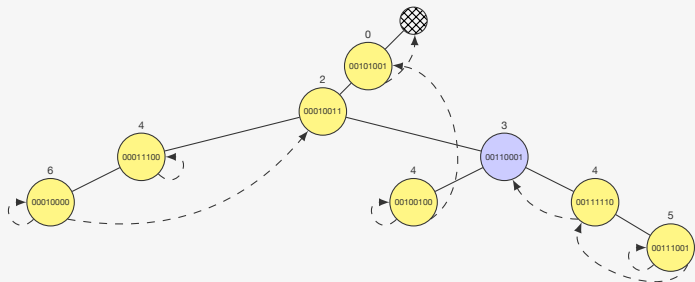
Patricia Trie - Inserção



Para inserir **00111011**:

- Procuramos por **00111011**
- Encontramos **00111001**
- Eles têm o mesmo prefixo até a posição **5**
- Fazemos uma nova busca verificando onde dividir a árvore

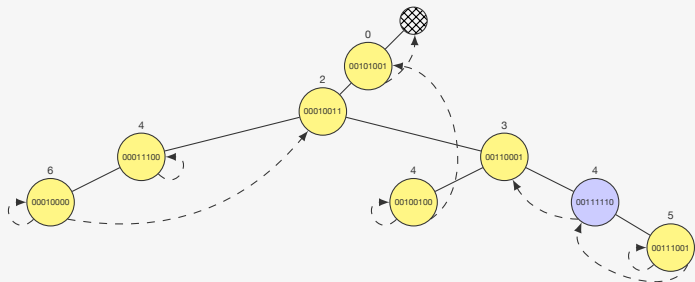
Patricia Trie - Inserção



Para inserir 00111011:

- Procuramos por 00111011
- Encontramos 00111001
- Eles têm o mesmo prefixo até a posição 5
- Fazemos uma nova busca verificando onde dividir a árvore

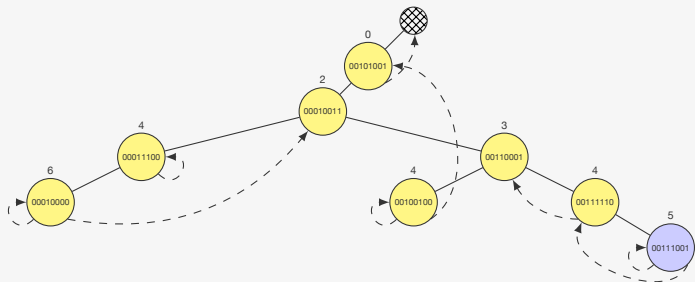
Patricia Trie - Inserção



Para inserir **00111011**:

- Procuramos por **00111011**
- Encontramos **00111001**
- Eles têm o mesmo prefixo até a posição **5**
- Fazemos uma nova busca verificando onde dividir a árvore

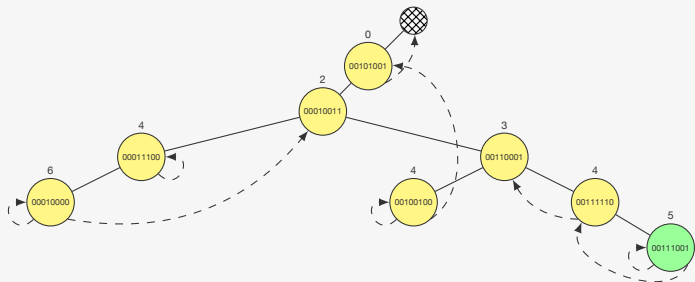
Patricia Trie - Inserção



Para inserir **00111011**:

- Procuramos por **00111011**
- Encontramos **00111001**
- Eles têm o mesmo prefixo até a posição **5**
- Fazemos uma nova busca verificando onde dividir a árvore

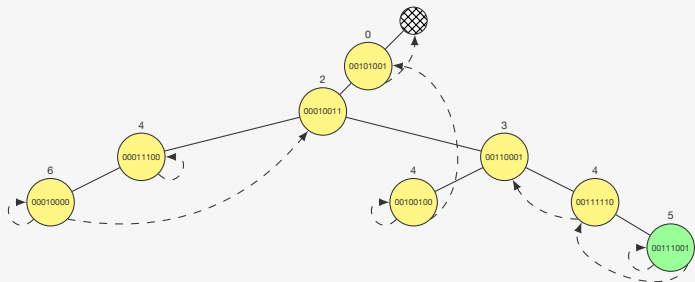
Patricia Trie - Inserção



Para inserir **00111011**:

- Procuramos por **00111011**
- Encontramos **00111001**
- Eles têm o mesmo prefixo até a posição **5**
- Fazemos uma nova busca verificando onde dividir a árvore

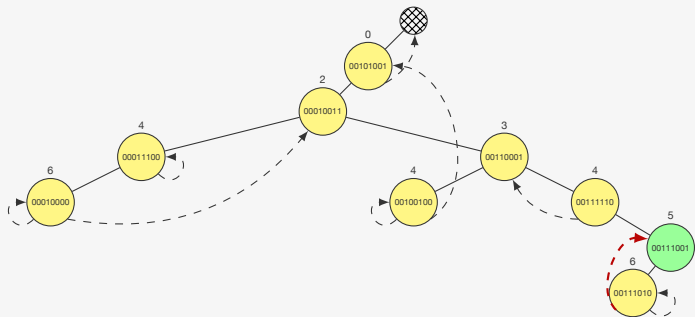
Patricia Trie - Inserção



Para inserir 00111011:

- Procuramos por 00111011
- Encontramos 00111001
- Eles têm o mesmo prefixo até a posição 5
- Fazemos uma nova busca verificando onde dividir a árvore
- Nesse caso, inserimos como uma folha

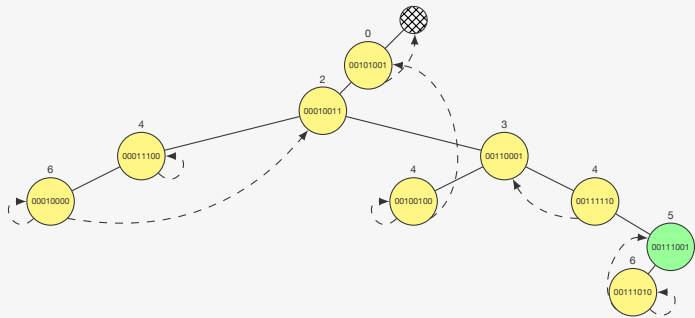
Patricia Trie - Inserção



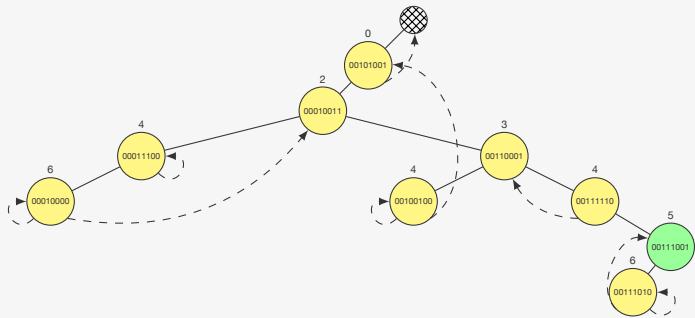
Para inserir **00111011**:

- Procuramos por **00111011**
- Encontramos **00111001**
- Eles têm o mesmo prefixo até a posição **5**
- Fazemos uma nova busca verificando onde dividir a árvore
- Nesse caso, inserimos como uma folha

Patricia Trie - Inserção

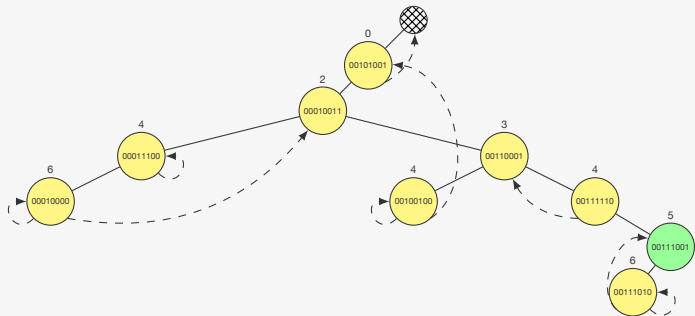


Patricia Trie - Inserção



Por que é um filho esquerdo?

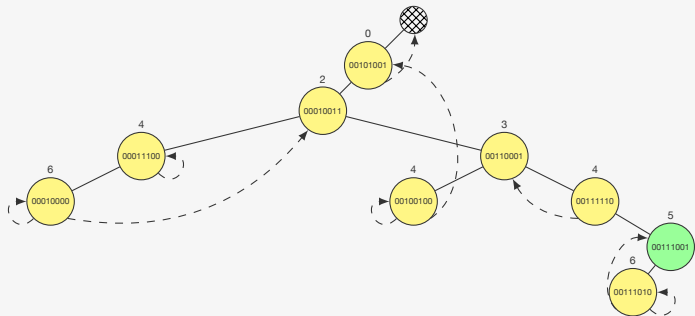
Patricia Trie - Inserção



Por que é um filho esquerdo?

- Porque o bit 5 é 0 e, por isso, a busca saiu pelo ponteiro esquerdo de 00111001

Patricia Trie - Inserção

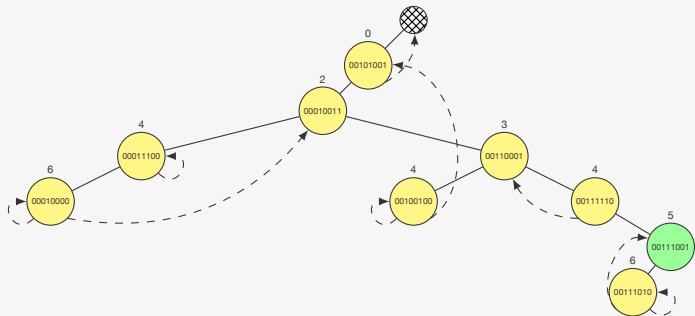


Por que é um filho esquerdo?

- Porque o bit 5 é 0 e, por isso, a busca saiu pelo ponteiro esquerdo de 00111001

Por que o ponteiro esquerdo aponta para 00111001 e o direito para 00111010?

Patricia Trie - Inserção



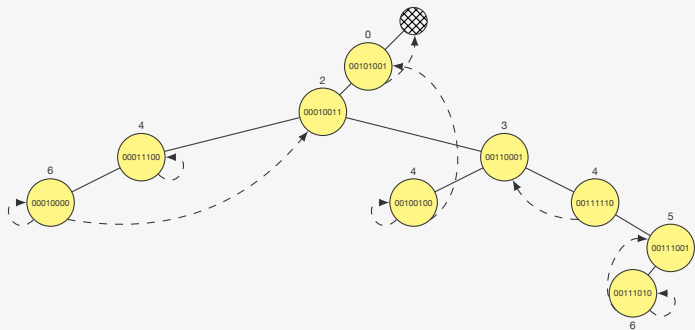
Por que é um filho esquerdo?

- Porque o bit 5 é 0 e, por isso, a busca saiu pelo ponteiro esquerdo de 00111001

Por que o ponteiro esquerdo aponta para 00111001 e o direito para 00111010?

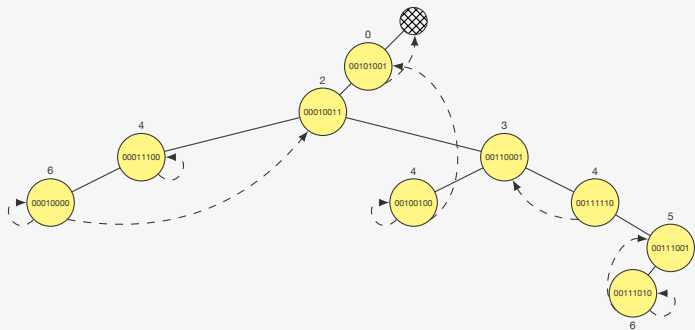
- Porque o bit 6 de 00111001 é 0 e o de 00111010 é 1

Patricia Trie - Inserção



Para inserir 000000:

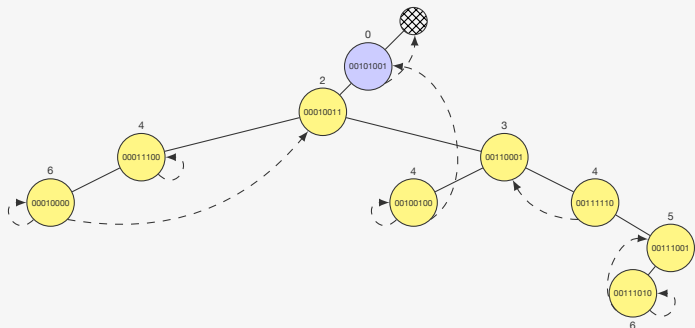
Patricia Trie - Inserção



Para inserir 000000:

- Procuramos por 000000

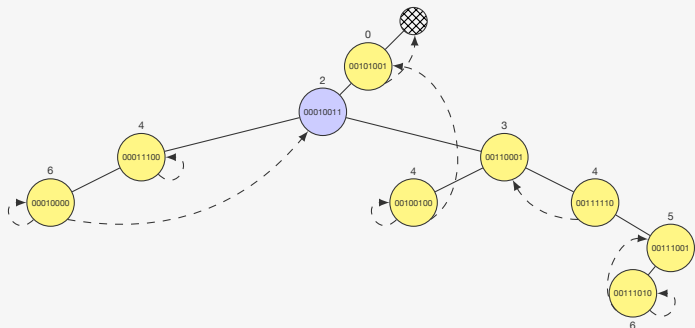
Patricia Trie - Inserção



Para inserir 000000:

- Procuramos por 000000

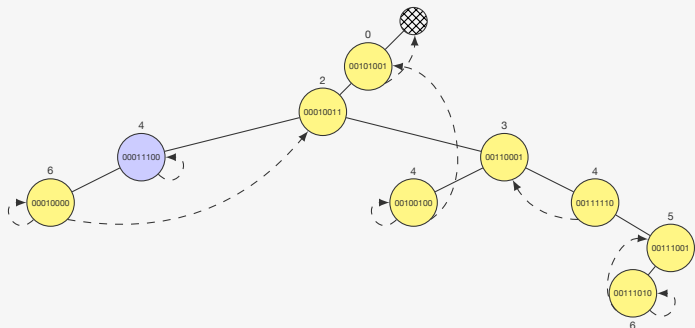
Patricia Trie - Inserção



Para inserir 000000:

- Procuramos por 000000

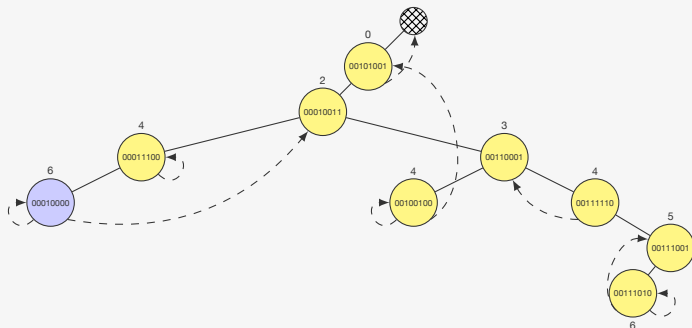
Patricia Trie - Inserção



Para inserir 000000:

- Procuramos por 000000

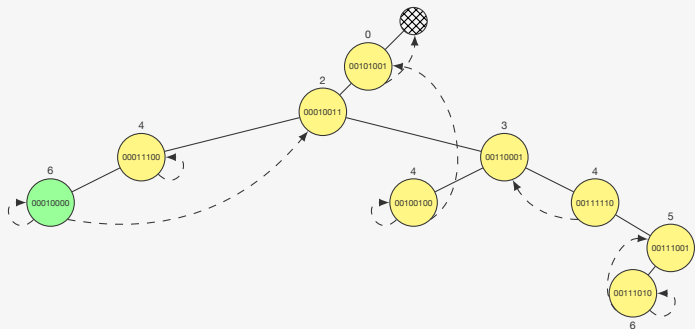
Patricia Trie - Inserção



Para inserir 000000:

- Procuramos por 000000

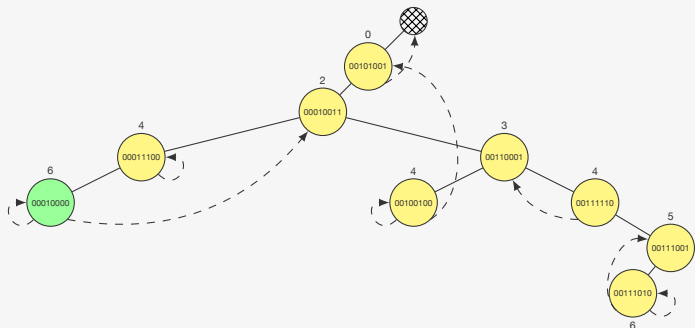
Patricia Trie - Inserção



Para inserir 000000:

- Procuramos por 000000

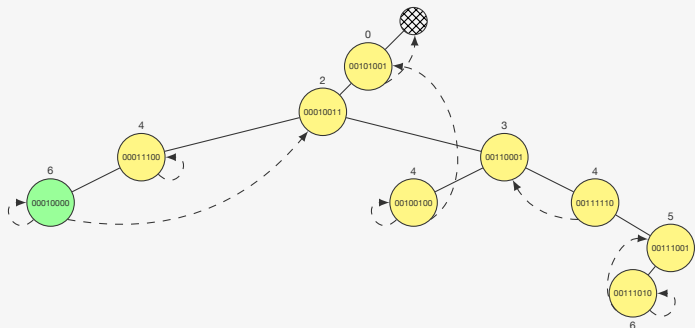
Patricia Trie - Inserção



Para inserir 000000:

- Procuramos por 000000
- Encontramos 00010000

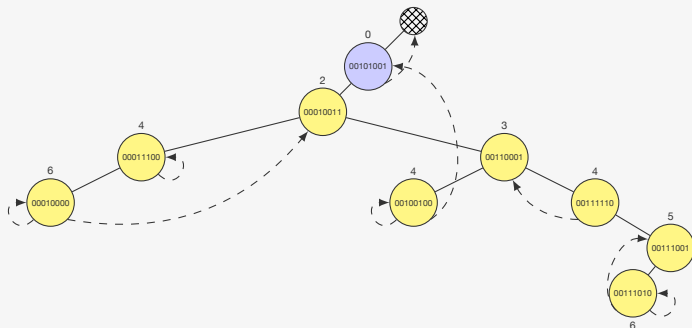
Patricia Trie - Inserção



Para inserir 000000:

- Procuramos por 000000
- Encontramos 00010000
- Eles têm o mesmo prefixo até a posição 2

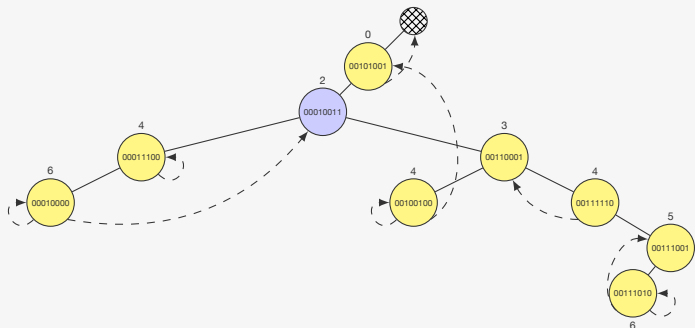
Patricia Trie - Inserção



Para inserir **000000**:

- Procuramos por **000000**
- Encontramos **00010000**
- Eles têm o mesmo prefixo até a posição **2**
- Fazemos uma nova busca por **000000**

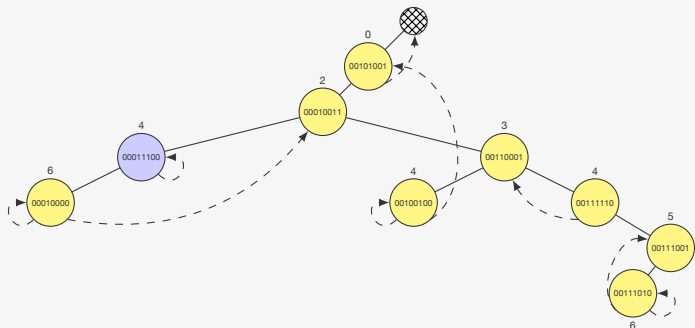
Patricia Trie - Inserção



Para inserir 000000:

- Procuramos por 000000
- Encontramos 00010000
- Eles têm o mesmo prefixo até a posição 2
- Fazemos uma nova busca por 000000

Patricia Trie - Inserção



Para inserir **000000**:

- Procuramos por **000000**
- Encontramos **00010000**
- Eles têm o mesmo prefixo até a posição **2**
- Fazemos uma nova busca por **000000**
- Precisamos diferenciar **000000** de **00010000** no bit **3**

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai);
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai);
```

```
1 void insere(No **arvore, unsigned chave) {
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai);
```

```
1 void insere(No **arvore, unsigned chave) {  
2   int i;  
3   No *t = busca_rec((*arvore)->esq, chave, -1);
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai);
```

```
1 void insere(No **arvore, unsigned chave) {  
2     int i;  
3     No *t = busca_rec((*arvore)->esq, chave, -1);  
4     if (chave == t->chave)  
5         return;
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai);
```

```
1 void insere(No **arvore, unsigned chave) {  
2     int i;  
3     No *t = busca_rec((*arvore)->esq, chave, -1);  
4     if (chave == t->chave)  
5         return;  
6     for (i = 0; bit(chave, i) == bit(t->chave, i); i++) ;
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai);
```

```
1 void insere(No **arvore, unsigned chave) {  
2     int i;  
3     No *t = busca_rec((*arvore)->esq, chave, -1);  
4     if (chave == t->chave)  
5         return;  
6     for (i = 0; bit(chave, i) == bit(t->chave, i); i++) ;  
7     (*arvore)->esq = insere_rec((*arvore)->esq, chave, i, *arvore);  
8 }
```


Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai) {
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai) {  
2     No *novo;  
3     if ((arvore->bit >= w) || (arvore->bit <= pai->bit)) {
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai) {
2     No *novo;
3     if ((arvore->bit >= w) || (arvore->bit <= pai->bit)) {
4         novo = malloc(sizeof(No));
5         novo->chave = chave;
6         novo->bit = w;
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai) {
2     No *novo;
3     if ((arvore->bit >= w) || (arvore->bit <= pai->bit)) {
4         novo = malloc(sizeof(No));
5         novo->chave = chave;
6         novo->bit = w;
7         if (bit(chave, novo->bit) == 1) {
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai) {
2     No *novo;
3     if ((arvore->bit >= w) || (arvore->bit <= pai->bit)) {
4         novo = malloc(sizeof(No));
5         novo->chave = chave;
6         novo->bit = w;
7         if (bit(chave, novo->bit) == 1) {
8             novo->esq = arvore;
9             novo->dir = novo;
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai) {
2     No *novo;
3     if ((arvore->bit >= w) || (arvore->bit <= pai->bit)) {
4         novo = malloc(sizeof(No));
5         novo->chave = chave;
6         novo->bit = w;
7         if (bit(chave, novo->bit) == 1) {
8             novo->esq = arvore;
9             novo->dir = novo;
10        } else {
11            novo->esq = novo;
12            novo->dir = arvore;
13        }
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai) {
2     No *novo;
3     if ((arvore->bit >= w) || (arvore->bit <= pai->bit)) {
4         novo = malloc(sizeof(No));
5         novo->chave = chave;
6         novo->bit = w;
7         if (bit(chave, novo->bit) == 1) {
8             novo->esq = arvore;
9             novo->dir = novo;
10        } else {
11            novo->esq = novo;
12            novo->dir = arvore;
13        }
14        return novo;
15    }
```

Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai) {
2   No *novo;
3   if ((arvore->bit >= w) || (arvore->bit <= pai->bit)) {
4     novo = malloc(sizeof(No));
5     novo->chave = chave;
6     novo->bit = w;
7     if (bit(chave, novo->bit) == 1) {
8       novo->esq = arvore;
9       novo->dir = novo;
10    } else {
11      novo->esq = novo;
12      novo->dir = arvore;
13    }
14    return novo;
15  }
16  if (bit(chave, arvore->bit) == 0)
17    arvore->esq = insere_rec(arvore->esq, chave, w, arvore);
18  else
19    arvore->dir = insere_rec(arvore->dir, chave, w, arvore);
```


Inserção - Implementação

```
1 No * insere_rec(No *arvore, unsigned chave, int w, No *pai) {
2   No *novo;
3   if ((arvore->bit >= w) || (arvore->bit <= pai->bit)) {
4     novo = malloc(sizeof(No));
5     novo->chave = chave;
6     novo->bit = w;
7     if (bit(chave, novo->bit) == 1) {
8       novo->esq = arvore;
9       novo->dir = novo;
10    } else {
11      novo->esq = novo;
12      novo->dir = arvore;
13    }
14    return novo;
15  }
16  if (bit(chave, arvore->bit) == 0)
17    arvore->esq = insere_rec(arvore->esq, chave, w, arvore);
18  else
19    arvore->dir = insere_rec(arvore->dir, chave, w, arvore);
20  return arvore;
21 }
```

Conclusão

Árvores Digitais de Busca:

Conclusão

Árvores Digitais de Busca:

- Altura $O(k)$ onde k é o número de bits das chaves

Conclusão

Árvores Digitais de Busca:

- Altura $O(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $O(k)$

Conclusão

Árvores Digitais de Busca:

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

Conclusão

Árvores Digitais de Busca:

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

Tries:

Conclusão

Árvores Digitais de Busca:

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

Tries:

- Resolvem o problema das árvores digitais de busca

Conclusão

Árvores Digitais de Busca:

- Altura $O(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $O(k)$
- 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 $O(k)$
- 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:

Conclusão

Árvores Digitais de Busca:

- Altura $O(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $O(k)$
- 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:

- Economizam espaço e tempo

Conclusão

Árvores Digitais de Busca:

- Altura $O(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $O(k)$
- 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:

- Economizam espaço e tempo
- Podem ser usadas para indexar chaves de tamanho variável

Conclusão

Árvores Digitais de Busca:

- Altura $O(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $O(k)$
- 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:

- Economizam espaço e tempo
- Podem ser usadas para indexar chaves de tamanho variável
- Podem ser usadas em Radix maiores que 2

Conclusão

Árvores Digitais de Busca:

- Altura $O(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $O(k)$
- 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:

- Economizam espaço e tempo
- Podem ser usadas para indexar chaves de tamanho variável
- Podem ser usadas em Radix maiores que 2
 - Por exemplo, 256 para indexar palavras

Conclusão

Árvores Digitais de Busca:

- Altura $O(k)$ onde k é o número de bits das chaves
- Isto é, algoritmos rodam em $O(k)$
- 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:

- Economizam espaço e tempo
- 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...