# AULA 20
# ESTRUTURA DE DADOS

## Árvores N-árias

Norton T. Roman & Luciano A. Digiampietri

# Árvores N-árias

**Imagine que tenhamos um dicionário em mãos, e queiramos fazer consultas**

# Árvores N-árias

Imagine que tenhamos um dicionário em mãos, e queiramos fazer consultas

Como faríamos com as buscas no dicionário?

# Árvores N-árias

Imagine que tenhamos um dicionário em mãos, e queiramos fazer consultas

Como faríamos com as buscas no dicionário?

- Uma vez que as palavras estão ordenadas, poderíamos usar busca binária

# Árvores N-árias

Imagine que tenhamos um dicionário em mãos, e queiramos fazer consultas

Como faríamos com as buscas no dicionário?

- Uma vez que as palavras estão ordenadas, poderíamos usar busca binária
- Ainda assim, quanto será que isso poderia demorar?

# Árvores N-árias

**Nos dicionários online hoje há coisa de 200.000 palavras**

# Árvores N-árias

**Nos dicionários online hoje há coisa de 200.000 palavras**

**A busca binária faz, no pior caso, $log_2 n$ comparações, em que $n$ é o número de dados da base**

# Árvores N-árias

**Nos dicionários online hoje há coisa de 200.000 palavras**

**A busca binária faz, no pior caso, $log_2 n$ comparações, em que $n$ é o número de dados da base**

> **Em um dicionário online, isso significa um máximo de 18 comparações. Nada mau.**

# Árvores N-árias

**Contudo, palavras não são comparadas em uma única operação**

# Árvores N-árias

**Contudo, palavras não são comparadas em uma única operação**

- **São comparadas letra a letra**

# Árvores N-árias

**Contudo, palavras não são comparadas em uma única operação**

- **São comparadas letra a letra**

**Então, se as palavras tiverem, em média, 5 letras, o total de comparações já sobe para** $18 \times 5 = 90$

# Árvores N-árias

**Contudo, palavras não são comparadas em uma única operação**

- **São comparadas letra a letra**

**Então, se as palavras tiverem, em média, 5 letras, o total de comparações já sobe para** $18 \times 5 = 90$

- **Ainda sem grandes crises... a menos que façamos isso inúmeras vezes...**

# Árvores N-árias

**E se conseguíssemos efetuar, no máximo, um número de comparações igual ao número de letras da palavra buscada?**

# Árvores N-árias

**E se conseguíssemos efetuar, no máximo, um número de comparações igual ao número de letras da palavra buscada?**

**5 letras $\rightarrow$ 5 comparações**

# Árvores N-árias

**E se conseguíssemos efetuar, no máximo, um número de comparações igual ao número de letras da palavra buscada?**

**5 letras $\rightarrow$ 5 comparações**

**Será possível?**

# Árvores N-árias

**E se conseguíssemos efetuar, no máximo, um número de comparações igual ao número de letras da palavra buscada?**

**5 letras $\rightarrow$ 5 comparações**

**Será possível?**

**Com a estrutura de dados correta, sim**

# Trie

**Uma trie (de *retrieval*), é uma árvore n-ária projetada para recuperação rápida de chaves de busca**

# Trie

**Uma trie (de *re**trie**val*), é uma árvore n-ária projetada para recuperação rápida de chaves de busca**

**Características:**

# Trie

**Uma trie (de *retrieval*), é uma árvore n-ária projetada para recuperação rápida de chaves de busca**

**Características:**

- **Ela não armazena nenhuma chave explicitamente.**
- **Chaves são codificadas nos caminhos a partir da raiz**

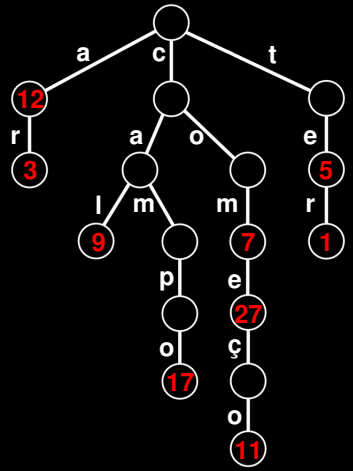
# Trie

**Uma trie (de *retrieval*), é uma árvore n-ária projetada para recuperação rápida de chaves de busca**

**Características:**

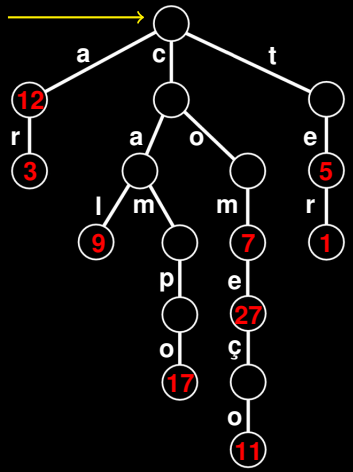
- **Ela não armazena nenhuma chave explicitamente.**
- **Chaves são codificadas nos caminhos a partir da raiz**
- **Todo nó tem um campo valor, retornado caso a chave termine nesse nó**

# Trie – Estrutura



# Trie – Estrutura

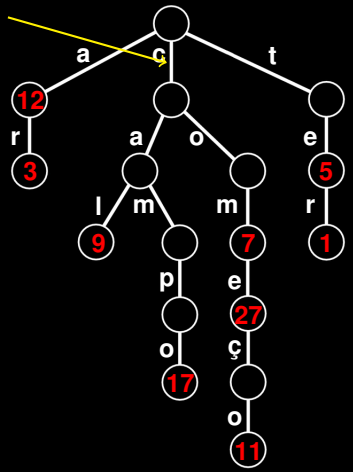
**Raiz**



# Trie – Estrutura

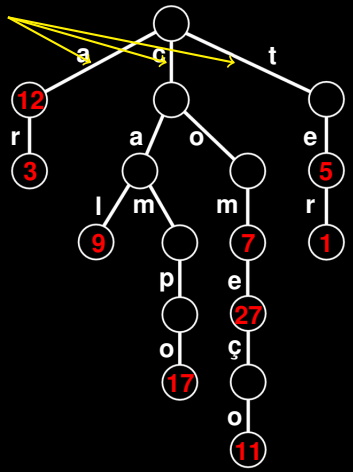
**Raiz**

**Caminho que leva à sub-trie com todas as chaves que começam com c**



# Trie – Estrutura

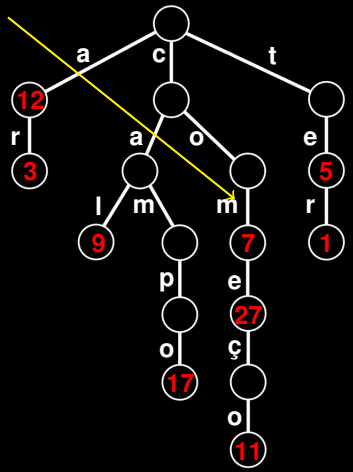
**Deve haver um caminho diferente para cada possível caractere inicial da chave**



# Trie – Estrutura

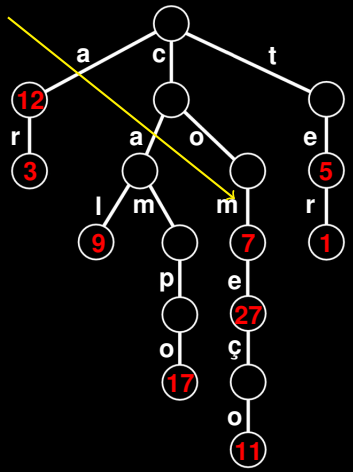
**Caminho que leva à sub-trie com todas as chaves que começam com *com***

***com*, *come* e *começo***



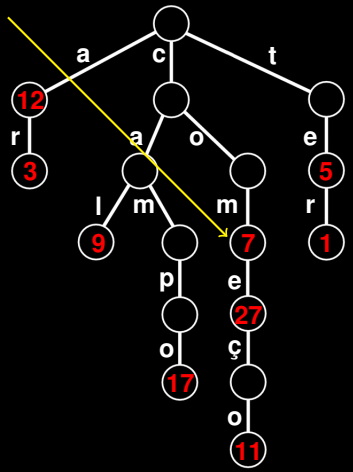
# Trie – Estrutura

**Note que cada nó no nível *i* representa o conjunto de todas as chaves que começam com a mesma sequência de *i* símbolos**



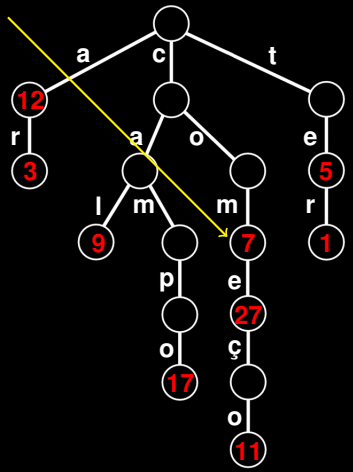
# Trie – Estrutura

**Valor de retorno para a chave *com***



# Trie – Estrutura

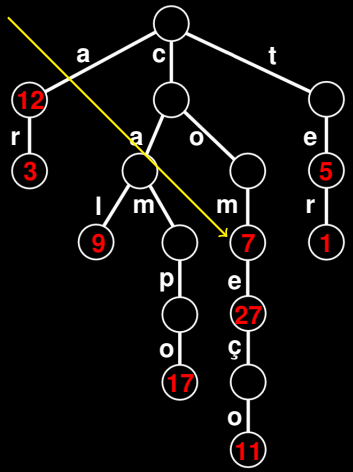
**É assim que sabemos se uma chave é válida**



# Trie – Estrutura

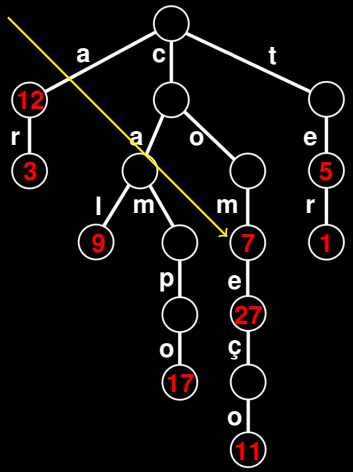
É assim que sabemos se uma chave é válida

- Se o caminho correspondente terminar em um valor válido



# Trie – Estrutura

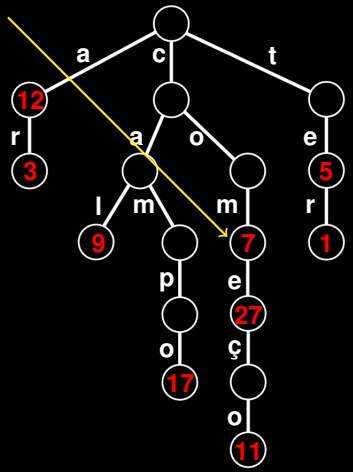
**E o que significa esse valor?**



# Trie – Estrutura

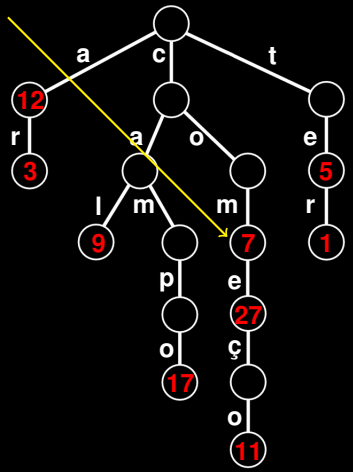
**E o que significa esse valor?**

**Pode ser qualquer coisa...**



# Trie – Estrutura

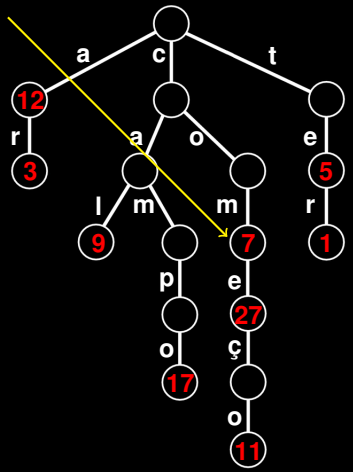
**Pode ser a posição em um arranjo com o significado da palavra**



# Trie – Estrutura

Pode ser a posição em um arranjo com o significado da palavra

Ou um booleano indicando tão somente que a palavra existe



# Trie – Implementação

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define true 1
#define false 0
#define N_ALFABETO 26

typedef int bool;
typedef bool TIPORET;
```

```
typedef struct no {
  struct no *filhos[N_ALFABETO];
  TIPORET fim;
} NO;

typedef NO* PONT;
```

# Trie – Implementação

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define true 1
#define false 0
#define N_ALFABETO 26

typedef int bool;
typedef bool TIPORET;
```

```c
typedef struct no {
  struct no *filhos[N_ALFABETO];
  TIPORET fim;
} NO;

typedef NO* PONT;
```

**Nesse exemplo, trataremos somente de letras minúsculas e sem acento**

# Trie – Implementação

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define true 1
#define false 0
#define N_ALFABETO 26

typedef int bool;
typedef bool TIPORET;
```

```c
typedef struct no {
  struct no *filhos[N_ALFABETO];
  TIPORET fim;
} NO;

typedef NO* PONT;
```

**Todo nó carrega ponteiros para potenciais *N_ALFABETO* subárvores**

# Trie – Implementação

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define true 1
#define false 0
#define N_ALFABETO 26

typedef int bool;
typedef bool TIPORET;
```

```c
typedef struct no {
  struct no *filhos[N_ALFABETO];
  TIPORET fim;
} NO;

typedef NO* PONT;
```

***fim*** **é um booleano que indica se chegamos ao fim de uma chave válida**

# Trie – Inicialização

```
PONT criaNo() {
  PONT p = NULL;

  p = (PONT)malloc(sizeof(NO));
  if (p) {
    p->fim = false;
    int i;
    for (i=0; i<N_ALFABETO; i++)
      p->filhos[i] = NULL;
  }
  return(p);
}
```

```
PONT inicializa() {
  return(criaNo());
}

int main() {
  PONT r = inicializa();
}
```

# Trie – Inicialização

```
PONT criaNo() {
  PONT p = NULL;

  p = (PONT)malloc(sizeof(NO));
  if (p) {
    p->fim = false;
    int i;
    for (i=0; i<N_ALFABETO; i++)
      p->filhos[i] = NULL;
  }
  return(p);
}
```

```
PONT inicializa() {
  return(criaNo());
}

int main() {
  PONT r = inicializa();
}
```

**Criamos um nó sem filhos na memória...**

# Trie – Inicialização

```
PONT criaNo() {
  PONT p = NULL;

  p = (PONT)malloc(sizeof(NO));
  if (p) {
    p->fim = false;
    int i;
    for (i=0; i<N_ALFABETO; i++)
      p->filhos[i] = NULL;
  }
  return(p);
}
```

```
PONT inicializa() {
  return(criaNo());
}

int main() {
  PONT r = inicializa();
}
```

**... e que não é o fim de uma chave**

# Trie – Mapeamento

**O grande ganho da trie vem do fato de conseguirmos mapear diretamente um símbolo da chave a uma posição do arranjo de filhos no nó**

# Trie – Mapeamento

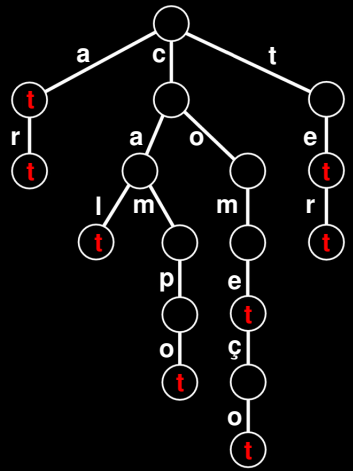
**O grande ganho da trie vem do fato de conseguirmos mapear diretamente um símbolo da chave a uma posição do arranjo de filhos no nó**

**No nosso caso, o mapeamento das letras minúsculas é direto:**

```
int mapearIndice(char c) {
  return((int)c - (int)'a');
}
```

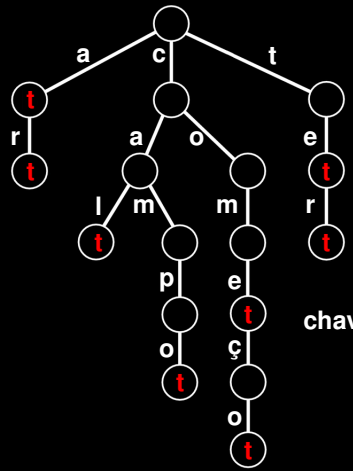
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



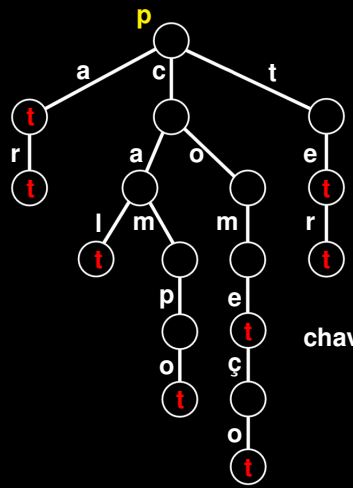
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



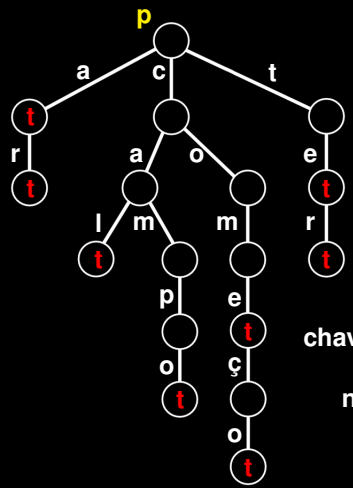
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



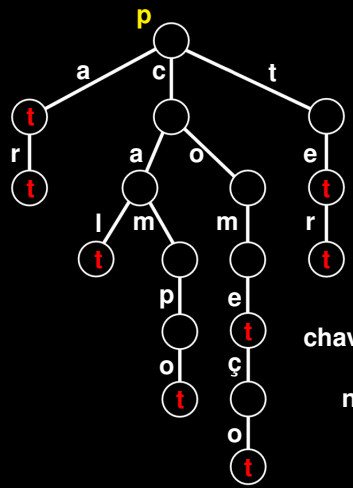
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



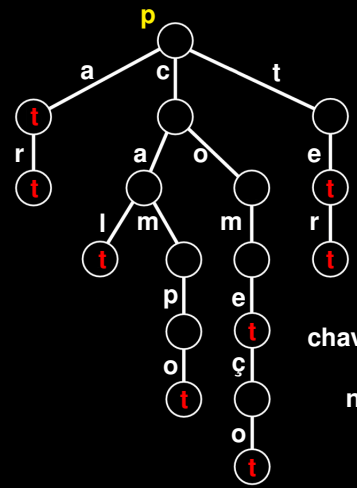
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



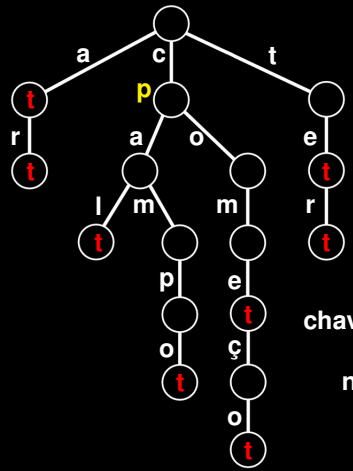
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```

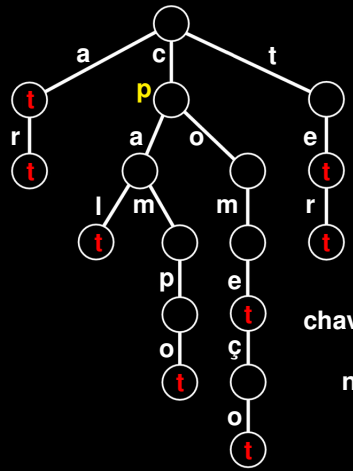



chave: “com”

nivel: 0

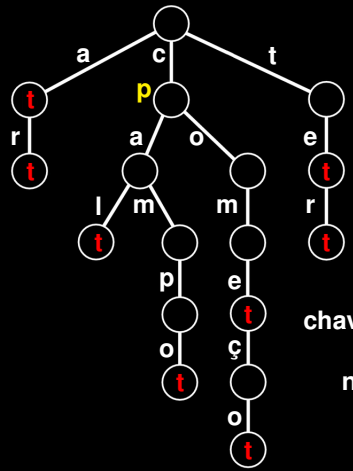
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```

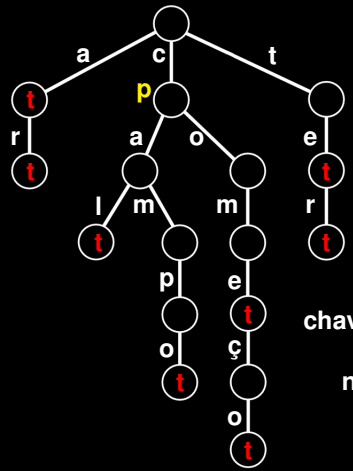



chave: “com”

nivel: 1

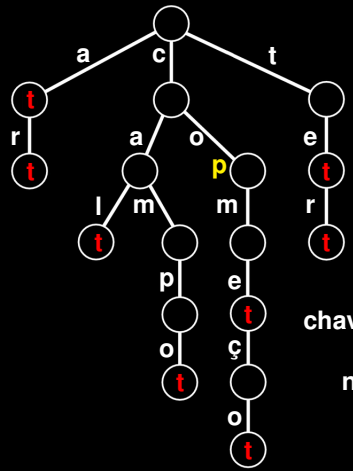
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



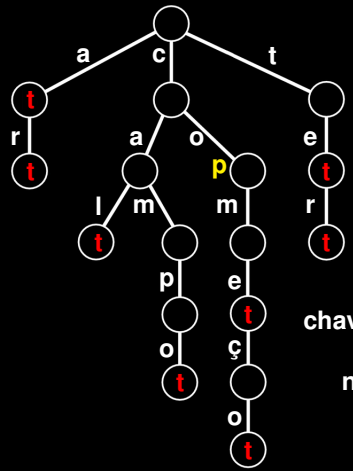
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



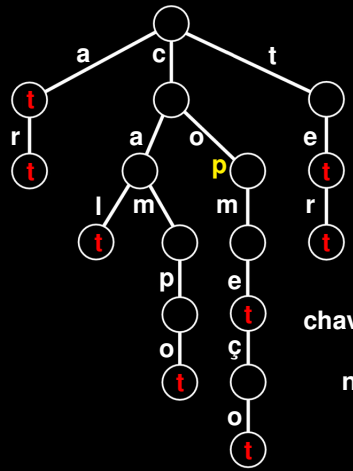
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



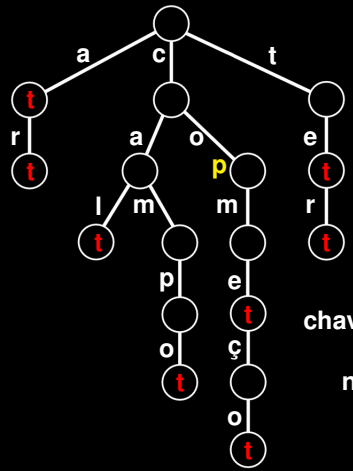
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



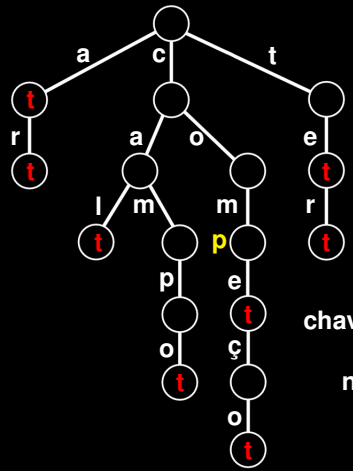
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



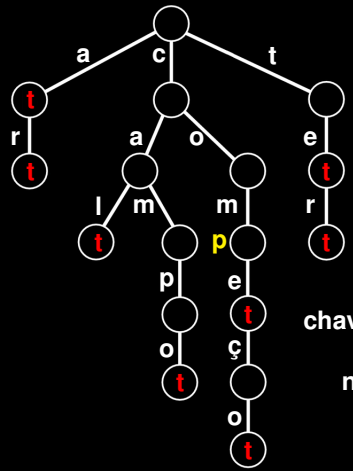
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



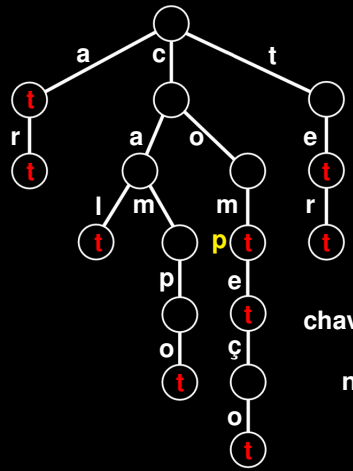
# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



# Trie – Inserção

```
void insere(PONT raiz, char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;

  PONT p = raiz;
  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i])
      p->filhos[i] = criaNo();
    p = p->filhos[i];
  }
  p->fim = true;
}
```



# Trie – Busca

```
bool busca(PONT raiz,char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;
  PONT p = raiz;

  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i]) return(false);
    p = p->filhos[i];
  }
  return(p && p->fim);
}
```

**Busca é semelhante à inserção...**

# Trie – Busca

```
bool busca(PONT raiz,char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;
  PONT p = raiz;

  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i]) return(false);
    p = p->filhos[i];
  }
  return(p && p->fim);
}
```

**Busca é semelhante à inserção... exceto que se um nó no caminho não existir, a busca falha**

# Trie – Busca

```
bool busca(PONT raiz,char *chave) {
  int nivel;
  int compr = strlen(chave);
  int i;
  PONT p = raiz;

  for (nivel=0; nivel<compr; nivel++) {
    i = mapearIndice(chave[nivel]);
    if (!p->filhos[i]) return(false);
    p = p->filhos[i];
  }
  return(p && p->fim);
}
```

**E no final sempre verificamos se o nó representa o fim de uma chave**

# AULA 20
# ESTRUTURA DE DADOS

## Árvores N-árias

Norton T. Roman & Luciano A. Digiampietri