Estruturas de Dados

Aula 18 — Tries e Indexação de Texto

Prof Ana Carolina Sokolonski

Bacharelado em Sistemas de Informação
Instituto Federal da Bahia — Campus Feira de Santana

2026

Agenda

O que é uma Trie - Árvore de Prefixos?

Inserção e Busca

Aplicações

Árvore de Prefixo vs. Tabela Hash

Exercícios

O que é uma Trie - Árvore de Prefixos?

Árvore de prefixos para armazenar texto

Motivação: o problema

Cenário

Precisamos armazenar um **conjunto de palavras** e responder, rapidamente:

- A palavra w está no conjunto?
- Quais palavras começam com o prefixo p?
- Qual é a próxima letra possível, dado o que já foi digitado?

Por que não usar tabela hash?

Uma tabela hash responde bem a "w está no conjunto?", mas **não tem noção de ordem nem de prefixo**: não há busca eficiente por "chaves que começam com `"ca"`".

Ideia da Trie

Organizar as palavras como um **caminho na árvore**, um nó por caractere. Palavras com o mesmo prefixo **compartilham** o mesmo caminho inicial – o prefixo é armazenado uma única vez.

Origem do nome

"Trie" vem de re**trie**val (recuperação), proposto por Edward Fredkin (1960). Também chamada de *árvore de prefixos* (*prefix tree*) ou *digital tree*.

Estrutura de uma Trie

Definição

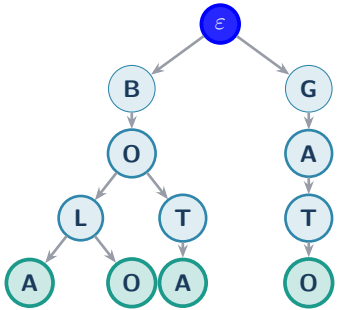
Uma Trie é uma árvore n-ária em que:

- Cada **aresta** representa um caractere.
- Cada **nó** representa um prefixo (a concatenação dos caracteres do caminho desde a raiz).
- Um nó é marcado como **fim de palavra** se o prefixo até ele é uma palavra completa do conjunto.
- A **raiz** representa a string vazia.

Compartilhamento de prefixos

As palavras `BOLA`, `BOLO` e `BOTA` compartilham o prefixo `BO`: ele é armazenado **uma única vez** na Trie.

Trie para {BOLA, BOLO, BOTA, GATO}



Verde = fim de palavra Azul escuro = raiz (ε)

Inserção e Busca

Percorrendo a árvore caractere por caractere

Representação do Nó

Vetor de filhos (alfabeto fixo)

Para alfabetos pequenos (ex.: 26 letras minúsculas), cada nó guarda um **vetor de ponteiros**, um por caractere possível:

- Acesso ao filho: $\mathcal{O}(1)$
- Custo de espaço: $\mathcal{O}(|\Sigma|)$ por nó, mesmo com poucos filhos usados

Alternativa: mapa de filhos

Para alfabetos grandes (Unicode, por exemplo), usar uma **tabela hash ou lista** de filhos por nó economiza espaço, ao custo de um acesso um pouco mais caro que $\mathcal{O}(1)$.

```c
#define ALFABETO 26

typedef struct TrieNo {
    struct TrieNo *filhos[ALFABETO];
    int fimDePalavra;//1 se é fim de
        palavra
} TrieNo;

TrieNo* criarNo(void) {
    TrieNo *no = (TrieNo*) malloc(
        sizeof(TrieNo));
    no->fimDePalavra = 0;
    for (int i = 0; i < ALFABETO; i++)
        no->filhos[i] = NULL;
    return no;
}
```

Índice do filho = `letra - 'a'`

Inserção: Algoritmo

Ideia

Percorrer a palavra caractere a caractere, a partir da raiz:

1. Para cada caractere c, se o filho correspondente não existir, **criar** um novo nó.
2. Avançar para esse filho.
3. Ao final da palavra, marcar o nó atual como **fim de palavra**.

```
1  void inserir(TrieNo *raiz, const char
       *palavra) {
2    TrieNo *atual = raiz;
3    for(int i=0;palavra[i]!='\0';i++){
4      int idx = palavra[i] - 'a';
5      if (atual->filhos[idx] == NULL)
6        atual->filhos[idx] = criarNo();
7      atual = atual->filhos[idx];
8    }
9    atual->fimDePalavra = 1;
10 }
```

Complexidade

Inserir uma palavra de tamanho L: $\mathcal{O}(L)$ – **independente** do número de palavras já armazenadas.

Busca de Palavra Exata

Ideia

Percorrer os caracteres como na inserção; se algum filho não existir, a palavra **não está** no conjunto. Ao chegar ao fim, a palavra só está presente se o nó final tiver `fimDePalavra = 1`.

Cuidado

Chegar ao fim do caminho **não** significa que a palavra existe: pode ser apenas um **prefixo** de outra palavra armazenada (ex.: `"BO"` é prefixo de `BOLA`, mas não foi inserida como palavra).

```
1  int buscar(TrieNo *raiz,const char *
       palavra){
2    TrieNo *atual = raiz;
3    for(int i=0; palavra[i]!='\0'; i++){
4      int idx = palavra[i] - 'a';
5      if (atual->filhos[idx] == NULL)
6          return 0;//não achou
7      atual = atual->filhos[idx];
8    }
9    return atual->fimDePalavra;
10     //é palavra completa?
11 }
```

Busca por Prefixo

Ideia

Mesmo percurso da busca exata, mas **sem** exigir que o nó final seja fim de palavra: basta o caminho existir para que haja **ao menos uma** palavra com esse prefixo no conjunto.

Base do autocompletar

Esta é exatamente a primeira etapa do autocompletar: localizar o nó do prefixo digitado antes de listar as palavras que dele derivam (próximo tópico).

```
1  int possuiPrefixo(TrieNo *raiz, const char
       *prefixo){
2    TrieNo *atual = raiz;
3    for(int i=0; prefixo[i]!='\0'; i++){
4      int idx = prefixo[i] - 'a';
5      if(atual->filhos[idx]==NULL) return 0;
6      atual = atual->filhos[idx];
7    }
8    return 1; //caminho existe
9  //ou seja, há palavras com esse prefixo
10 }
```

Aplicações

Autocompletar, corretores e mais

Autocompletar (Autocomplete)

Como funciona

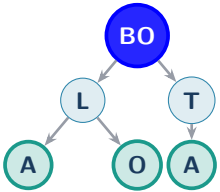
Dado um prefixo p já digitado pelo usuário:

1. Percorrer a Trie seguindo os caracteres de p até o nó correspondente.
2. A partir desse nó, fazer uma **busca em profundidade (DFS)** coletando todos os nós marcados como fim de palavra.
3. Cada caminho raiz→fim-de-palavra encontrado é uma **sugestão** de palavra completa.

Complexidade

Localizar o prefixo: $\mathcal{O}(|p|)$. Listar todas as k palavras com esse prefixo: $\mathcal{O}(|p| + k)$.

Sugestões para o prefixo "BO"



DFS a partir de `BO` encontra:
BOLA, BOLO, BOTA

Corretores Ortográficos e Outras Aplicações

▪ Corretor ortográfico

Verificação rápida: a palavra digitada existe na Trie (dicionário)? Se não, gerar candidatas variando 1–2 caracteres (inserção, remoção, troca) e verificar quais **existem** na *Trie* – muito mais rápido do que comparar com cada palavra do dicionário individualmente.

▪ Roteamento de IPs (IP routing)

Tabelas de roteamento usam **Árvores de Prefixos Binárias** (*trie de bits*) para encontrar o prefixo de rede mais específico que casa com um endereço IP (*longest prefix match*).

▪ Indexação de texto

Motores de busca e editores de texto usam tries (ou variantes como *suffix tries*) para indexar palavras de um documento e responder buscas por prefixo/substring rapidamente.

▪ Variante: Trie compacta (Radix Tree)

Quando há longos trechos sem ramificação, comprimi-los em uma única aresta rotulada por uma **substring** (em vez de um caractere) economiza memória – usado em bancos de dados e sistemas de arquivos.

Árvore de Prefixo vs. Tabela Hash

Quando usar cada estrutura?

Comparando as Duas Estruturas

Operação	Trie	Tabela Hash
Inserir palavra de tamanho L	$\mathcal{O}(L)$	$\mathcal{O}(L)$ amortizado
Buscar palavra exata	$\mathcal{O}(L)$	$\mathcal{O}(1)$ médio
Buscar por prefixo	$\mathcal{O}(L + k)$	$\mathcal{O}(n)$ (sem suporte nativo)
Ordenar chaves	natural (DFS em ordem)	precisa reordenar tudo
Pior caso	$\mathcal{O}(L)$ garantido	$\mathcal{O}(n)$ (muitas colisões)
Uso de memória	maior (nó por caractere)	geralmente menor

Use Árvore de Prefixo quando...

- Precisar de busca por **prefixo**
- Quiser **autocompletar** ou sugestões
- Precisar de chaves **ordenadas**

Use Tabela Hash quando...

- Só precisar de existência/igualdade exata
- Memória for o fator mais crítico
- Não houver consultas por prefixo

Exercícios

Pratique Tries - Árvores de Prefixos

Exercícios

1. **Construção manual:** Desenhe a Trie resultante da inserção, em ordem, das palavras `CASA`, `CARRO`, `CARTA`, `CAIXA`. Indique quais nós são marcados como fim de palavra.

2. **Implementação – contarPalavras:** Implemente `int contarPalavras(TrieNo *raiz)`, que retorna quantas palavras completas estão armazenadas na Trie (percorra recursivamente todos os nós).

3. **Implementação – listarComPrefixo:** Implemente uma função que recebe um prefixo e imprime todas as palavras da Trie que começam com ele, usando DFS a partir do nó do prefixo.

4. **Análise:** Compare o custo de memória de armazenar 1000 palavras de tamanho médio 6 em uma Trie (vetor de 26 filhos por nó) e em uma hash table com encadeamento. Em qual cenário a Trie deixa de ser vantajosa?

Fim da Aula 18

Dúvidas e próximos passos

Encerrando a Aula 18

▪ Resumo da aula

Trie: árvore de prefixos, um nó por caractere
Inserção/busca: $\mathcal{O}(L)$, independente do número de palavras
Busca por prefixo: vantagem central sobre tabelas hash
Aplicações: autocompletar, corretores, roteamento de IP
Radix tree: compressão de caminhos sem ramificação
Trade-off: Trie ganha em prefixo/ordenação, tabela hash ganha em memória e busca exata média

▪ Próxima aula — Compressão de Dados

- Codificação de Huffman
- Árvores de codificação e códigos de prefixo
- Construção via fila de prioridade
- Aplicações: compactação de arquivos e texto

carolsoko@ifba.edu.br

Dúvidas? Mande uma mensagem!