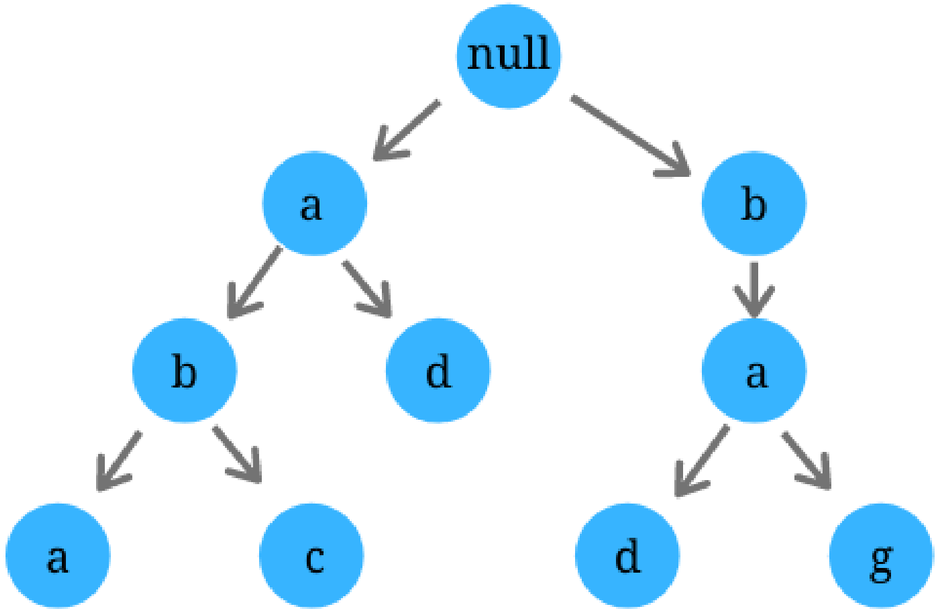
Tries
(Árvores Digitais)

Alunos: Ângelo Mutti, Arthur Soares, Christian Will, Eduardo Curcino

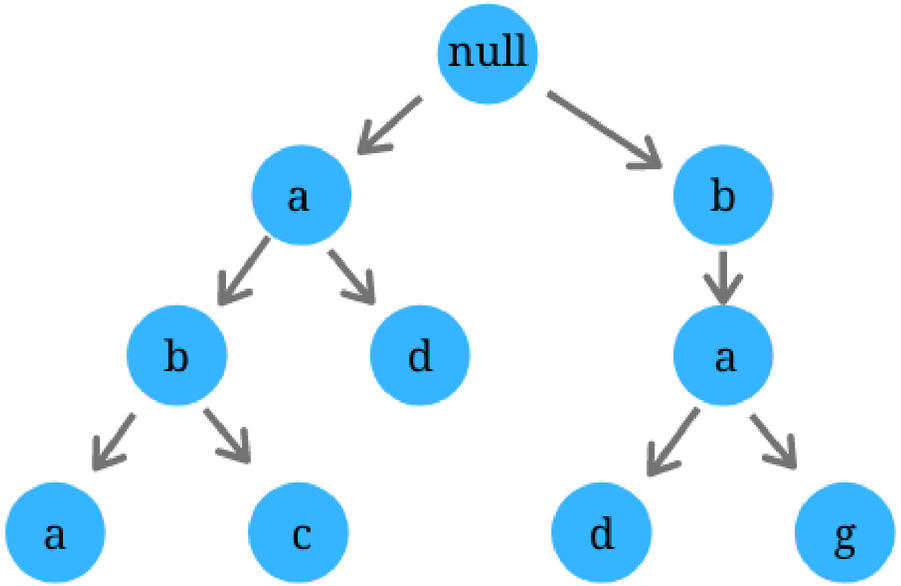
O que é uma trie?



a b
a b a
a b c
a d
b a
b a d
b a g

O que é uma trie?

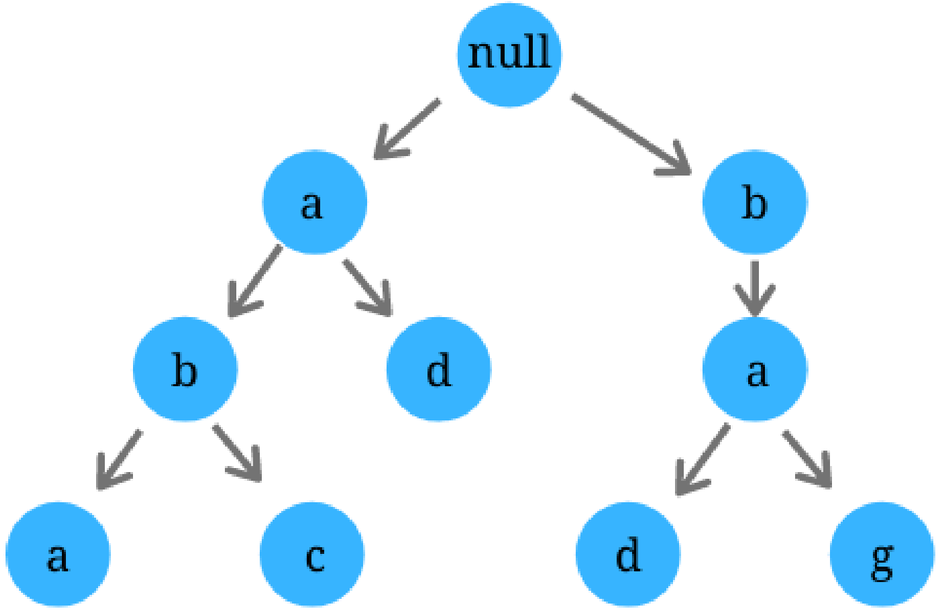
- É uma estrutura de dados em árvore que armazena sequências de caractéres.



a b
a b a
a b c
a d
b a
b a d
b a g

O que é uma trie?

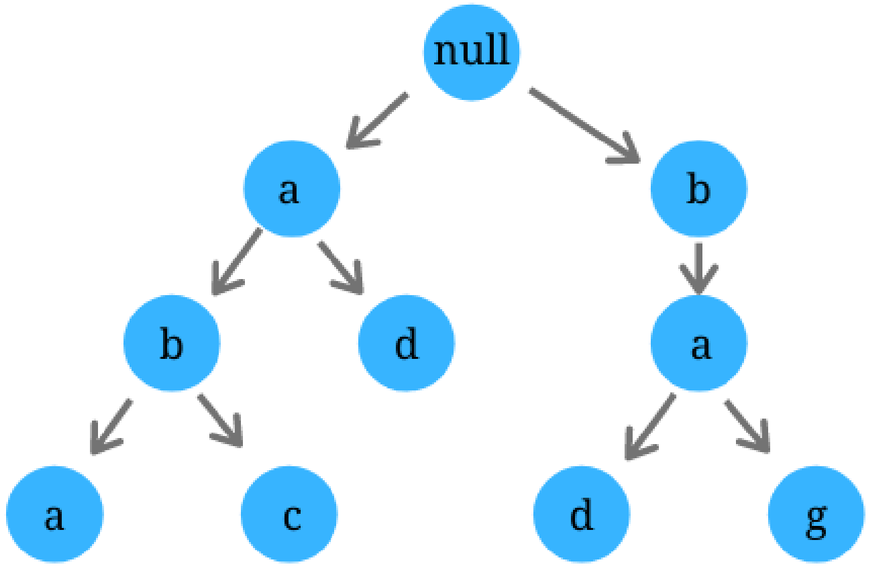
- O caminho da raiz até um nó representa um prefixo.



a b
a b a
a b c
a d
b a
b a d
b a g

O que é uma trie?

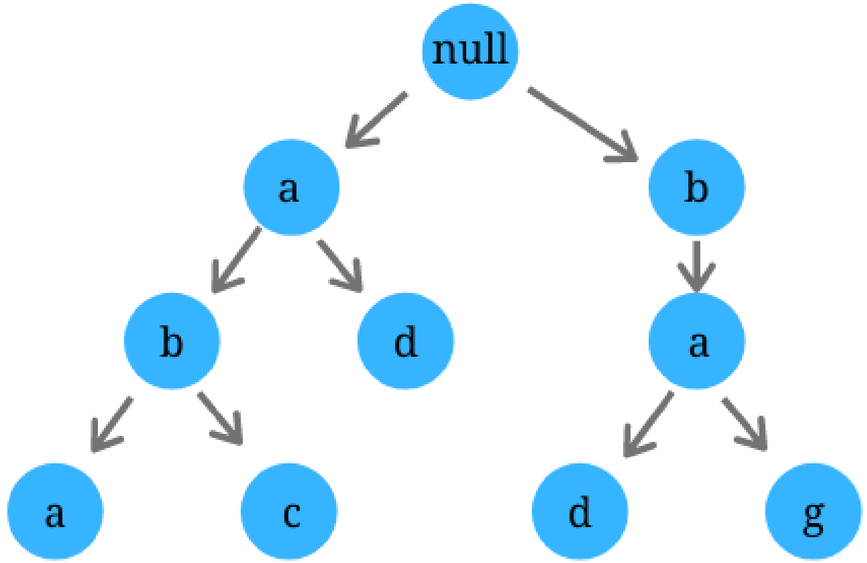
- Palavras com o mesmo prefixo compartilham o mesmo caminho inicial.



a b
a b a
a b c
a d
b a
b a d
b a g

O que é uma trie?

- Vantagem principal: Busca por prefixos de forma extremamente rápida e eficiente.



a b
a b a
a b c
a d
b a
b a d
b a g

Estrutura de um nó da trie

- Um nó de uma Trie é bem simples e geralmente tem dois componentes principais:

1. Conjunto de 'filhos' (children)
2. Marcador **isEndOfWord**

Estrutura de um nó da trie

- Conjunto de filhos (children)

  É um ponteiro ou uma coleção de ponteiros. Cada ponteiro que não é nulo aponta para outro nó, representando a continuação de um prefixo com um caractere adicional.

Estrutura de um nó da trie

- Marcador **isEndOfWord**

Sua função é responder a uma pergunta fundamental: "O caminho da raiz até este exato nó representa uma palavra completa que foi inserida na Trie?"

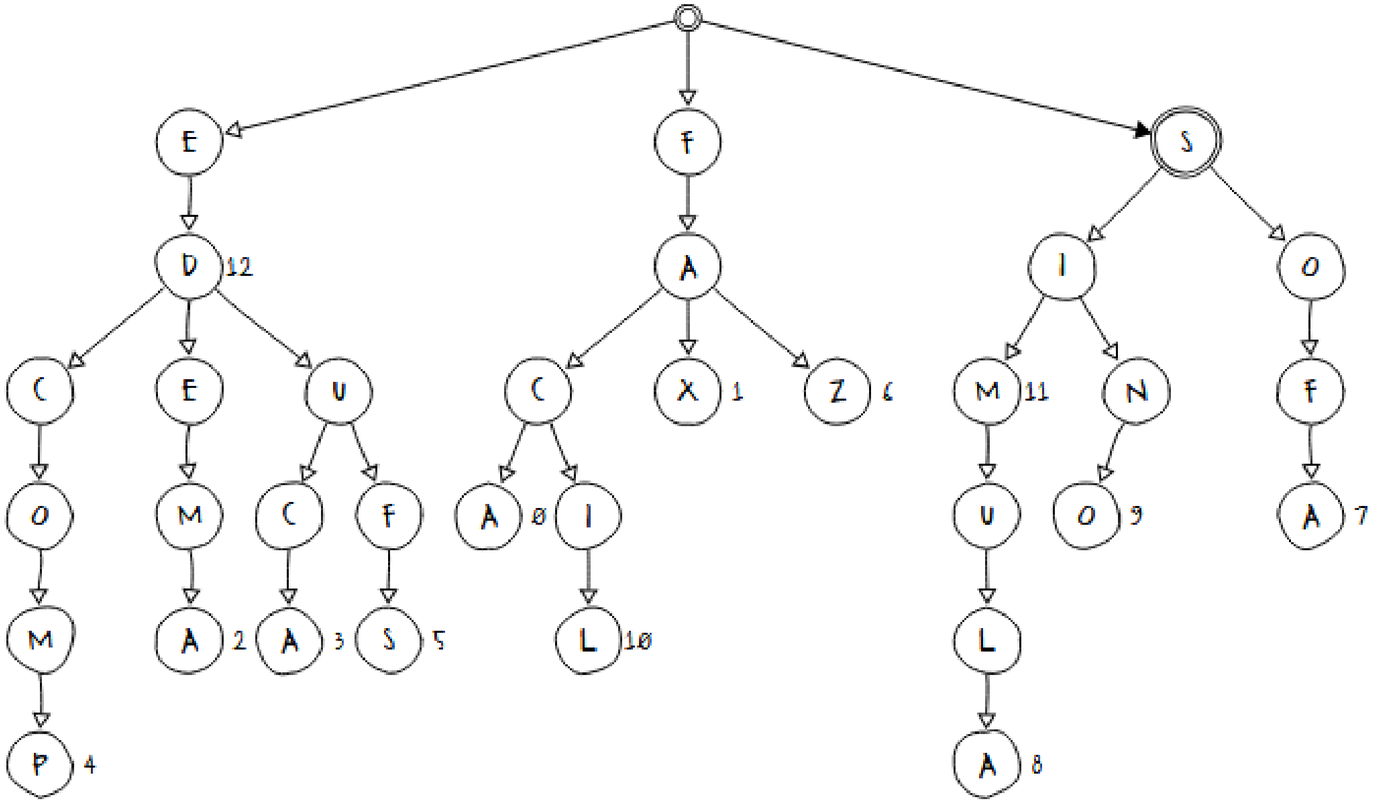
Implementação

- Inserção

```
void inserir(NoTrie* raiz, const char* palavra) {
    NoTrie* atual = raiz;
    while (*palavra) {
        int indice = *palavra - 'a';
        if (!atual->filhos[indice])
            atual->filhos[indice] = criarNo();
        atual = atual->filhos[indice];
        palavra++;
    }
    atual->ehFimDePalavra = true;
}
```

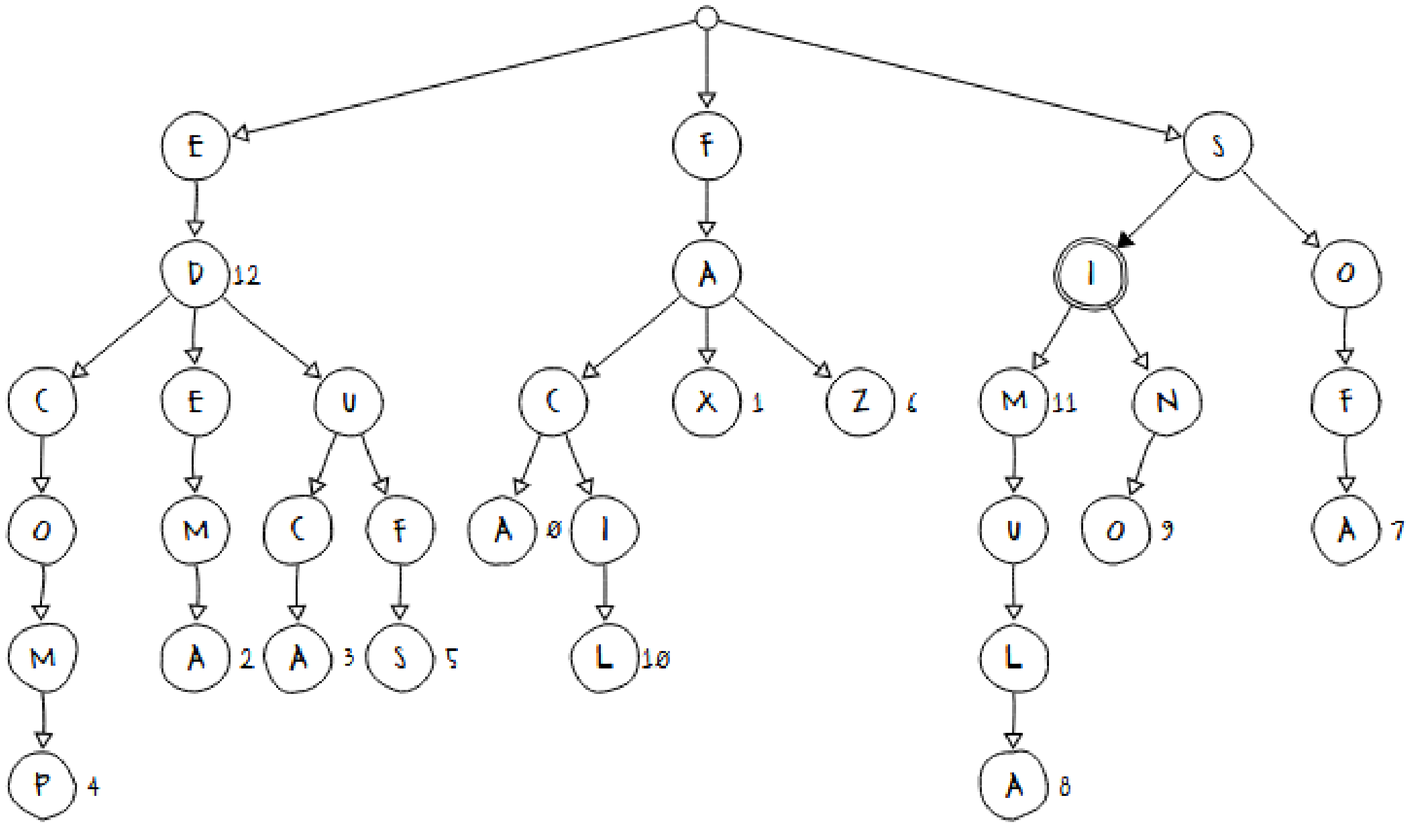
Implementação

- Inserção



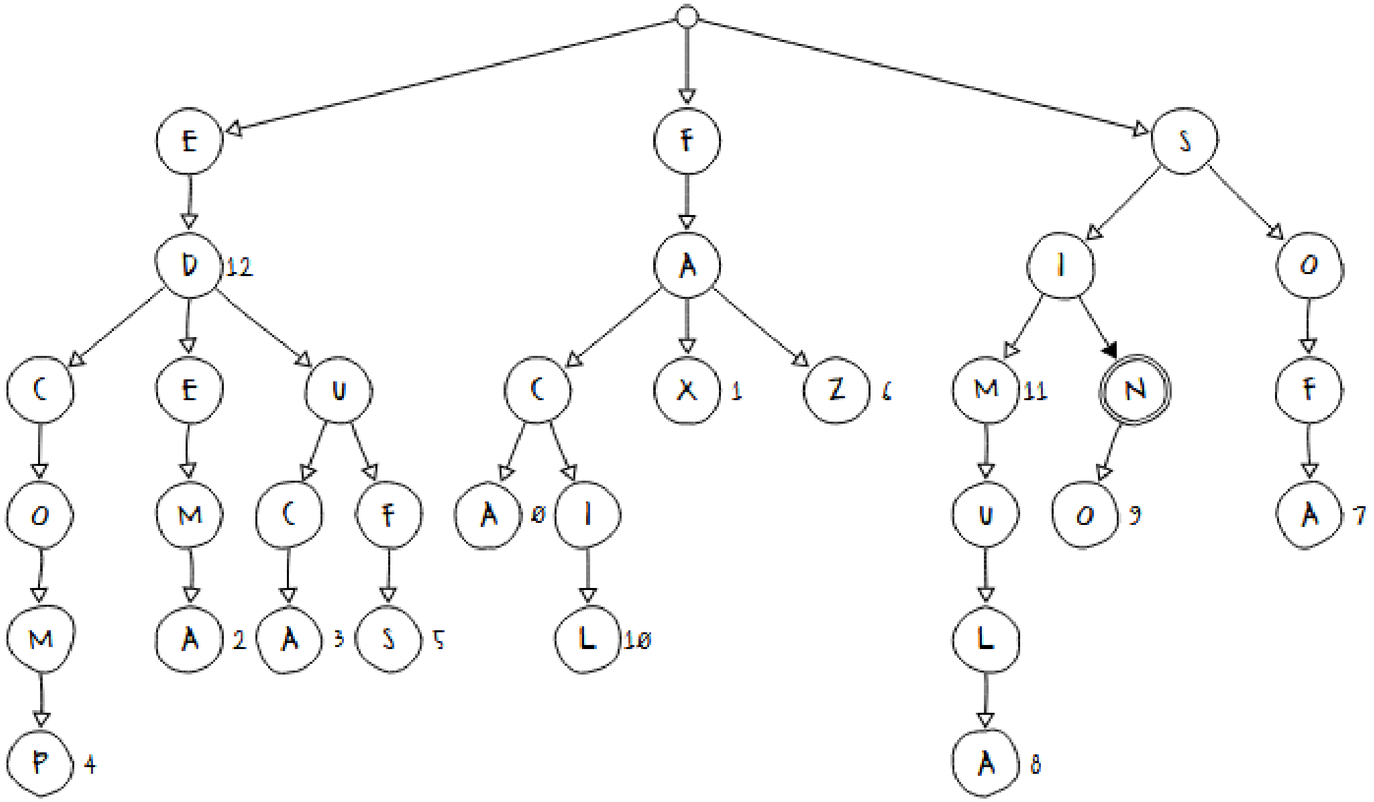
Implementação

- Inserção



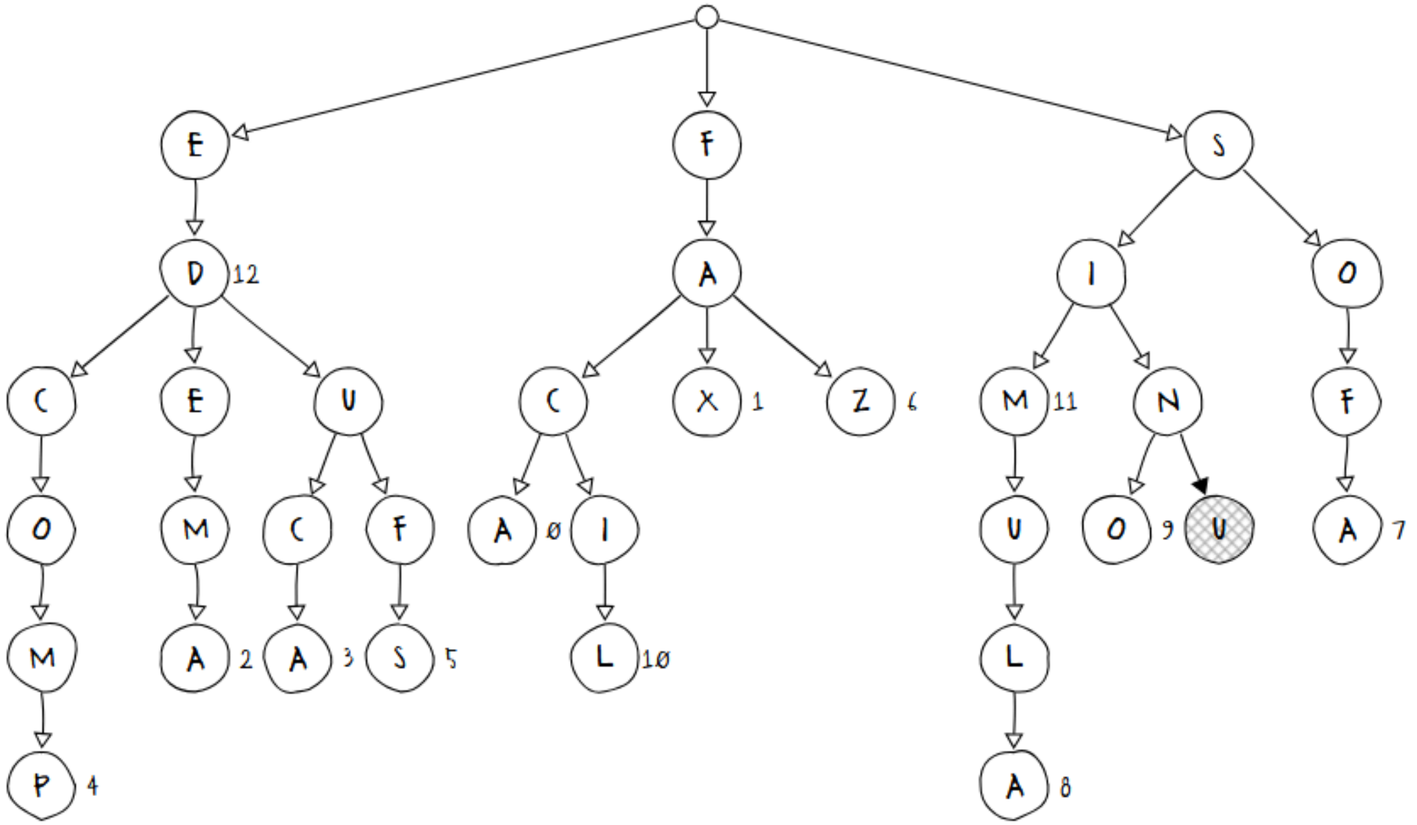
Implementação

- Inserção



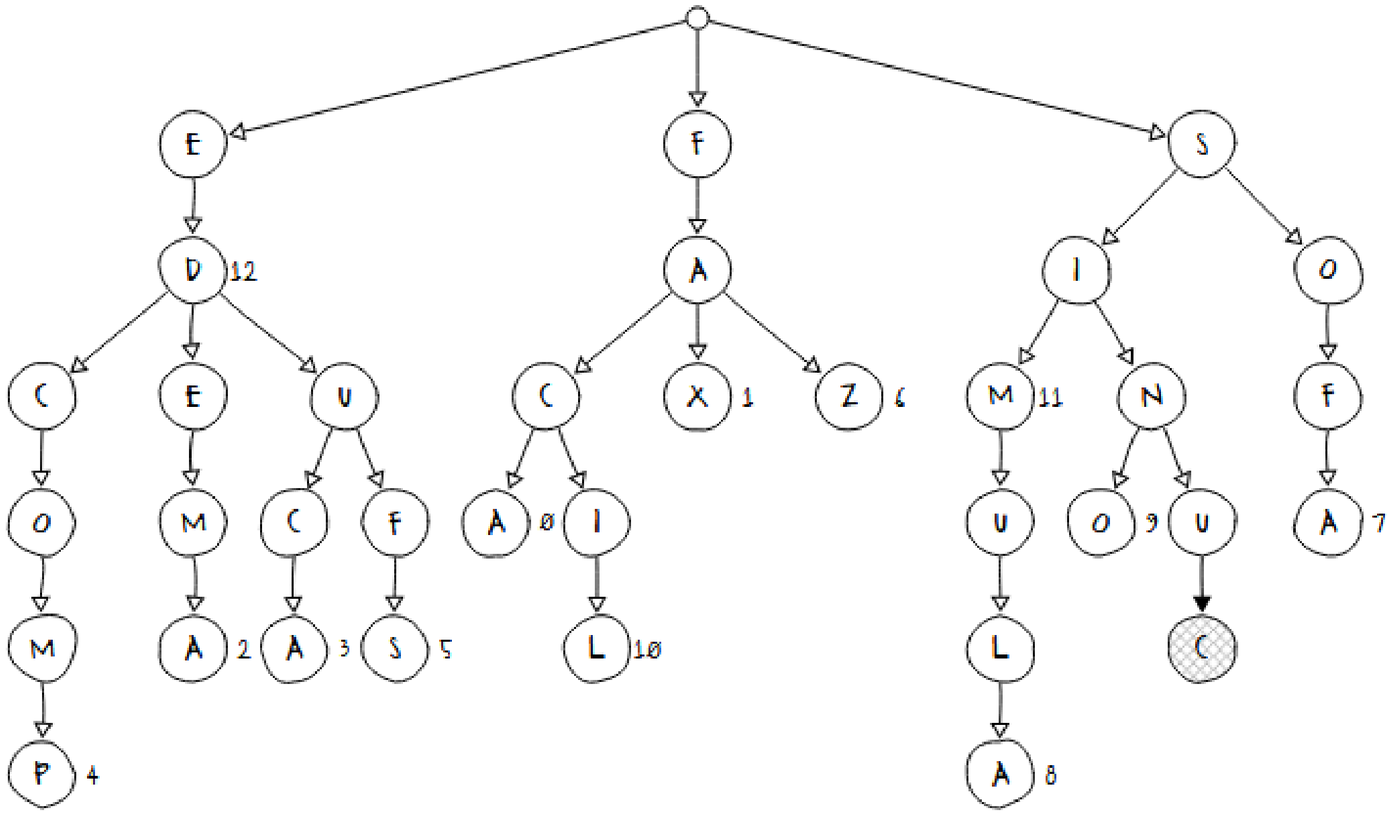
Implementação

- Inserção



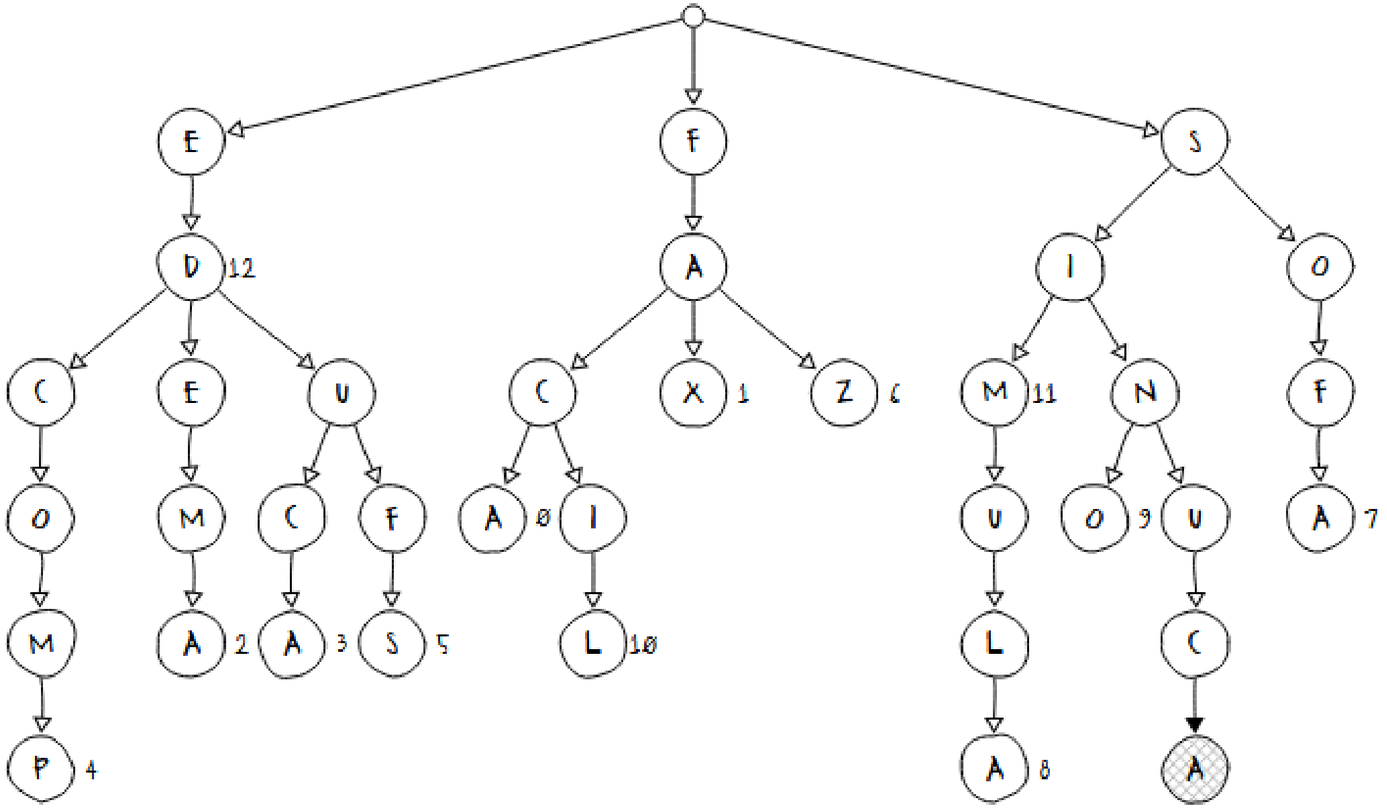
Implementação

- Inserção



Implementação

- Inserção



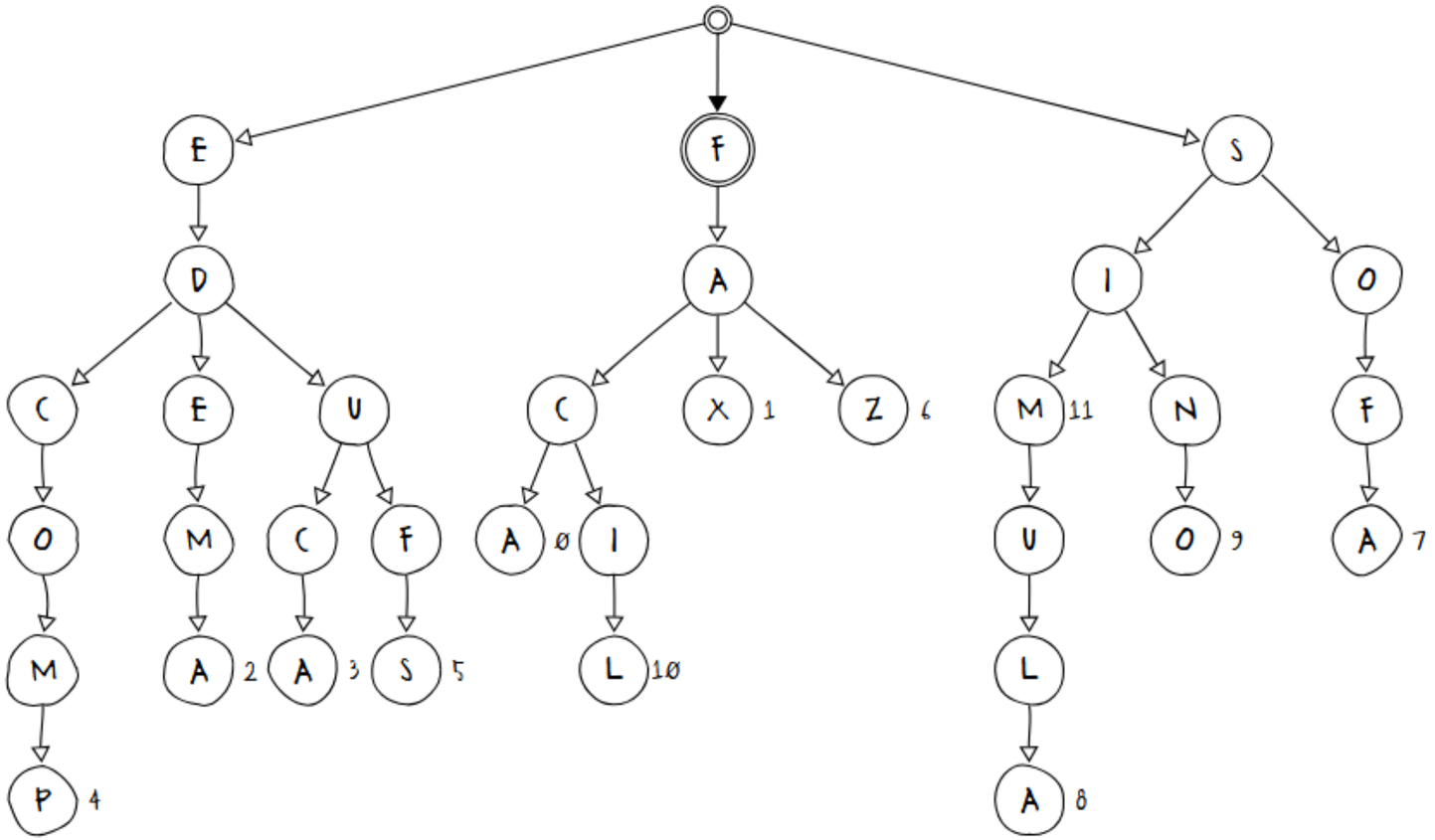
Implementação

- Busca

```
bool buscar(NoTrie* raiz, const char* palavra) {
    NoTrie* atual = raiz;
    while (*palavra) {
        int indice = *palavra - 'a';
        if (!atual->filhos[indice])
            return false;
        atual = atual->filhos[indice];
        palavra++;
    }
    return atual != NULL && atual->ehFimDePalavra;
}
```

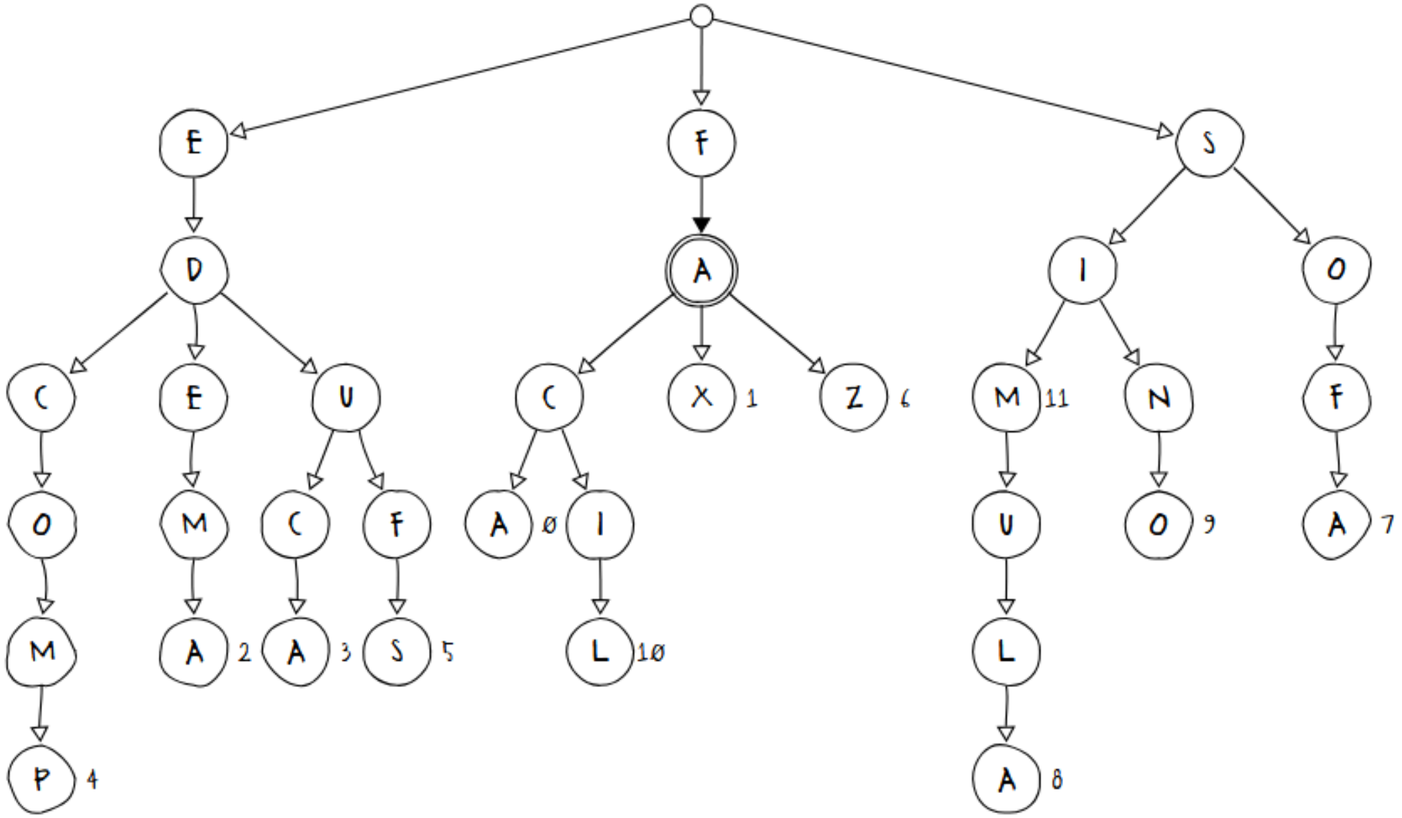
Implementação

- Busca



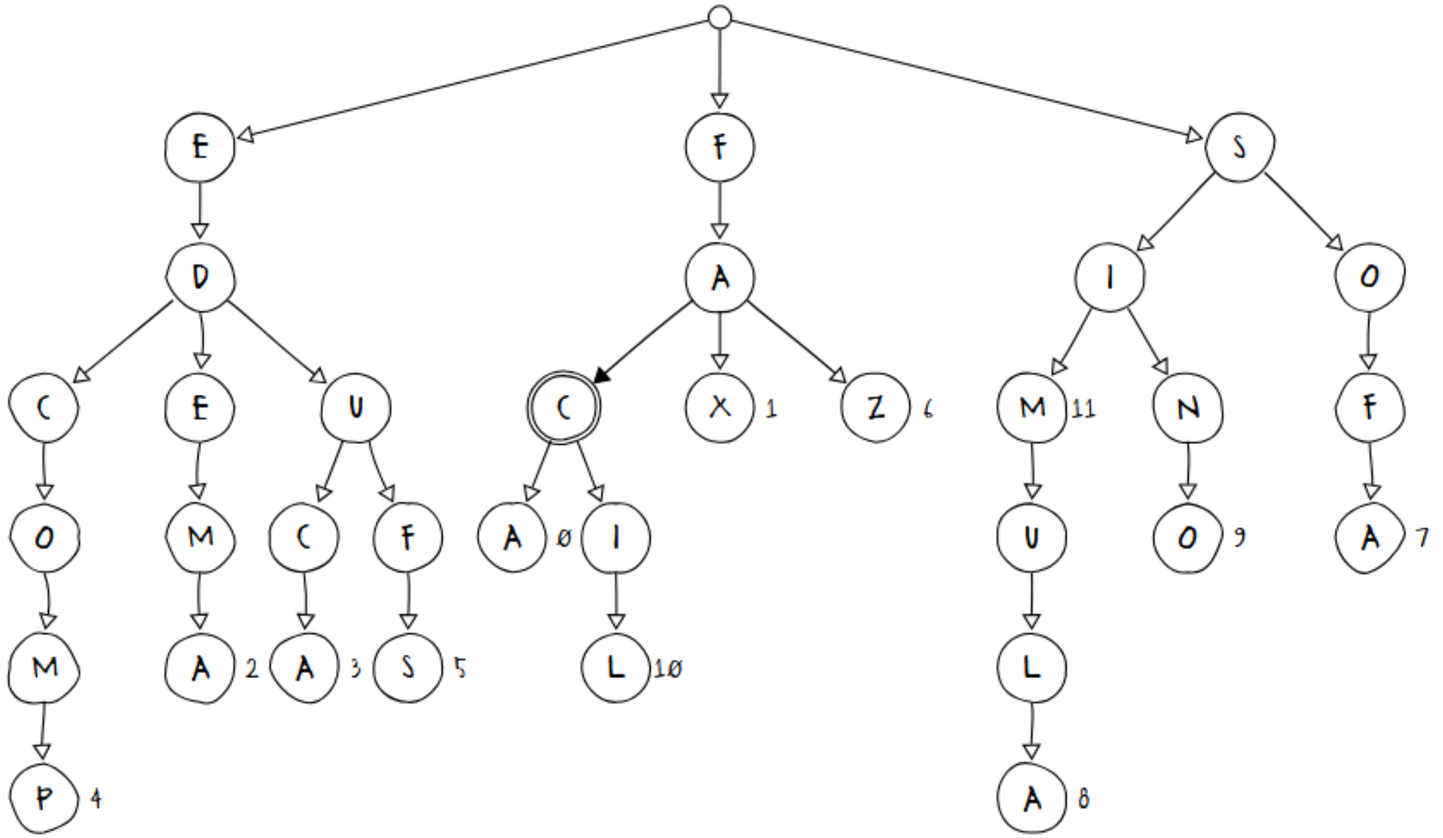
Implementação

- Busca



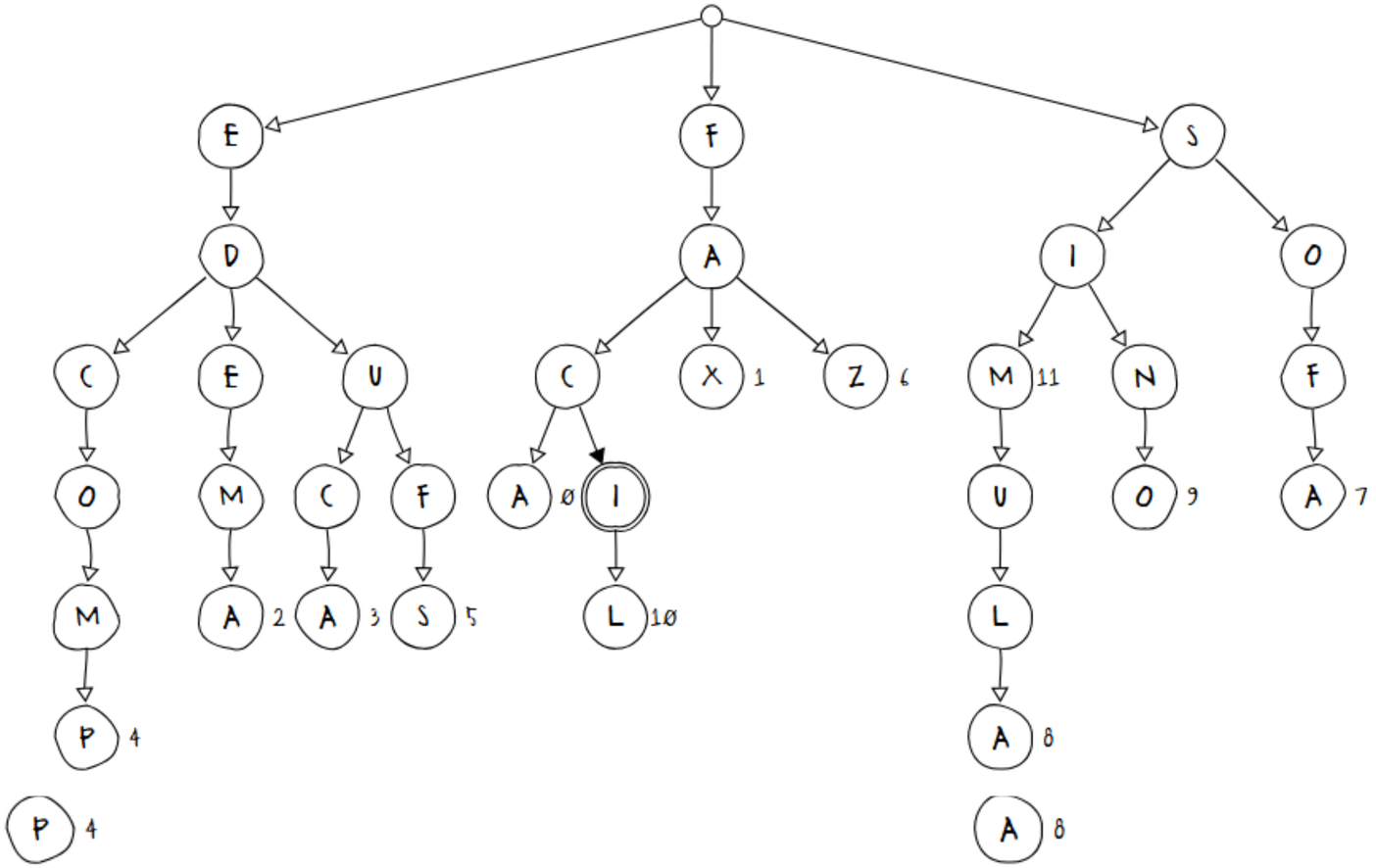
Implementação

- Busca



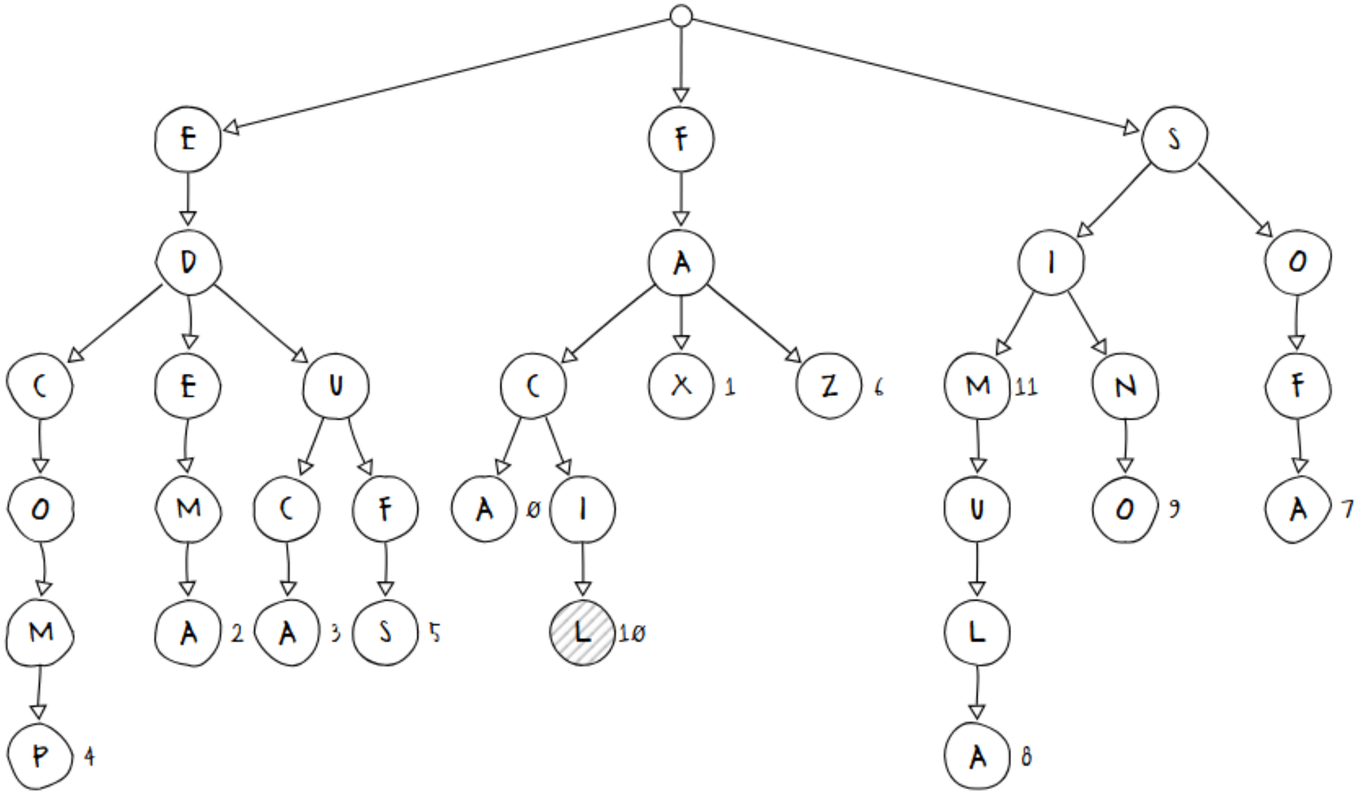
Implementação

- Busca



Implementação

- Busca



Implementação

- Remoção

```
bool auxiliarDeletar(NoTrie* atual, const char* palavra) {
    if (*palavra == '\0') {
        if (!atual->ehFimDePalavra)
            return false;
        atual->ehFimDePalavra = false;

        for(int i = 0; i < TAMANHO_ALFABETO; i++)
            if (atual->filhos[i] != NULL)
                return false;
        return true;
    }

    int indice = *palavra - 'a';
    if (!atual->filhos[indice])
        return false;

    bool deveDeletarNoAtual = auxiliarDeletar(atual->filhos[indice], palavra + 1);
```

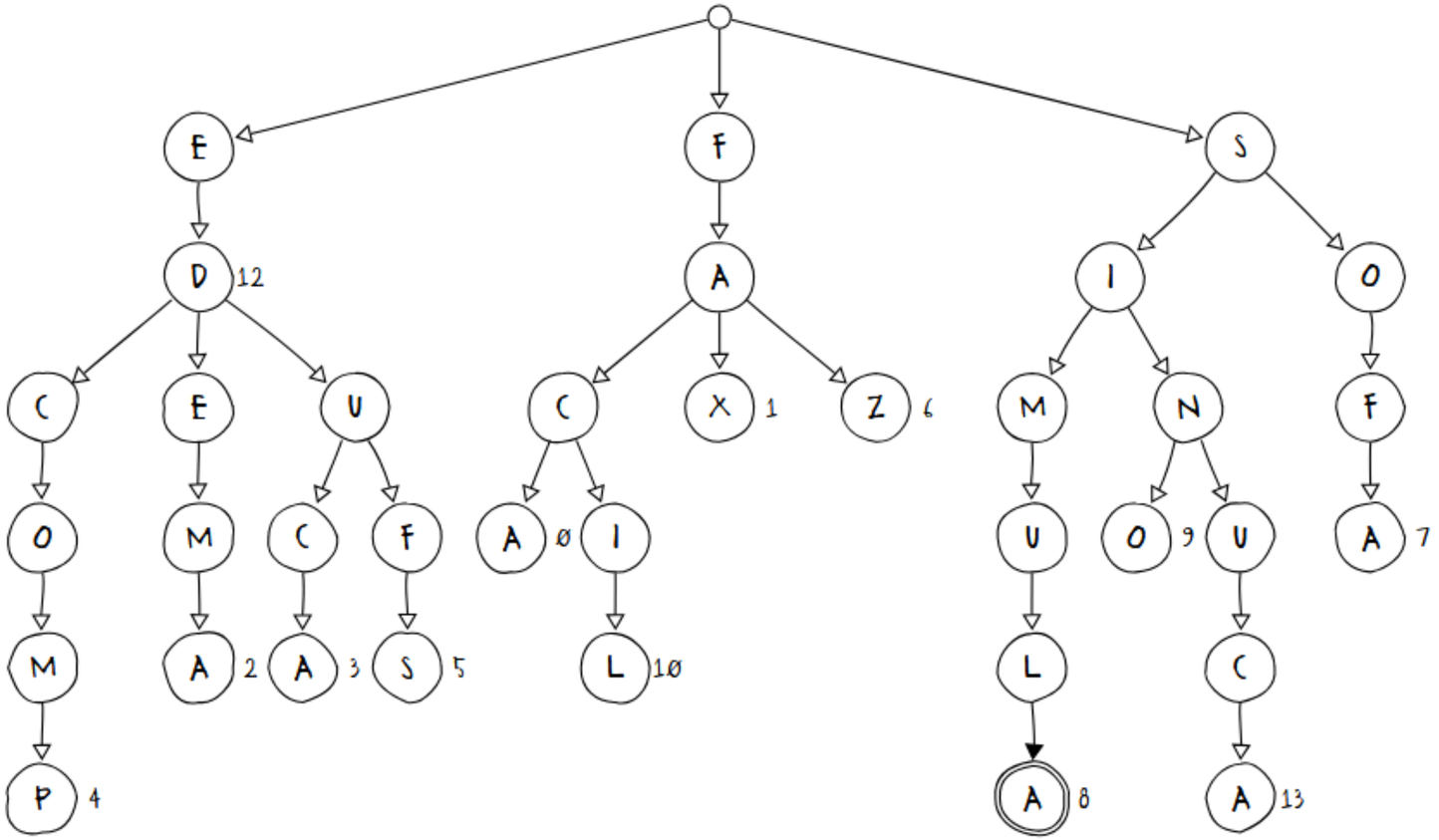
Implementação

- Remoção

```
        if (deveDeletarNoAtual) {
            free(atual->filhos[indice]);
            atual->filhos[indice] = NULL;

            if (!atual->ehFimDePalavra) {
                for(int i = 0; i < TAMANHO_ALFABETO; i++)
                    if (atual->filhos[i] != NULL)
                        return false;
                return true;
            }
        }

        return false;
    }

    void deletarPalavra(NoTrie* raiz, const char* palavra) {
        auxiliarDeletar(raiz, palavra);
    }
```

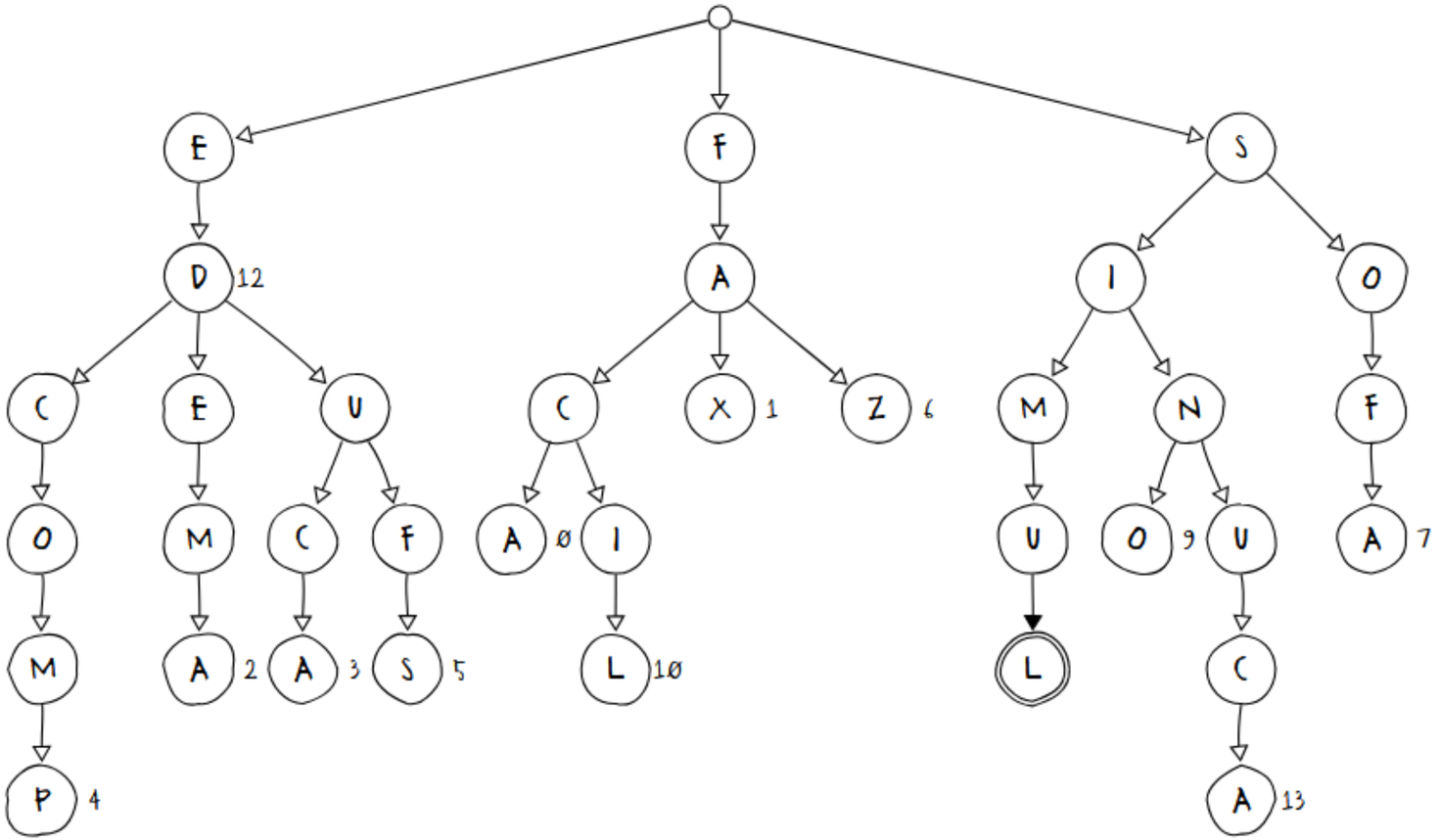
Implementação

- Remoção



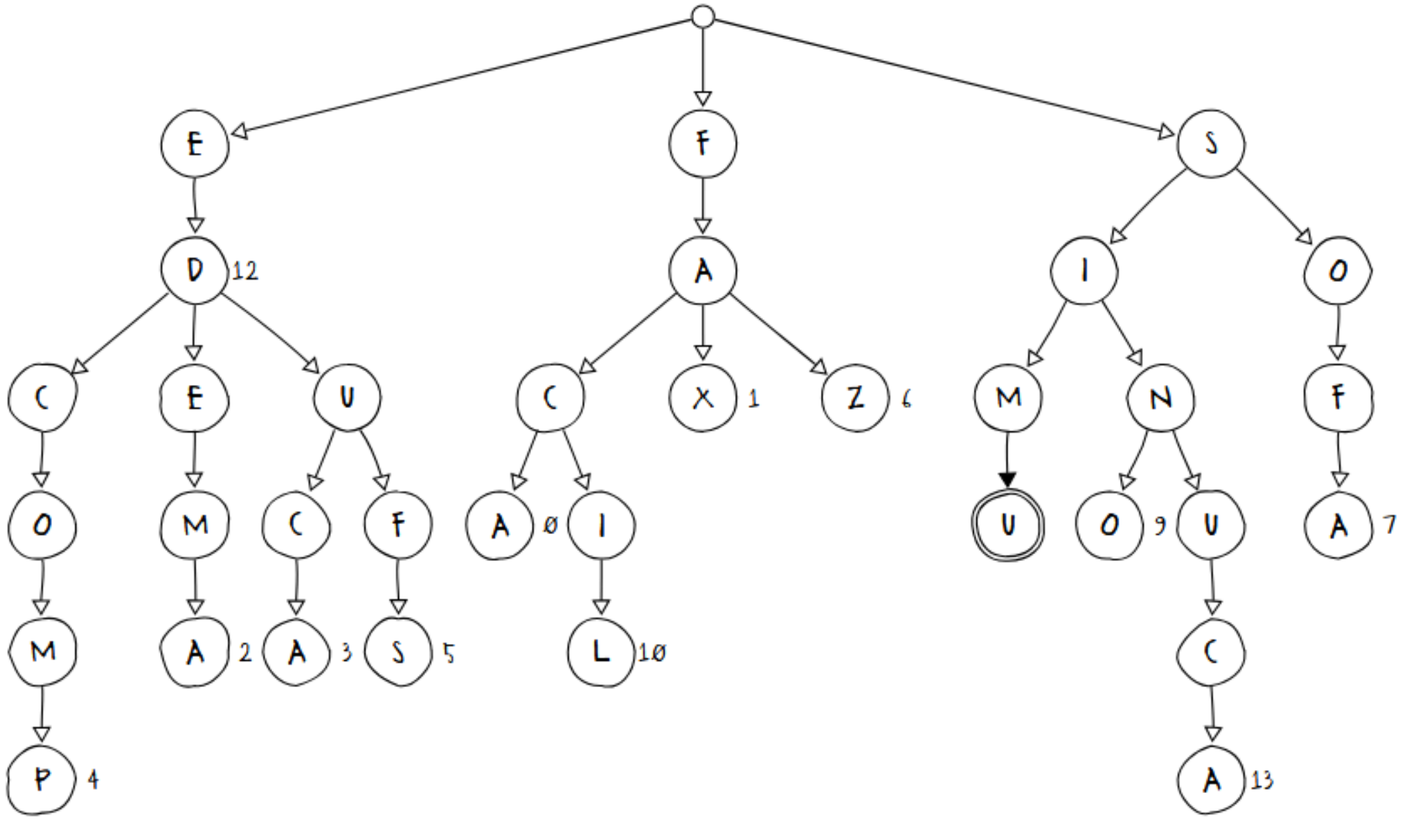
Implementação

- Remoção



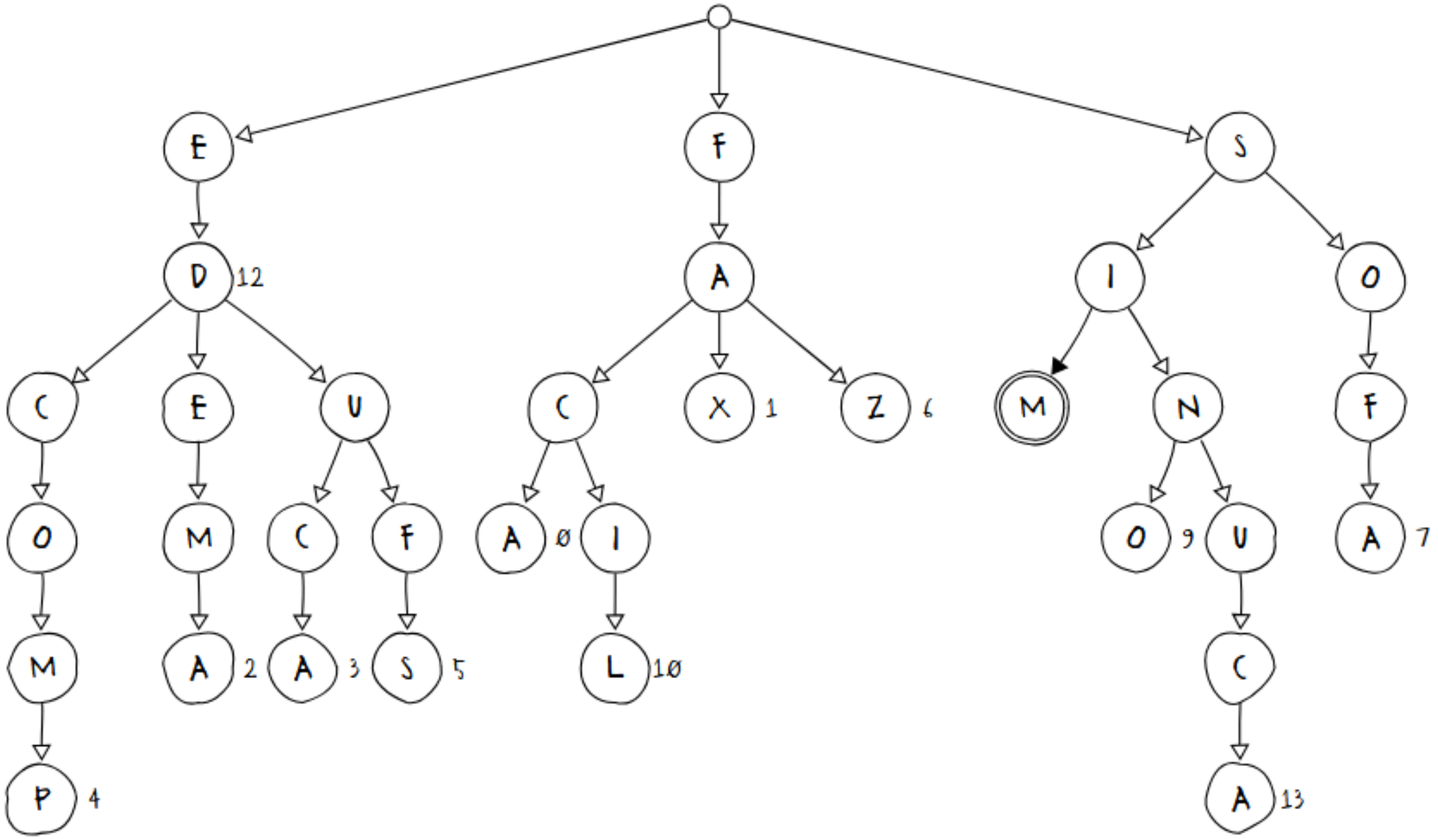
Implementação

- Remoção



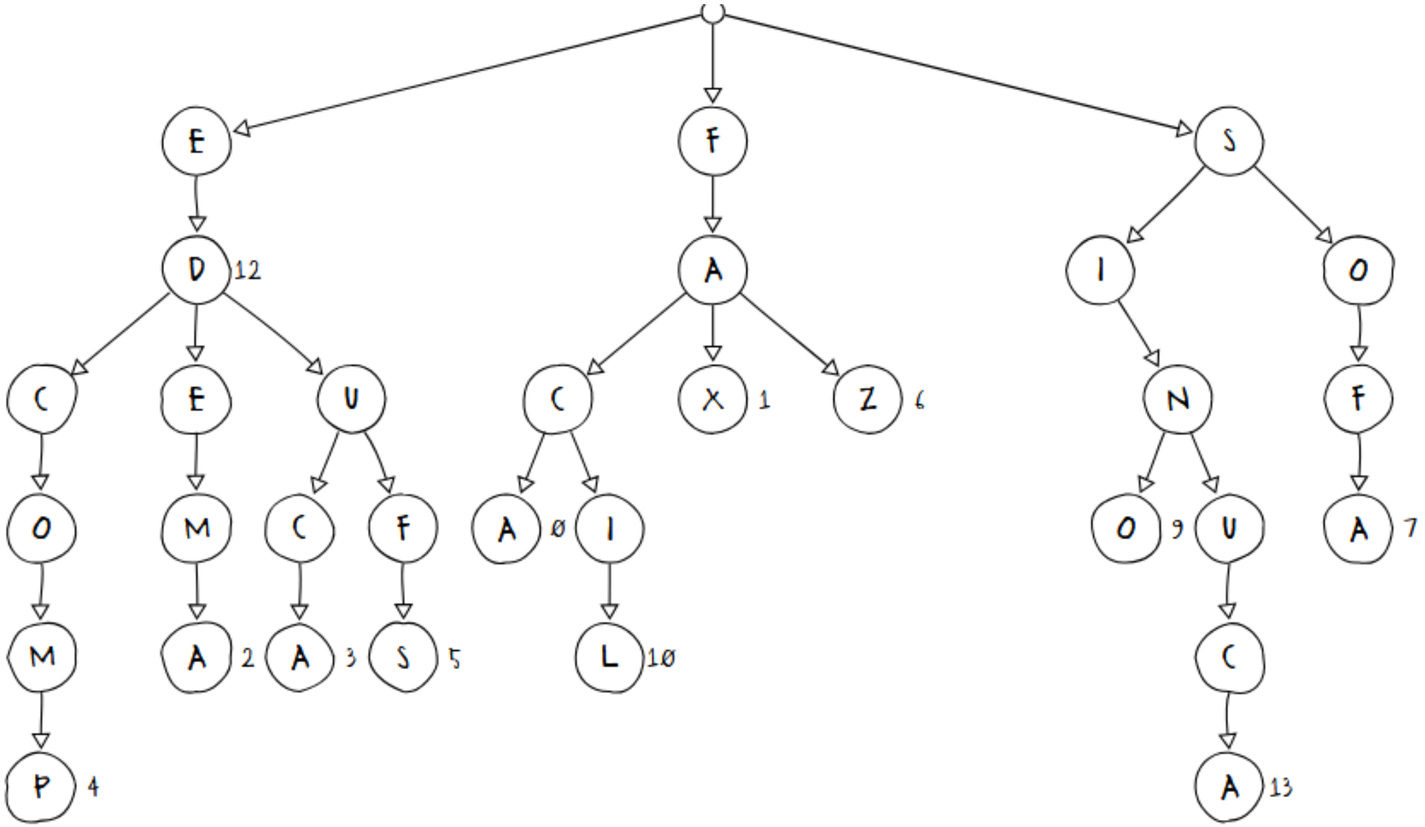
Implementação

- Remoção



Implementação

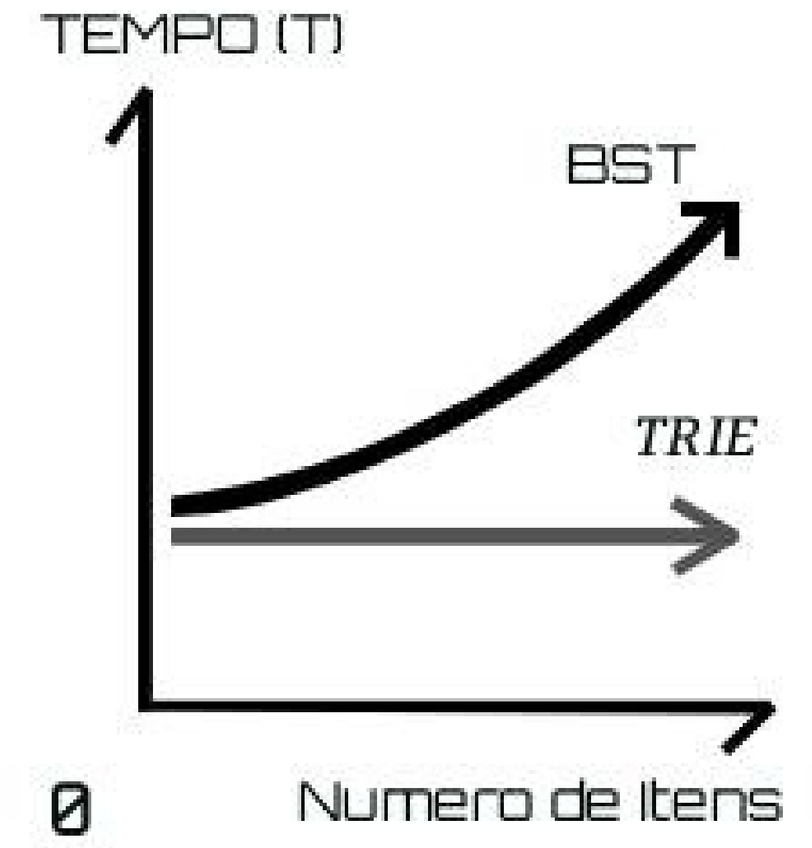
- Remoção

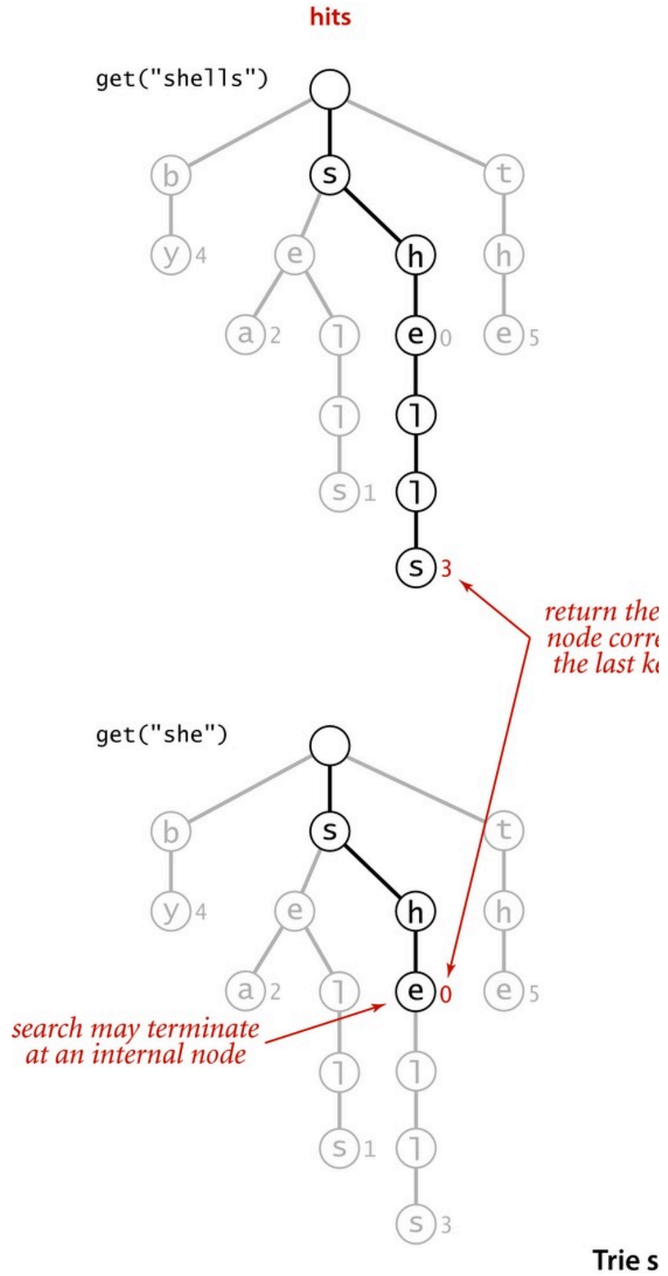


Complexidade O(L)

- **A principal característica** de uma Trie é que a complexidade de tempo para suas operações de busca, inserção e remoção depende do comprimento da chave (L), e não do número total de chaves (N) na estrutura.
- Exemplo: Para buscar uma string de comprimento L, você percorre a árvore, nó por nó, por L vezes. A cada passo, você faz uma única operação de acesso ao array

Complexidade O(L)





Vantagens e Desvantagens

Vantagens:

1. Velocidade de busca
2. Eficiência para prefixos

Desvantagens:

1. Consumo de memória
2. Implementação Complexa

Comparação com outras estruturas

BST

1. Eficiência de memória e flexibilidade para diversos tipos de dados
2. Desempenho de Busca: O(log N), onde N é o número de itens na árvore.

Trie

1. Otimizada para operações com strings e prefixos
2. Desempenho de Busca: O(L), onde L é o comprimento da palavra.

Comparação com outras estruturas

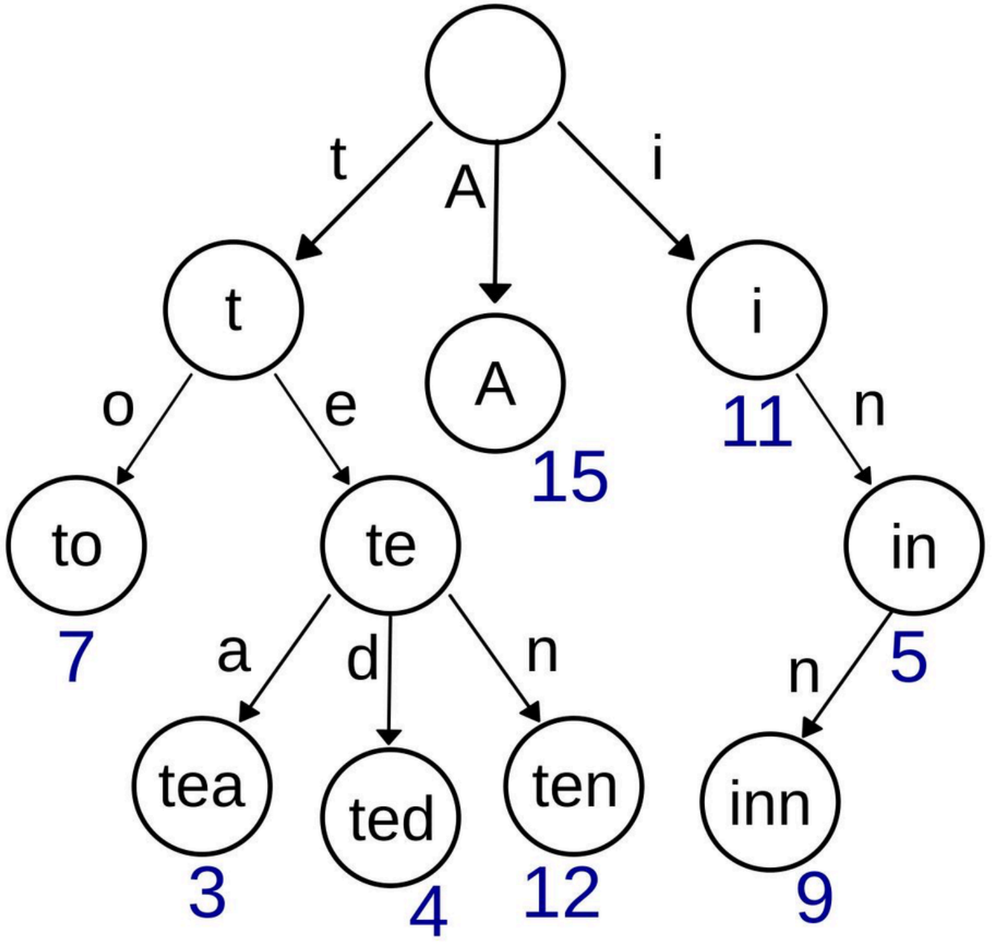
Hash

1. A mais rápida para buscas exatas, inserção e remoção, com tempo médio de O(1).
2. Não suporta busca por prefixo ou qualquer tipo de ordenação.

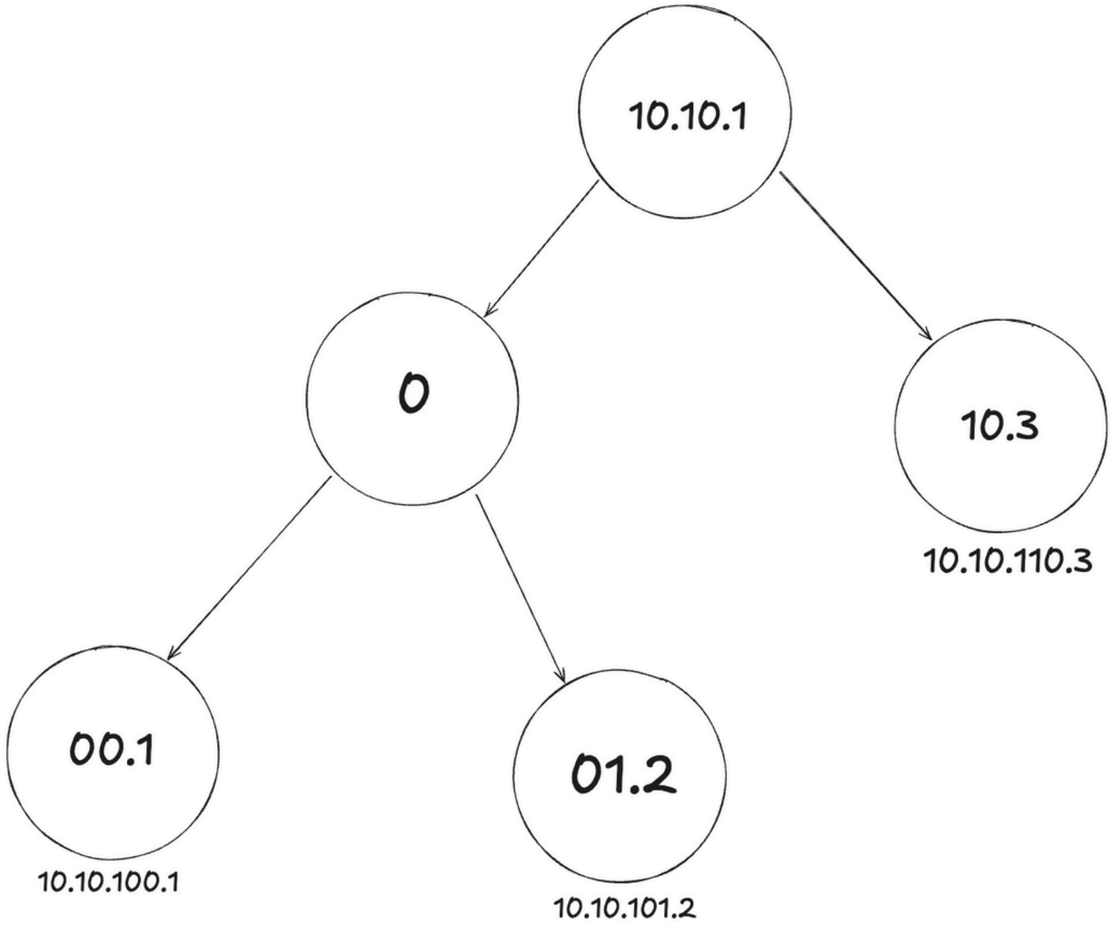
Trie

1. Busca por prefixo e mantém a ordem lexicográfica (alfabética) dos dados.
2. Não sofre com colisões, garantindo um desempenho mais estável no pior caso.

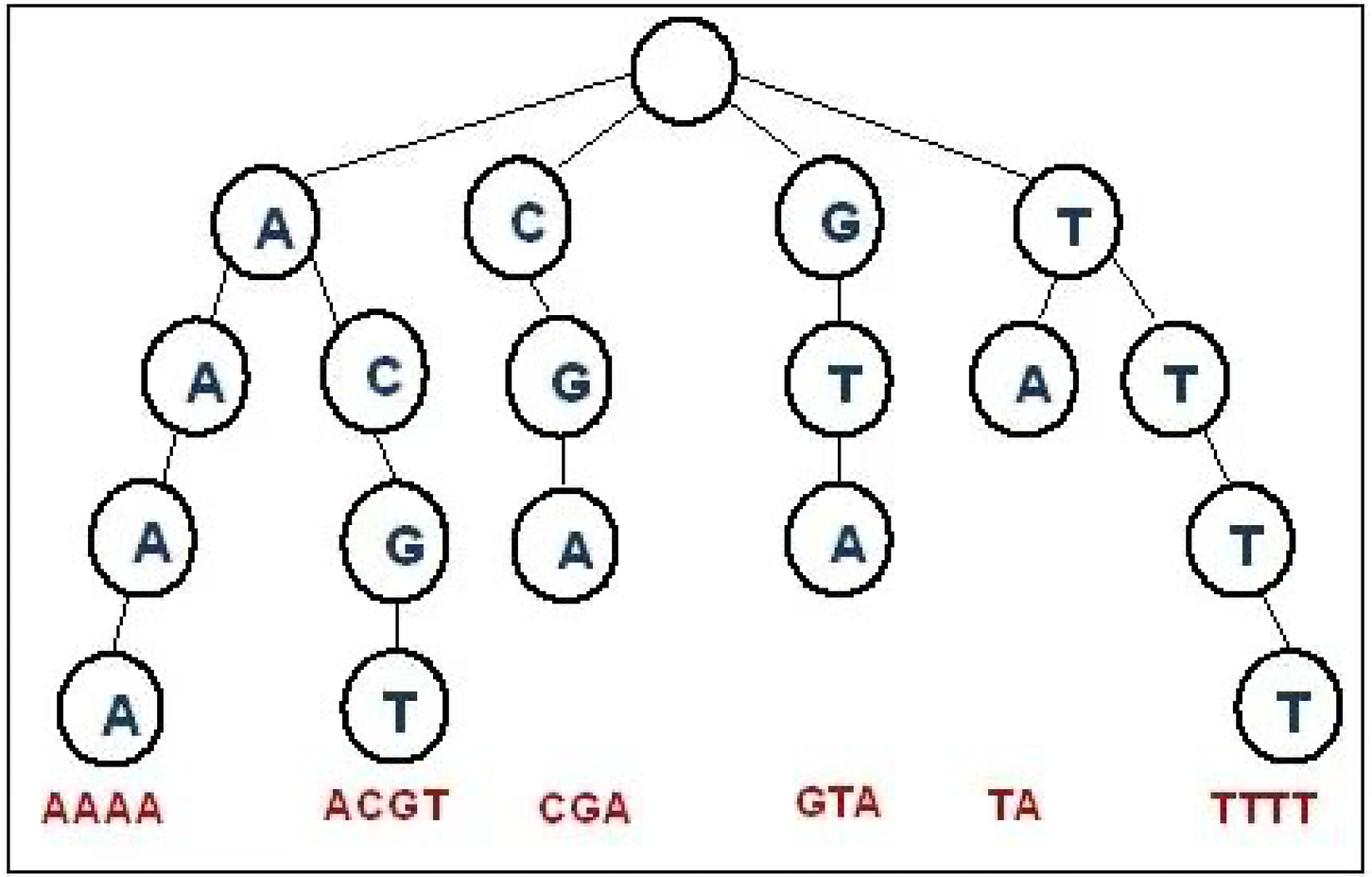
Outra implementação



Outros Alfabetos



Outros Alfabetos



Referências

- Slides do prof. Bruno Prado. UFS, 2025
- J. L. Szwarcfiter. Estruturas de Dados e Seus Algoritmos.
- https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/tries.html
- https://www.researchgate.net/figure/Suffix-trie-built-for-two-DNA-sequences-S1-ACGT-and-S2-ACT_fig1_220195697