

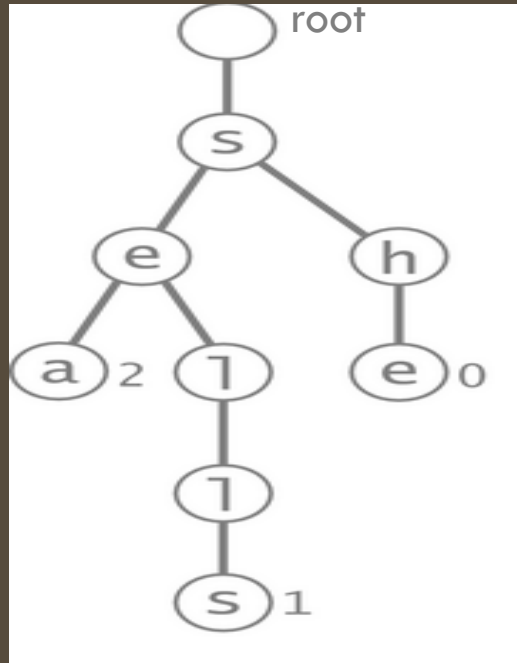


TRIE

Bruna Damaris
Heitor Almeida
Mayra Lessa
Paulo Silva
Thamires Gonçalves

TRIE

- Uma TRIE é um tipo de árvore usado para implementar TSs (tabelas de símbolos) de strings.
- Edward Fredkin, o inventor, usa o termo **trie** (de "retrieval", recuperação), porque essa estrutura é basicamente usada na recuperação de dados.
- Tries são também conhecidas como árvores de prefixos.
- Um exemplo de trie:

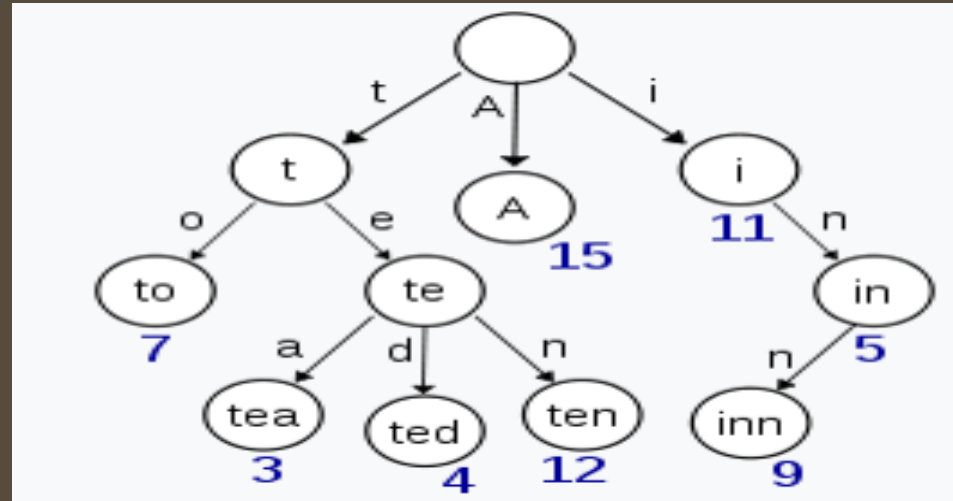


TRIE

- Ao contrário de uma árvore de busca binária, nenhum nó nessa árvore armazena a chave associada a ele;
- Ao invés disso, ela é determinada pela sua posição na árvore.
- Todos os descendentes de qualquer nó têm um prefixo comum com a cadeia associada com aquele nó, e a raiz é associada com a cadeia vazia.
- Normalmente, valores não são associados com todos os nós, apenas com as folhas e alguns nós internos que correspondem a chaves de interesse
- Embora seja o mais comum, **tries** não precisam utilizar cadeias de caracteres como chaves. Os mesmos algoritmos podem facilmente ser adaptados para funções similares de listas ordenadas de qualquer construção, como permutações em uma lista de dígitos, etc.

TRIE

- No exemplo abaixo, as chaves estão listadas nos nós e os valores abaixo dos mesmos.
- Cada palavra completa tem um valor inteiro associado a ela.
- Uma **trie** pode ser vista como um autômato finito determinístico, embora o símbolo em cada aresta é frequentemente implícito na ordem dos galhos.
- Não é necessário que as chaves sejam explicitamente armazenadas nos nós. Na imagem, as palavras são mostradas apenas para ilustrar como a **trie** funciona.



VANTAGENS E DESVANTAGENS COM RELAÇÃO A ÁRVORE DE BUSCA

- Usando Trie, as complexidades de busca podem ser levadas ao limite ideal (comprimento da chave).
- Se armazenarmos chaves na árvore de busca binária, uma BST bem balanceada precisará de tempo proporcional a $M * \log N$, onde M é o comprimento máximo da string e N é o número de chaves na árvore.
- Usando Trie, podemos procurar a chave no tempo $O(M)$. No entanto, a penalidade é sobre os requisitos de armazenamento Trie.
- A principal desvantagem da Trie é que ela precisa de muita memória para armazenar as strings.
- Para cada nó, temos muitos ponteiros de nós (igual ao número de caracteres do alfabeto). Se o espaço for o assunto, então a Árvore de busca Ternária pode ser preferida para implementações de dicionários.

CRIANDO A TRIE

- Cada nó do Trie consiste em vários ramos. Cada ramo representa um possível caractere de chaves. Precisamos marcar o último nó de cada chave como “fim do nó da palavra”.
- A função ‘isEndOfWord’ da Trie é usada para distinguir o nó como o final do nó da palavra. Uma estrutura simples para representar nós do alfabeto pode ser a seguinte,

```
struct TrieNode {  
    struct TrieNode *children[ALPHABET_SIZE];  
    bool isEndOfWord;  
    // isEndOfWord is true if the node represents end of a word  
};
```

CRIANDO A TRIE

```
#define ARRAY_SIZE(a) sizeof(a)/sizeof(a[0])

#define ALPHABET_SIZE (26)

#define CHAR_TO_INDEX(c) ((int)c - (int)'a') // Converts key current character into index.

struct TrieNode {
    struct TrieNode *children[ALPHABET_SIZE];
    bool isEndOfWord; // isEndOfWord is true if the node represents end of a word
};

struct TrieNode *getNode(void) {
    struct TrieNode *pNode = NULL;

    pNode = (struct TrieNode *)malloc(sizeof(struct TrieNode));

    if (pNode) {
        int i;
        pNode->isEndOfWord = false;
        for (i = 0; i < ALPHABET_SIZE; i++) {
            pNode->children[i] = NULL;
        }
    }
    return pNode;
}
```

INSERÇÃO NA TRIE

- Inserir uma chave no Trie é uma abordagem simples. Cada caractere da chave de entrada é inserido como um nó Trie individual.
- Observe que os filhos são um array de ponteiros (ou referências) para os nós do próximo nível. O caractere chave atua como um índice nos filhos do array.
- Se a chave de entrada é nova ou uma extensão da chave existente, precisamos construir nós não existentes da chave e marcar o final da palavra para o último nó.
- Se a chave de entrada for o prefixo da chave existente no Trie, simplesmente marcaremos o último nó da chave como fim da palavra.
- O comprimento da chave determina a profundidade Trie.

INSERÇÃO NA TRIE

```
// If not present, inserts key into trie
// If the key is prefix of trie node, just marks
leaf node
void insert(struct TrieNode *root, const char *key)
{
    int level;
    int length = strlen(key);
    int index;
    struct TrieNode *pCrawl = root;
```

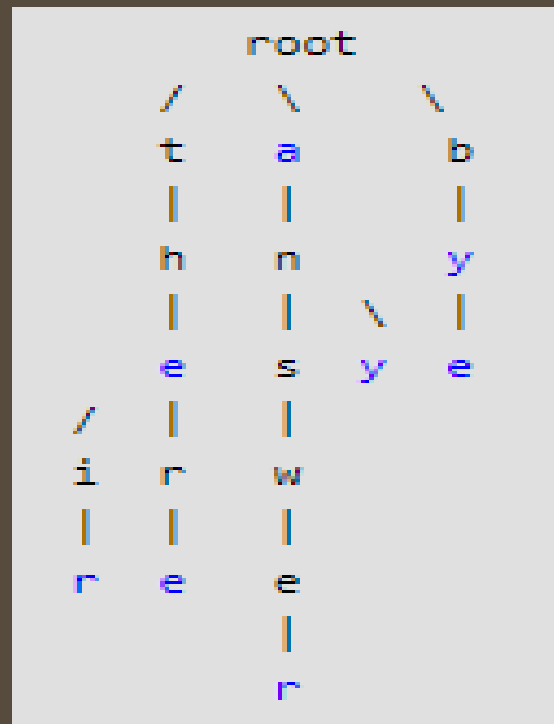
```
    for (level = 0; level < length; level++) {
        index = CHAR_TO_INDEX(key[level]);
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();
        pCrawl = pCrawl->children[index];
    }
    // mark last node as leaf
    pCrawl->isEndOfWord = true;
}
```

BUSCA NA TRIE

- Procurar por uma chave é semelhante à operação de inserção, no entanto, só comparamos os caracteres e movemos para baixo.
- A busca pode parar devido ao final da sequência de caracteres ou falta de chave na Trie.
- No primeiro caso, se o campo `isEndofWord` do último nó for `True`, a chave existirá no trie.
- No segundo caso, a busca termina sem examinar todos os caracteres da chave, já que a chave não está presente na trie.

BUSCA NA TRIE

- A figura a seguir explica a construção de trie usando as chaves fornecidas no exemplo abaixo,



- Na foto, cada caractere é do tipo `trie_node_t`. Por exemplo, a raiz é do tipo `trie_node_t`, e os filhos `a`, `b` e `t` são preenchidos, todos os outros nós da raiz serão `NULL`. Da mesma forma, "`a`" no próximo nível tem apenas um filho ("`n`"), todos os outros filhos são `NULL`. Os nós folha estão em azul.

BUSCA NA TRIE

// Returns true if key presents in trie, else false

```
bool search(struct TrieNode *root, const char *key)
```

```
{
```

```
    int level;
```

```
    int length = strlen(key);
```

```
    int index;
```

```
    struct TrieNode *pCrawl = root;
```

```
    for (level = 0; level < length; level++)
```

```
    {
```

```
        index = CHAR_TO_INDEX(key[level]);
```

```
        if (!pCrawl->children[index])
```

```
            return false;
```

```
        pCrawl = pCrawl->children[index];
```

```
    }
```

```
    return (pCrawl != NULL && pCrawl->isEndOfWord);
```

```
}
```

REFERÊNCIAS

- R. Sedgewick and K. Wayne, *Algorithms*, 4th Edition, Addison-Wesley, 2011.
- <https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/tries.html>
- <https://www.geeksforgeeks.org/trie-insert-and-search/>
- <https://pt.wikipedia.org/wiki/Trie>