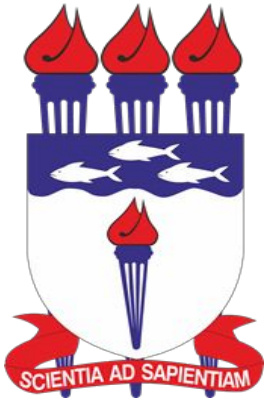


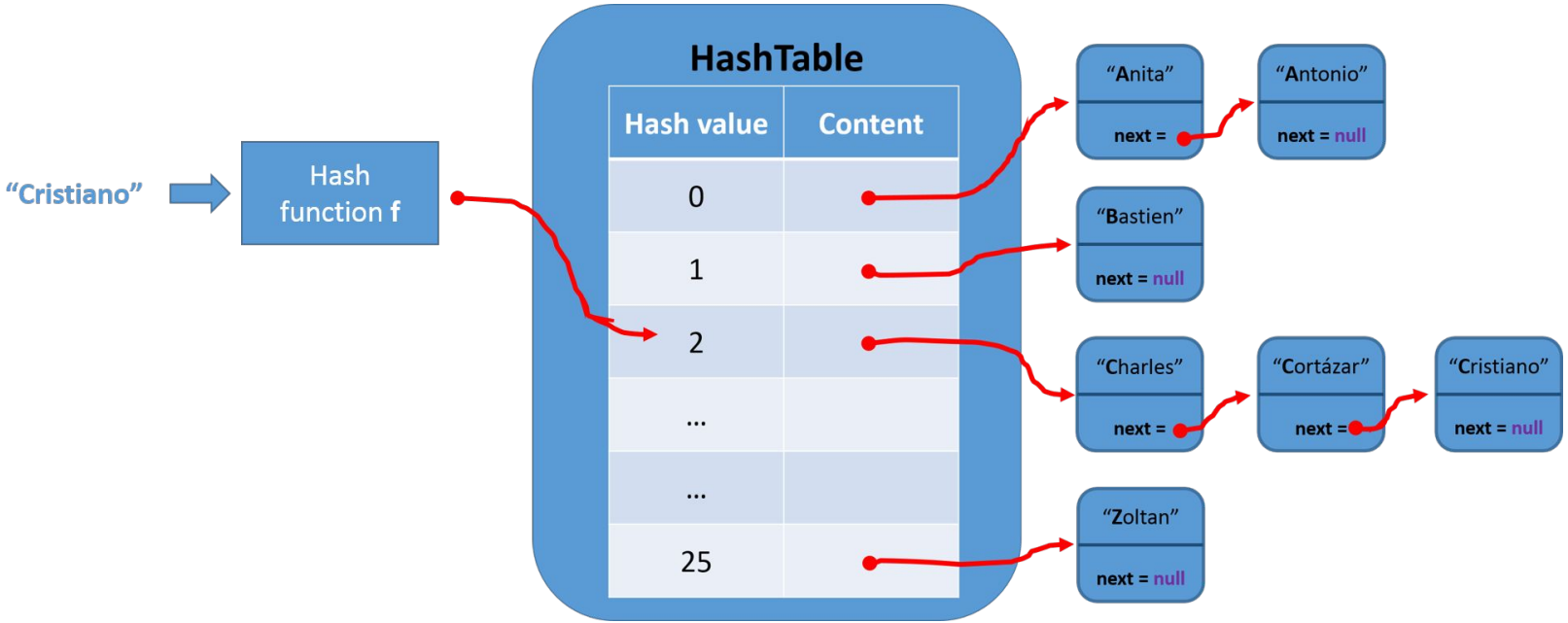
Trie

Clemens August, João Victor Falcão Santos Lima, Pedro Henrique de Brito e Rodrigo Santos da Silva

Motivação

Temos um conjunto de strings e precisamos realizar buscas eficientes dentro desse conjunto. Como fazer isso?

Hash Tables?

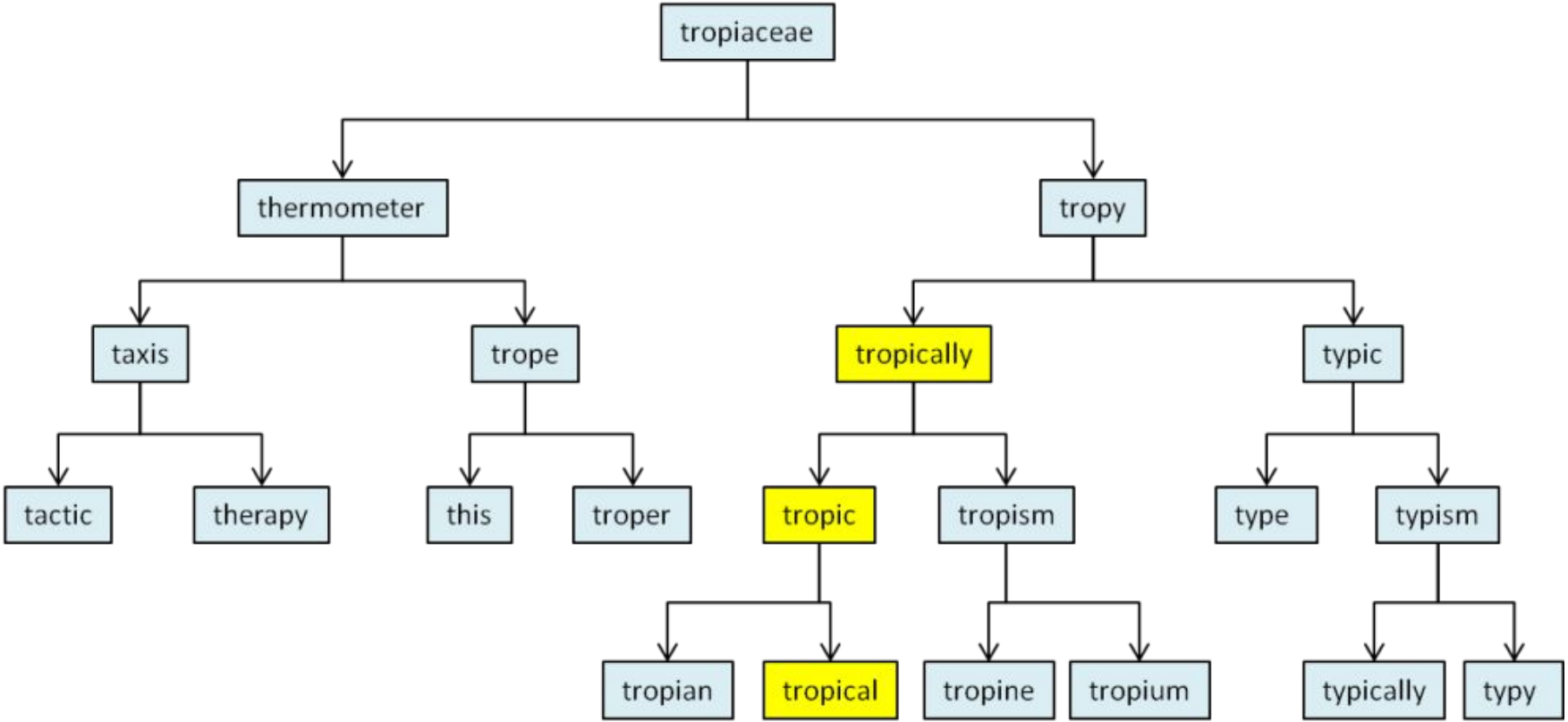

"Cristiano"
Hash function f
HashTable
Hash value
Content
0
1
2
...
...
25
"Anita"
next =
"Antonio"
next = null
"Bastien"
next = null
"Charles"
next =
"Cortázar"
next =
"Cristiano"
next = null
"Zoltan"
next = null

Problemas da Hash

- A busca por uma chave é feita em ***O(1)***, mas a busca por palavras é em ***O(n)***

- Não temos acesso a prefixos

AVLs?


tropiaceae
thermometer
tropy
taxis
trope
tropically
typic
tactic
therapy
this
troper
tropic
tropism
type
typism
tropian
tropical
tropine
tropium
typically
typy

Problemas da AVL

- Pior caso de uma busca é em ***O(m log n)***

- Operações de balanceamento podem ser custosas

- Também não tem acesso a prefixos

Como melhorar?

TRIE

O que é uma Trie?

- Tipo de árvore utilizada trabalhar com a associação de caracteres.

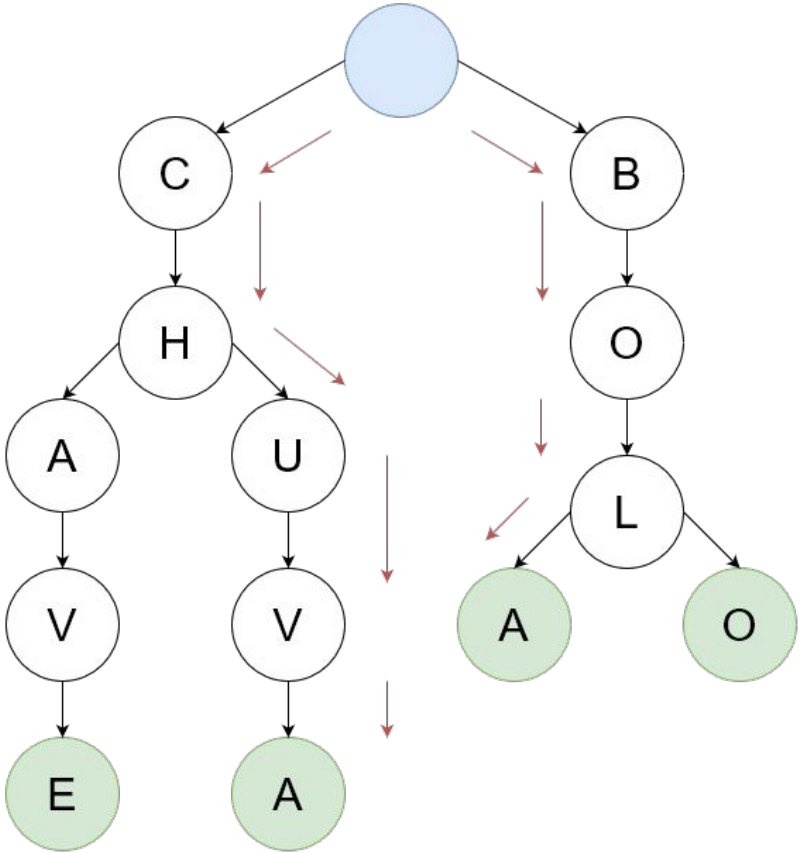
- Uma árvore utilizada para realizar buscas eficientes.

Estrutura da trie:

A struct de Trie tem dois elementos:

- Um array dicionário, onde a posição de cada ponteiro representa um caractere alfabético da tabela ASCII.

- Uma variável booleana que indica se aquele node é ou não o fim de uma palavra.



Exemplos:

CHAVE

CHUVA

BOLA

BOLO

Inserção

- Complexidade de tempo: ***O(m)*** (Para cada caractere, checamos ou criamos um node na Trie até chegar ao fim da string inserida).

- Complexidade de espaço: ***O(m)*** (No pior caso a string inserida não compartilha nenhum prefixo com as chaves até então inseridas)

Busca por Palavra/Prefixo

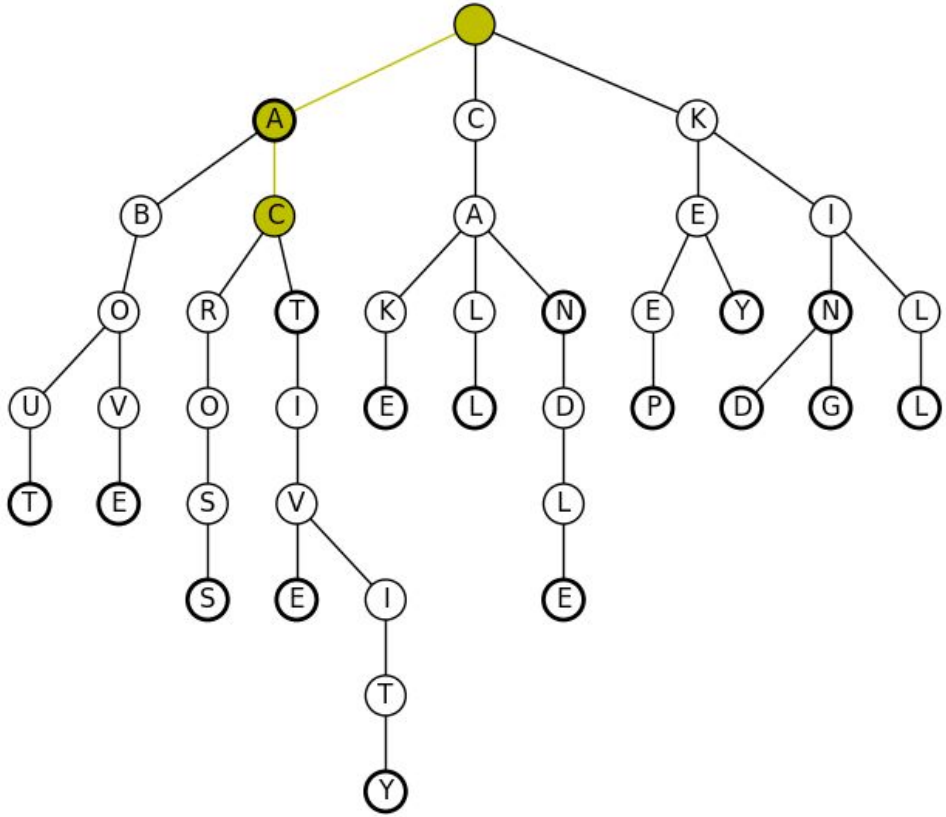
- Complexidade de tempo: ***O(m)***

  (Em cada passo do algoritmo procuramos a próxima chave de caractere)

- Remoção também em ***O(m)***

Word: AC

A	B	O	V
C	A	K	E
T	I	Y	E
G	N	D	P



TAD

```
TRIE* create_new_trie_node();

void insert_word(TRIE *head, char *string);

int search_word(TRIE *head, char *string);

int has_children(TRIE *node);

int remove_word(TRIE **node, char *string);

int empty_trie(TRIE *head);
```

```
#define CHAR_SIZE 26

struct trie

{

    int leaf;

    struct trie *character[CHAR_SIZE];

};

typedef struct trie TRIE;
```

```
TRIE *create_new_trie_node()
 {
    TRIE *node = (TRIE *)malloc(sizeof(TRIE));
    node->leaf = 0;
    int i;
    for (i = 0; i < CHAR_SIZE; i++){
        node->character[i] = NULL;
    }
    return node;
}
```

```
void insert_word(TRIE *head, char *string)
{
	TRIE *current = head;
	while (*string)
	{
		if (current->character[*string - 'a'] == NULL)
		{
			current->character[*string - 'a'] = create_new_trie_node();
		}
		current = current->character[*string - 'a'];

		string++;
	}
	current->leaf = 1;
}
```

```
int has_children(TRIE *node)
{
    int i;
    for (i = 0; i < CHAR_SIZE; i++){
        if (node->character[i]) return 1;
    }
    return 0;
}
```

```
int search_word(TRIE *head, char *string){
    if (head == NULL)  return 0;
    TRIE *current = head;
    while (*string){
        current = current->character[*string - 'a'];
        if (current == NULL)  return 0;
        string++;
    }
    return current->leaf;
}
```

```
int remove_word(TRIE **node, char* string){
        if (*node == NULL) return 0;
        if (*string){
                if (*node != NULL && (*node)->character[*string - 'a'] != NULL &&
                        remove_word(&((*node)->character[*string - 'a']), string + 1) &&
                        (*node)->leaf == 0) {
                        if (!has_children(*node)){
                                free(*node);
                                (*node) = NULL;
                                return 1;
                        }
```

```
        else  return 0;
        }
}
if (*string == '\0' && (*node)->leaf){
        if (!has_children(*node)){
                free(*node);
                (*node) = NULL;
                return 1;
        }
```

```
        else
                {
                        (*node)->leaf = 0;
                        return 0;
                }
        }
        return 0;
}
```

Visualizando a Trie e suas operações

https://www.cs.usfca.edu/~galles/visualization/Trie.html

Vantagens

- A Trie é favorecida contra AVLs, pois não precisa lidar com o problema de balanceamento na remoção/inserção;

- Com busca em **O(m)**, onde m = tamanho da palavra, a Trie é mais eficiente que Hash Tables (***O(n)***) e AVLs (***O(m log n)***);

- Possibilita a busca por prefixos;

- Pode economizar espaço em conjuntos menores de palavras com o mesmo prefixo.

Desvantagens

- Em quantidades pequenas de espaço a Trie possui desvantagem, pois precisa alocar memória para cada possível caractere.

Possíveis Aplicações

- Corretor automático dos telefones ou o completar automaticamente.

- Sistemas de cadastro com erros na hora da inserção

- IP routing (algoritmo de correspondência de prefixo)