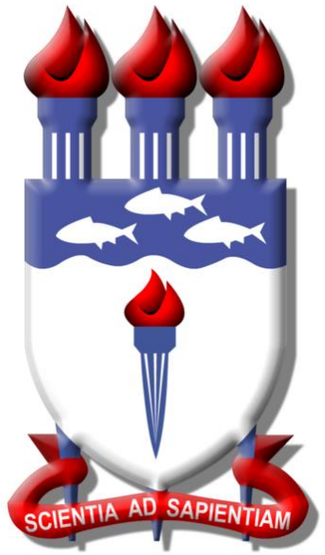


Trie

Bruno Lemos de Lima
João Pedro Cajueiro Marcolino
José Ferreira Leite Neto

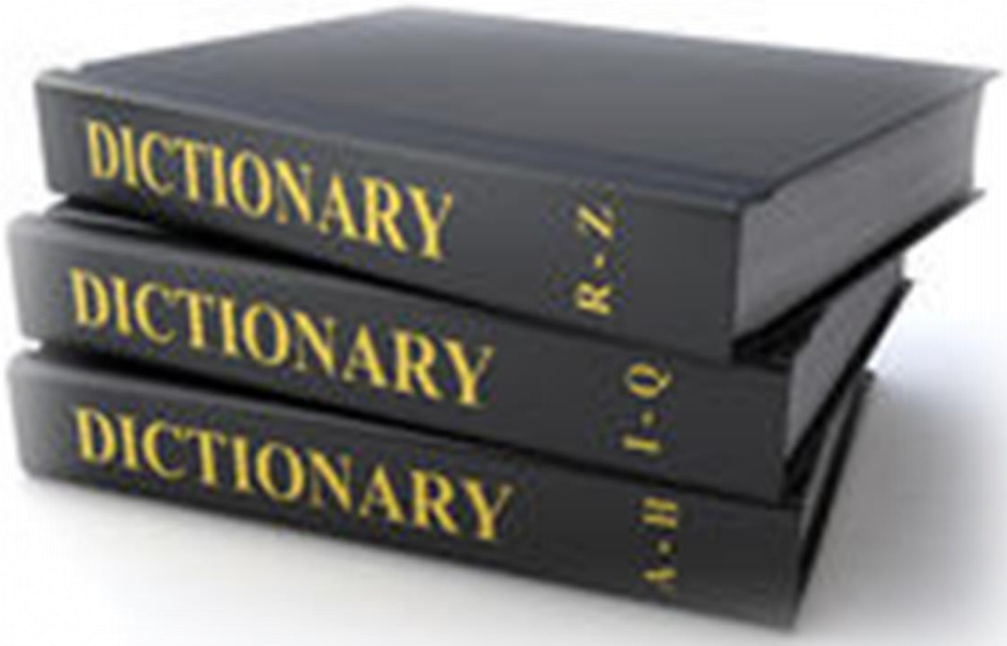
github.com/jflnetobr/huffman

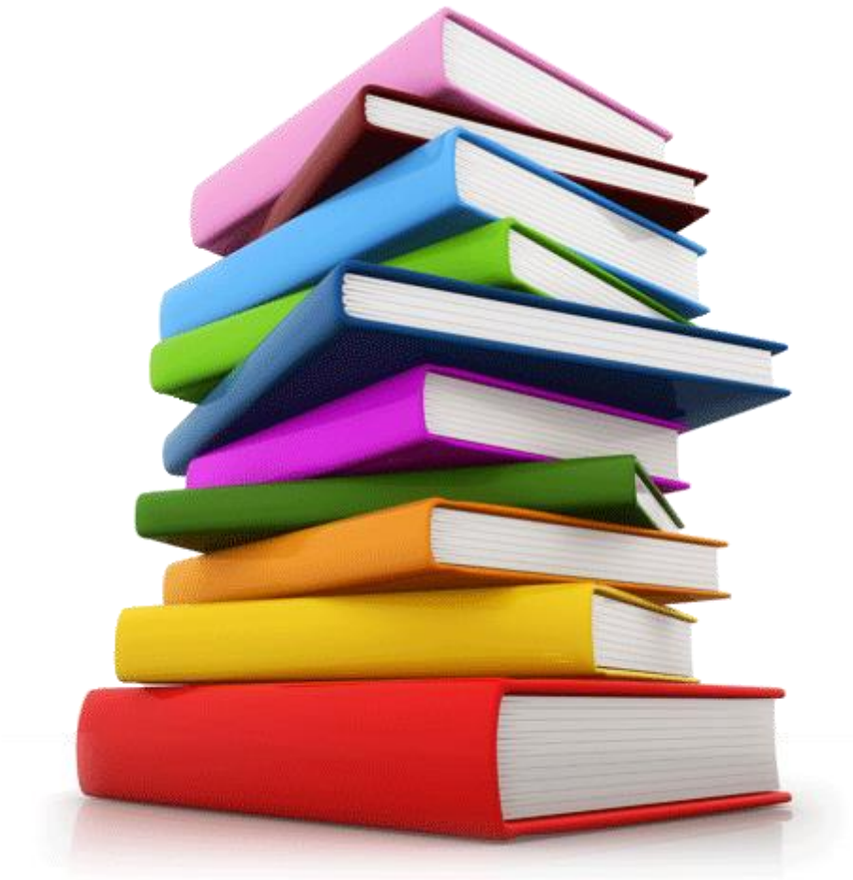
Motivação

- Armazenar **strings** de maneira que se possa buscá-las depois de maneira rápida;

Motivação

- Armazenar **strings** de maneira que se possa buscá-las depois de maneira rápida;





Usando o que conhecemos

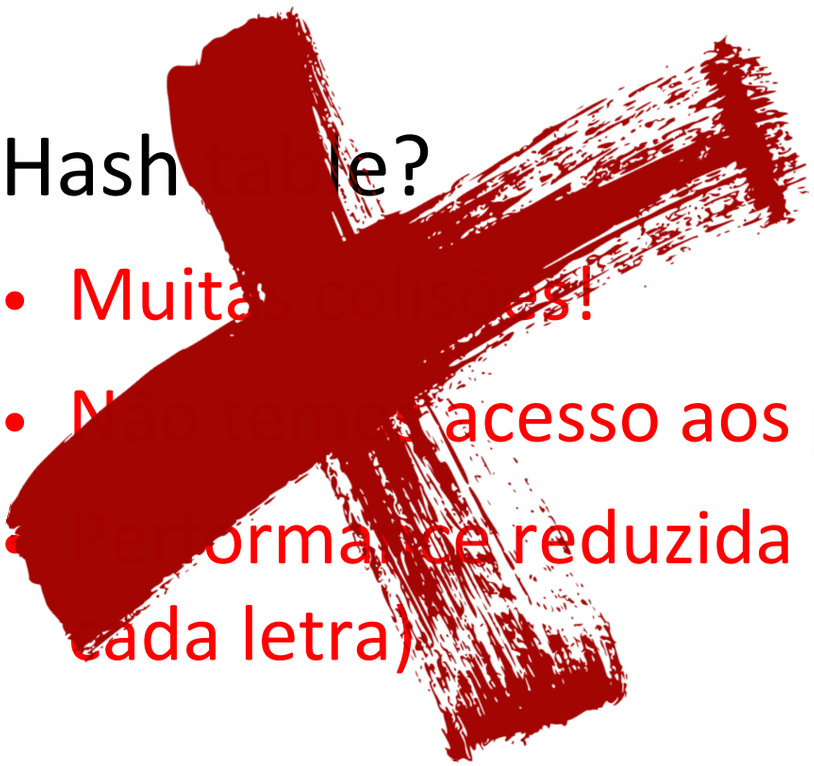
- Hash table?

Usando o que conhecemos

- Hash table?
  - Muitas colisões!
  - Não temos acesso aos prefixos!
  - Performance reduzida (teremos que percorrer a lista associada a cada letra)

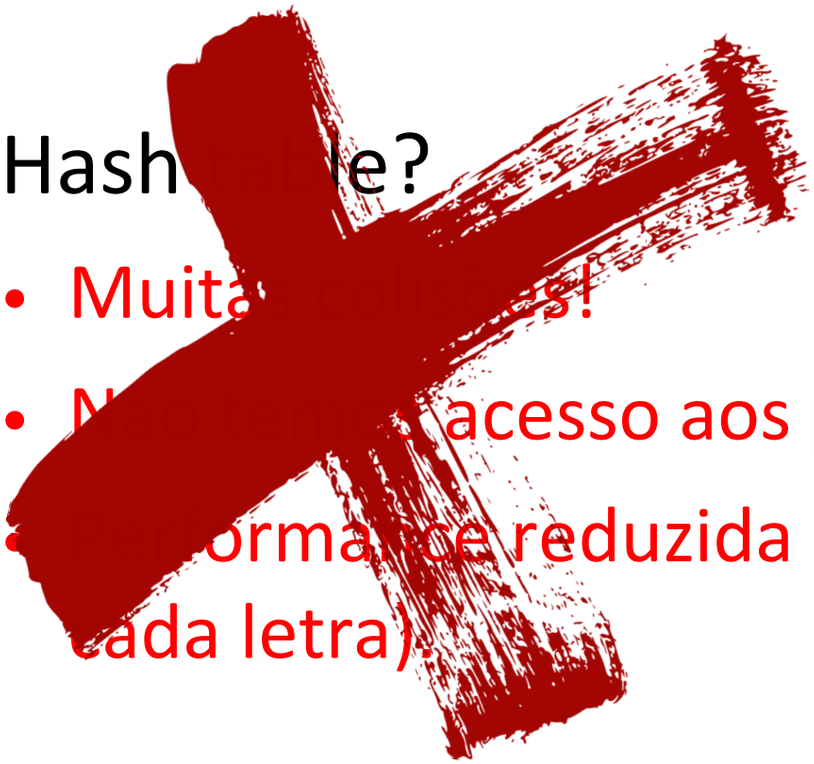
Usando o que conhecemos

- Hash table?
  - Muita memória!
  - Não permite acesso aos prefixos!
  - Performance reduzida (teremos que percorrer a lista associada a cada letra)



- AVL?

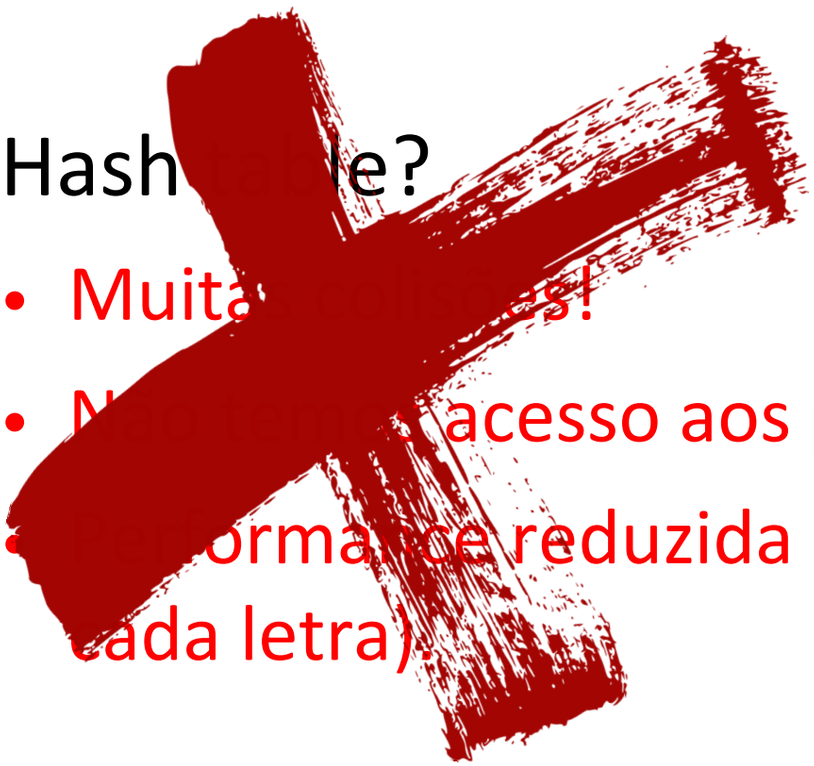
Usando o que conhecemos



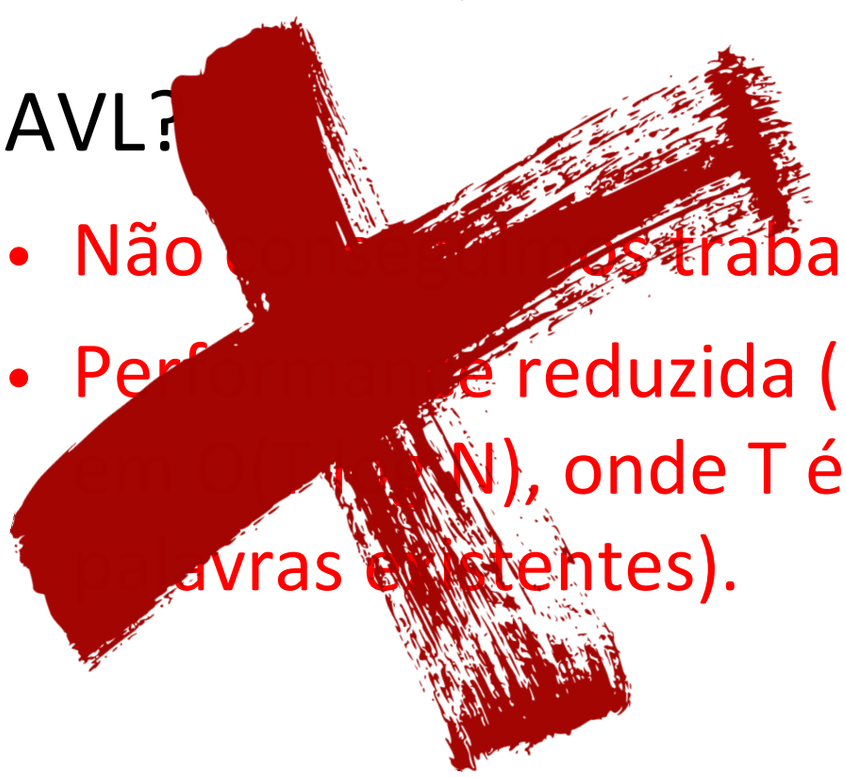
- Hash table?
  - Muita memória!
  - Não permite acesso aos prefixos!
  - Performance reduzida (teremos que percorrer a lista associada a cada letra).
- AVL?
  - Não conseguimos trabalhar direito com os prefixos!
  - Performance reduzida (na AVL, inserimos, buscamos e deletamos em O(T log N), onde T é o tamanho da palavra e N a quantidade de palavras existentes).

Usando o que conhecemos

- Hash table?
  - Muitas colisões!
  - Não permite acesso aos prefixos!
  - Performance reduzida (teremos que percorrer a lista associada a cada letra).



- AVL?
  - Não consegue trabalhar direito com os prefixos!
  - Performance reduzida (na AVL, inserimos, buscamos e deletamos em O(T log N), onde T é o tamanho da palavra e N a quantidade de palavras existentes).

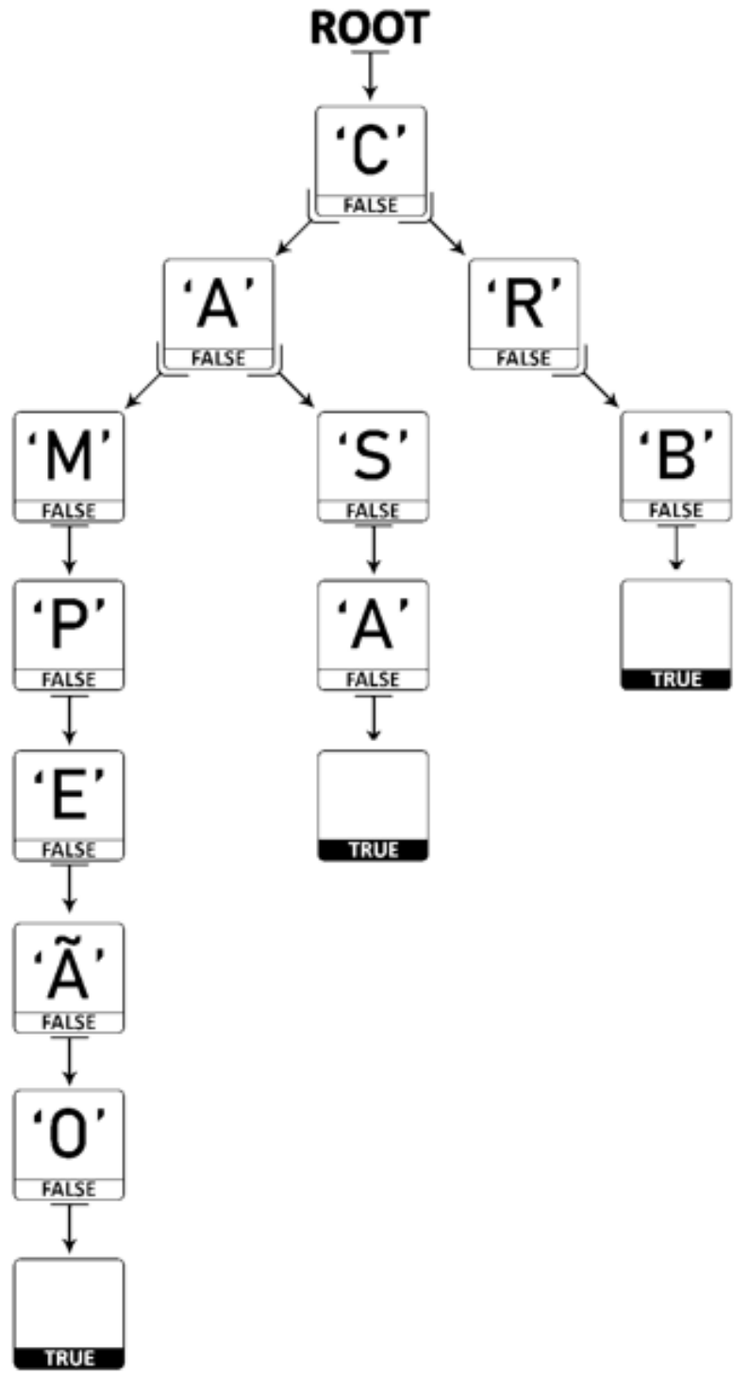


TRY

TRIE!

Trie

- Árvore ordenada que guarda, normalmente, cadeias de caracteres;
- Foi definida em 1960 por Edward Fredkin e seu nome vem de re**trie**val;
- Normalmente se pronuncia **try** para não haver confusão com tree;
- Permite também encontrar prefixos em comum entre as palavras (o que pode ser muito útil).



Conceitos importantes

- Cada nó tem um array de ponteiros, cujo tamanho é o tamanho do alfabeto utilizado;

- Um nó vazio com o atributo **isWord** verdadeiro marca o final de uma palavra;

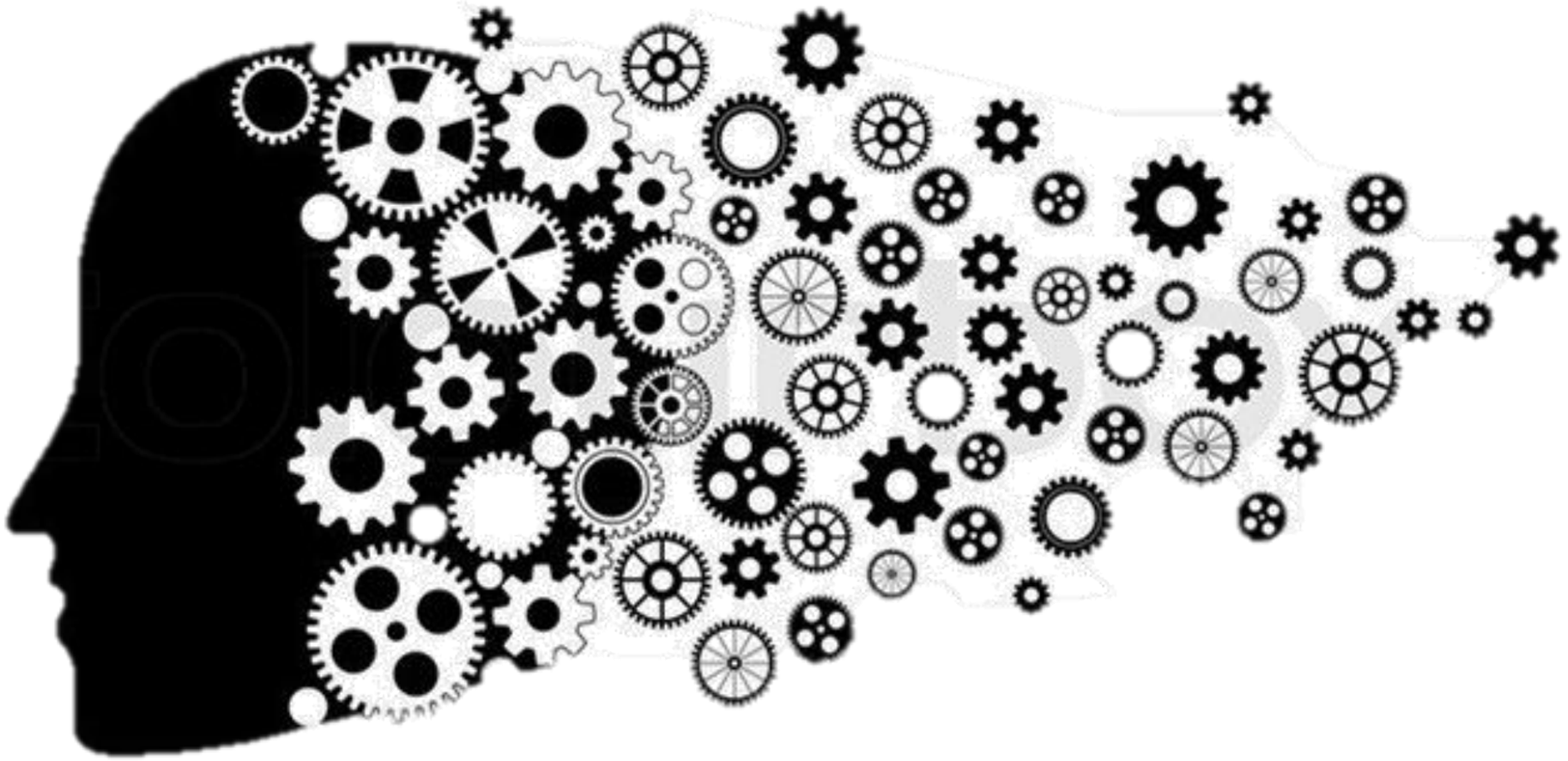
- A estrutura da Trie não depende da ordem em que as chaves são inseridas (ao contrário da BST);

- O consumo de tempo das operações não depende do número de chaves presentes;

- **Um nó NÃO armazena a chave, conhecemos a chave através do caminho feito pelos ponteiros.**

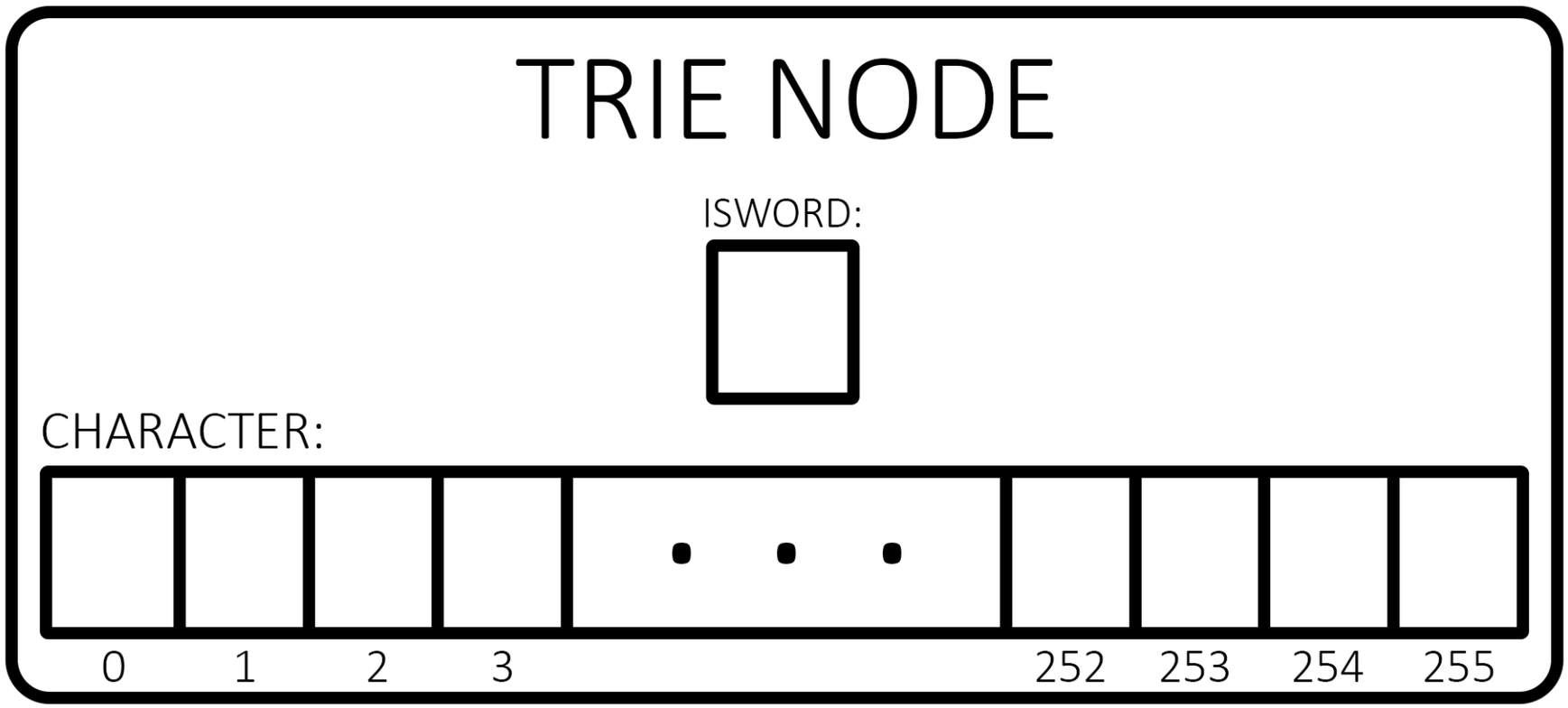
ADT

```
trie *create_trie();
int is_empty(trie *current);
void insert(trie *root, char *string);
int search(trie *root, char *string);
trie *delete(trie *root, char *string, int depth);
```



Struct

```
struct trie_node {
   int isWord;
   struct trie_node *character[CHARACTER_SIZE];
};
```



Código

```
trie_node *create_trie()
{
   int i;
   trie_node *new_node = (trie_node*) malloc(sizeof(trie_node));

   new_node->isWord = 0;

   for(i = 0; i < CHARACTER_SIZE; i++)
   {
      new_node->character[i] = NULL;
   }

   return new_node;
}
```

Código

```
int is_empty(trie_node *current)
{
   int i;
   for(i = 0; i < CHARACTER_SIZE; i++)
    {
      if (current->character[i]) return 0;
    }
   return 1;
}
```

Código

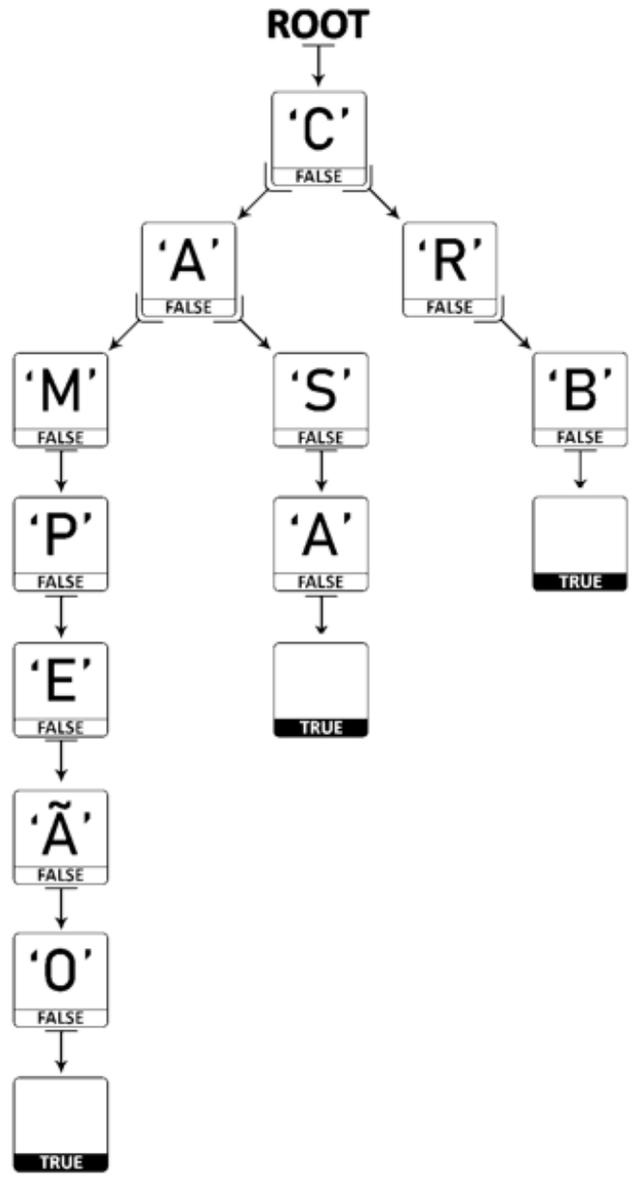
```
void insert(trie_node *root, char *string)
{
   int i;
   trie_node *current = root;

   for(i = 0; i < strlen(string); i++)
   {
      if (current->character[string[i]] == NULL)
      {
         current->character[string[i]] = create_trie();
      }
      current = current->character[string[i]];
   }
   current->isWord = 1;
}
```

Código

```
int search(trie_node *root, char *string)
{
   if (root == NULL) return 0;

   int i;
   trie_node *current = root;

   for(i = 0; i < strlen(string); i++)
   {
      current = current->character[string[i]];
      if (current == NULL) return 0;
   }

   return current->isWord;
}
```

Código

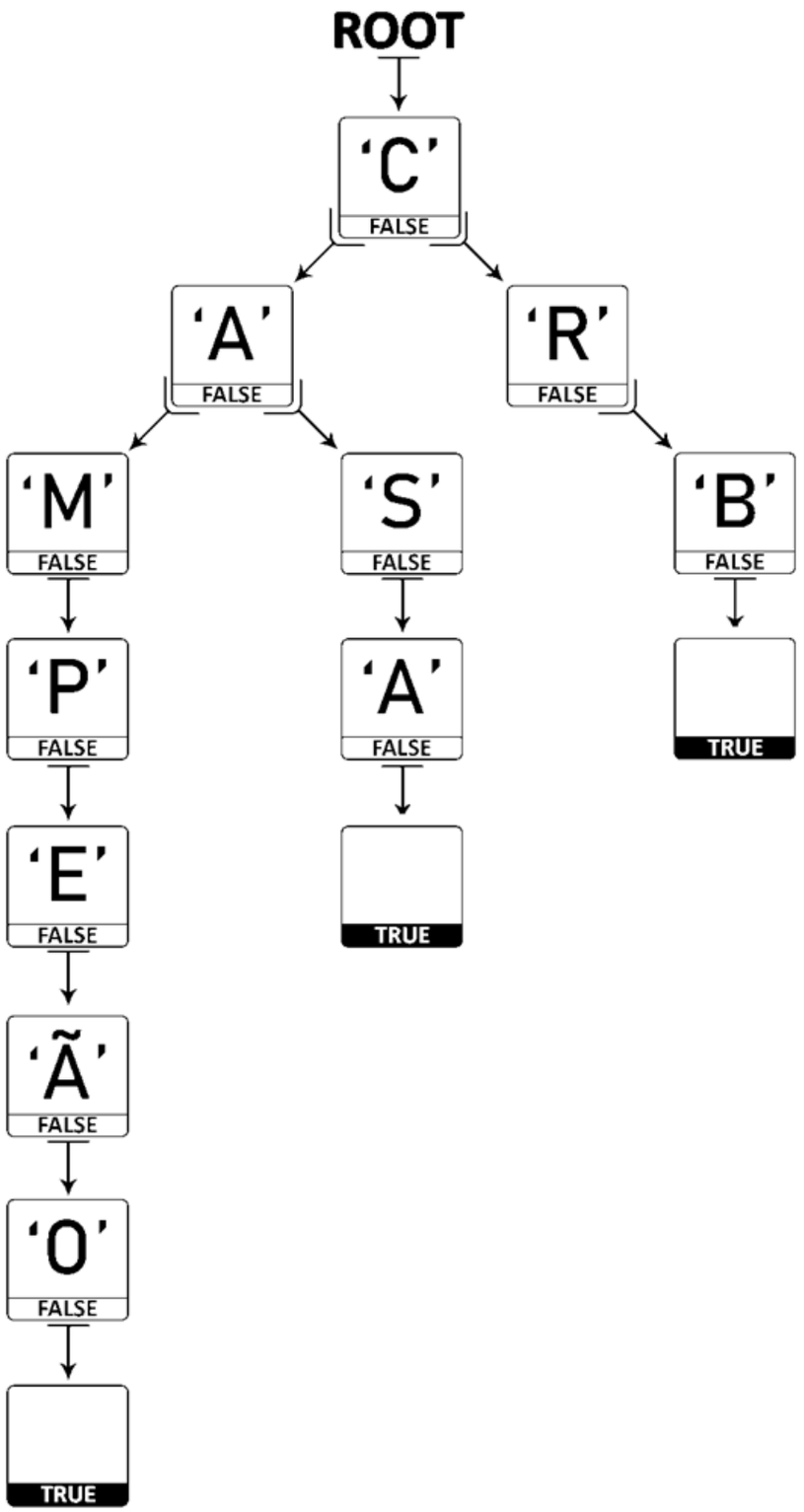
```
trie *delete(trie_node *root, char *string, int depth)
{
    if(root == NULL) return NULL;

    if(depth == strlen(string))
    {
        if(root->isWord == 1) root->isWord = 0;

        if(is_empty(root))
        {
            free(root);
            root = NULL;
        }

        return root;
    }

    root->character[string[depth]] = delete(root->character[string[depth]], string, depth+1);

    if(is_empty(root) && root->isWord == 0)
    {
        free(root);
        root = NULL;
    }

    return root;
}
```



Inserção

ROOT

Exclusão


ROOT
'C'
FALSE
'A'
FALSE
'R'
FALSE
'M'
FALSE
'S'
FALSE
'B'
FALSE
'P'
FALSE
'A'
FALSE
TRUE
'E'
FALSE
TRUE
'Ã'
FALSE
'O'
FALSE
TRUE

De volta à Motivação...

- Armazenar **strings** de maneira que se possa buscá-las depois de maneira rápida;

De volta à Motivação...

- Eficiência (onde T é o tamanho da chave):

Inserção

O(T)

Exclusão

O(T)

Busca

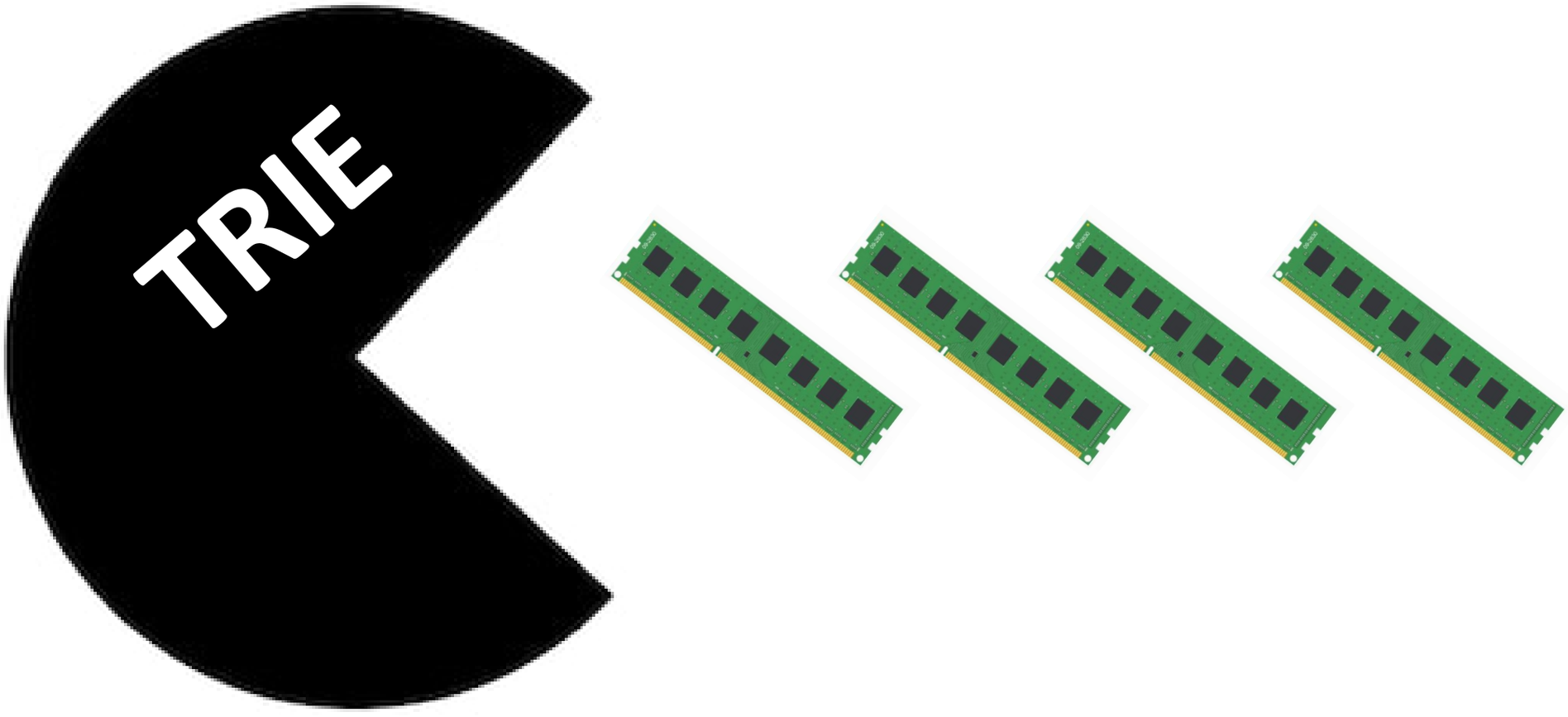
O(T)

- Além disso, conseguimos trabalhar com os prefixos (saber os prefixos em comum entre as palavras, por exemplo);


HOUSTON
WE HAVE A
PROBLEM

Problema

- Alto consumo de memória para chaves grandes (cuidado com o tamanho do alfabeto utilizado).
- Para palavras grandes: **TST (Ternary search tree)**



Aplicações

- Manuseamento de dicionários;
- Corretor ortográfico;
- Sistema de sugestão de palavras;
- Pesquisa em textos de grande dimensão.