# Trabalho Prático 2: Corretor Ortográfico

[1]Alexandre Aparecido Scrocaro Junior R.A.: 2135485, [2]Pedro Acácio Rodrigues R.A.: 2135655
*Universidade Tecnológica Federal do Paraná – UTFPR*
*COCIC – Coordenação do Curso de Bacharelado em Ciência da Computação*
*Campo Mourão, Paraná, Brasil*
[1]alexandre.2001@alunos.utfpr.edu.br
[2]pedrorodrigues.2019@alunos.utfpr.edu.br

## Resumo

*Esse projeto tem como objetivo explorar uma solução para correções ortográficas por meio do uso de TRIEs. O uso dessa estrutura de dados possibilita análise ortográfica letra-a-letra de acordo com as regras especificadas e uma regra extra criada.*

## 1. Modularização

Para a implementação desenvolvemos TADs para modularizar o projeto e assim ambos os participantes puderam desenvolver em conjunto, os TADs utilizados foram:

- TAD_Corretor:

Esse módulo foi criado para tratar das palavras incorretas e sugerir soluções.

arquivoParaLista: Lê o arquivo, percorre até o final selecionando seus caracteres e quando encontra uma palavra verifica se está incorreta e a adiciona na lista, após isso retorna a lista de palavras erradas.

PreencherTrieAux: Insere a lista de correções em uma trie.

regraUm: Insere um coringa (*) em cada uma das posições da String de forma alternada, chama a função TRIE_ChavesQueCasam para criar uma lista com todas as correções possíveis e assim preenche a trie correções.

regraDois: O tamanho da String é diminuído conforme os valores especificados pelo professor, chama função TRIE_ChavesComPrefixo com a nova String para criar uma lista com todas as correções possíveis e assim preenche a trie correções.

regraTres: Pega o maior prefixo da palavra passada, cria uma lista com todas as chaves que casam com esse prefixo, cria-se uma lista com o maior prefixo e preenche a trie com ele.

regraTeste: Alterna dois coringas entre as posições de uma palavra incorreta, verifica as chaves que casam com essa nova String e preenche a trie. Dessa forma é possível sugerir soluções para palavras que possuam 2 caracteres incorretos em posições aleatórias. Nesta regra, havia sido feita a mesclagem das regras 1 e 2, porém notamos que recriando a regra 1 de forma duplicada encontra-se melhores sugestões para palavras incorretas.

CorrigirOrtografia: A partir da lista com as palavras erradas, aplica-se as regras para cada palavra da lista, cria uma nova lista com as sugestões partindo da trie preenchida pelas regras, ordena-se a lista de sugestões conforme suas exigências e após isso a lista é impressa. Ao fim da função tem-se os resultados inseridos na tabela de análise.

- TRIE:

Esse módulo foi utilizado para conter as funções que manuseiam a estrutura de dados TRIE.

AT_Buscar: Preenche AT_Buscar_R para o início da busca e chama a função recursiva.

AT_Buscar_R: Busca nó a nó até completar a chave, ao final retorna o nó ou NULL para quando a palavra não existe na trie.

AT_Criar: Criação de um nó da trie;

AT_Inserir: Preenche AT_Inserir_R para o início da busca e chama a função recursiva e atualiza o tamanho da estrutura.

AT_Inserir_R: Insere na trie cada caractere contido na chave.

AT_Imprimir: Preenche AT_Imprimir_R para o início da busca e chama a função recursiva.

AT_Imprimir_R: Mostra a trie, delimitando o nível do nó (-) e apresenta o estado dele (‘O’ ou ‘L’).

AT_Limpa: Desaloca a trie.

ConstruirDicionario: Insere as palavras do arquivo dicionário.txt na trie.

- TAD_FuncoesChave:

Esse módulo foi implementado para tratar de funções que possuem relações diretas com a chave. Lembrando que todas as funções não recursivas criam uma lista.

TRIE_ChavesComPrefixo: Busca o prefixo na trie e passa o nó obtido para a função recursiva e retorna a lista resultante.

TRIE_ChavesComPrefixo_R: Realiza os casos base para verificar se a trie está vazia; de forma recursiva verifica os filhos do nó atual até obter as chaves com o prefixo passado.

TRIE_ChavesQueCasam: Adiciona um coringa (*) ao final do padrão passado, chama-se a função recursiva e ao final retorna a lista das chaves que casam.

TRIE_ChavesQueCasam_R: Verifica o caso base para a trie igual a NULL, recursivamente analisa todos os filhos do padrão passado e insere na lista todas as chaves que casam com esse padrão.

TRIE_ChaveMaiorPrefixoDe: Chama a função recursiva, verifica o maior prefixo da lista resultante através da função maiorPrefix e o retorna.

TRIE_ChaveMaiorPrefixoDe_R: Busca no dicionário a String passada, verifica recursivamente uma String com um caractere a menos que a anterior, e salva todos os prefixos possíveis em uma lista.

maiorPrefix: Verifica o maior prefixo possível da lista que seja uma palavra e o retorna.

- TAD_Lista:

A estrutura de dados escolhida para armazenar palavras incorretas, corretas e todas do texto foi a lista, enfim chaves. Também temos nesse TAD a criação das STRUCT Lista e do nó, neste TAD temos as seguintes funções.

criarLista: Criação de uma estrutura lista e de um nó vazio para onde a lista aponta.
atribuirPalavra: Insere uma palavra (chave) na estrutura.
imprimirLista: Mostra o conteúdo de todos os integrantes da estrutura.
bubbleSort: Utilizado para ordenação da estrutura, usado na lista de sugestões.

## 2. Análise

Fazendo uma análise das correções, vemos que as melhores sugestões seguem a regra 1. Por outro lado, a regra 2 tende a falhar em alguns casos apresentando palavras desnecessariamente grandes, a regra extra criada se mostra eficaz em palavras maiores com dois erros, porém quando se trata de menos caracteres suas sugestões são pouco precisas. Ao analisar a tabela obtida, vemos que o número médio de sugestões por palavra incorreta se mantém constante junto ao tempo de execução que apenas aumenta, quando o número de palavras cresce drasticamente

- **Tabela 1:**

| | Total de palavras | Palavras incorretas | Número médio de sugestões por palavra incorreta | Tempo de execução (s) |
|---|---|---|---|---|
| casmurro1.txt | 826 | 68 | 43 | 0.93 |
| casmurro2.txt | 530 | 46 | 32 | 0.451 |
| casmurro3.txt | 452 | 39 | 35 | 0.398 |
| casmurro4.txt | 797 | 57 | 59 | 0.979 |
| memposthumas1.txt | 697 | 98 | 30 | 0.662 |
| memposthumas2.txt | 529 | 45 | 26 | 0.386 |
| memposthumas3.txt | 563 | 46 | 55 | 0.865 |
| casmurro_all.txt | 2605 | 212 | 43 | 2.558 |
| memposthumas_all.txt | 1789 | 159 | 36 | 1.75 |
| all.txt | 4394 | 368 | 41 | 5.305 |

- Considerações adicionais:

Ao fazer alguns free's, dava erro dentro da biblioteca stdlib, então decidimos fazer somente outros com certeza que não fariam diferença, negativa, para o funcionamento do código.

## 3. Distribuição do projeto

O trabalho não foi dividido em partes para os dois alunos, ambos participaram de todas as etapas de produção do trabalho, sendo realizado da seguinte forma:

- Código: para a interpretação, elaboração da lógica e desenvolvimento dos arquivos(.c e .h), nos reunimos pelo aplicativo Discord usando chamadas de voz e compartilhamento de tela;

- Relatório: para escrever o presente documento, também fizemos uso do discord para chamadas de voz e da ferramenta online Docs, do google; desse modo conseguimos redigir o relatório de forma simultânea.