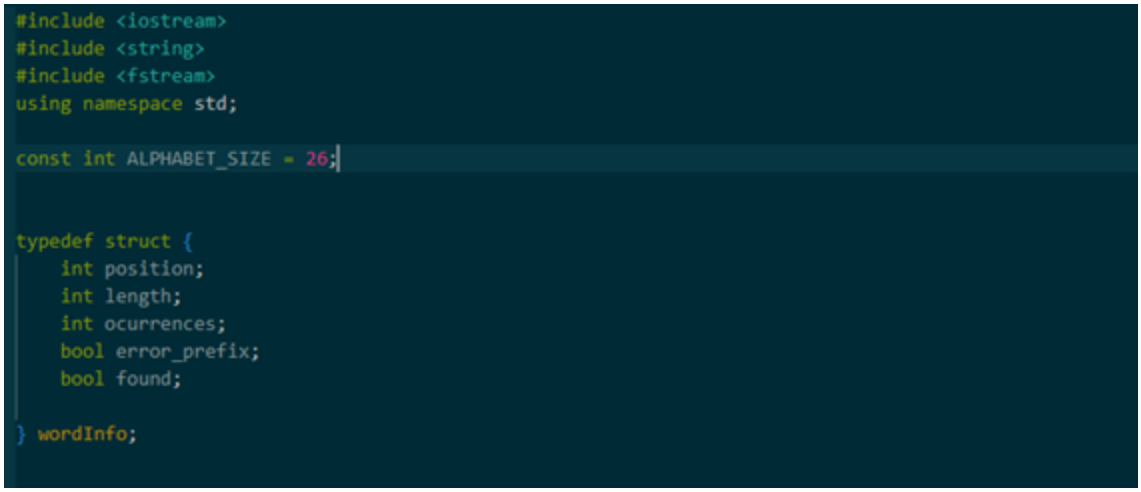
RELATÓRIO PROJETO II

Alunos: Luan Diniz Moraes *(21204000)*
Leonardo Guimaraes de Melo Brito *(21200896)*

Matéria: Estrutura de Dados (INE5408, turma 03208/A),
08 de dezembro de 2022.

Obs: Nossa implementação se baseou no link disponiblizado pelo professor:

https://www.geeksforgeeks.org/trie-insert-and-search/



```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

const int ALPHABET_SIZE = 26;


typedef struct {
    int position;
    int length;
    int ocurrences;
    bool error_prefix;
    bool found;

} wordInfo;
```

(Bibliotecas utilizadas e a declaração de um novo tipo)

PROBLEMA: Identificação de prefixos e indexação de dicionários.

Os dois problemas foram trabalhados e resolvidos simultaneamente na implementação da *Trie*. Basicamente, a função *search* retorna toda a informação que queremos através de um retorno com o tipo *wordInfo*. Após isso, a função *main* avalia esses dados e faz as impressões adequadas na tela.

1. A Trie:

```
// nó trie
struct TrieNode
{
    struct TrieNode *children[ALPHABET_SIZE];
    int position;
    int length;
    bool is_root;
    bool isEndOfWord;
};
```

(A *struct* base da Trie.)

Cada nó da *struct* consiste em um vetor (tamanho 26) de outros *TrieNode,* onde cada posição representa uma letra do alfabeto. Além disso, informações da posição e tamanho são armazenados (caso aquele nó represente o fim de uma palavra).

A Trie tem as seguintes funções: *getNode*, *insert*, *findOcurrences*, *search* e *deleteTrie*.

```
// Retorna um novo nó da trie
struct TrieNode *getNode(bool root)
{
    struct TrieNode *pNode = new TrieNode;

    pNode->isEndOfWord = false;
    pNode-> is_root = root;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = nullptr;

    return pNode;
}
```

O *getNode* inicializa um novo nó, e seu único argumento serve para definir se aquele nó é a raiz da árvore ou não.

```
void insert(struct TrieNode *root, string key, int pos, int length)
{
    struct TrieNode *pCrawl = root;

    for (int i = 0; i < key.length(); i++)
    {
        int index = key[i] - "a"[0];
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode(false);

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isEndOfWord = true;
    pCrawl->position = pos;
    pCrawl->length = length;
}
```

Já o *insert*, recebe a raiz da árvore (que deve ser criada no *main*) além de uma palavra e mais dois argumentos, posição e largura (foram calculados no *main*).

A função vai percorrer os nós da árvore, com base em cada caractere da palavra. O *index* é uma representação em inteiro para os caracteres, cada posição do vetor *children* representa uma letra do alfabeto.

Os nós que representam a letra que está sendo iterada serão acessados (ou criados, caso eles não existam ainda) e, na última iteração, marca-se no nó que ele representa o final de uma palavra e também armazena-se os dados *position* e *length* nele.

```
int findOcurrences(struct TrieNode *node, int &cont) {
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (node->children[i] != nullptr) {
            findOcurrences(node->children[i], cont);
        }

        if (i == ALPHABET_SIZE - 1 and node->isEndOfWord){
            cont += 1;
        }
        }
    return cont;
}
```

O *find_Ocurrences* foi utilizado somente dentro da função *search* e tem como objetivo contar a quantidade de prefixos recursivamente. Os argumentos a serem passados são um contador (inicializado em 0) e o nó onde está o fim do prefixo a ser analisado. Ele vai varrer todo

o resto da árvore em busca de nós com o *TrieNode->isEndofWord == true.* Cada vez que encontrar, o contador aumenta em 1.

```
wordInfo search(struct TrieNode *root, string key) {
    struct TrieNode *pCrawl = root;
    wordInfo info;
    info.error_prefix = false;
    //info.found recebera uma atribuicao mais a frente
    int cont_ocurrences = 0;

    for (int i = 0; i < key.length(); i++) {
        int index = key[i] - "a"[0];
        if  (pCrawl->children[index] != nullptr) {
            pCrawl = pCrawl->children[index];
        } else {
            info.error_prefix = true; //marca que o prefixo nao existe)
        }
    }
    //analisa se a palavra existe
    info.found = pCrawl->isEndOfWord;

    if (info.found) {
        info.position = pCrawl->position;
        info.length = pCrawl->length;
    } else {
        info.position = 0;
        info.length = 0;
    }

    if (!info.error_prefix) {
        info.ocurrences = findOcurrences(pCrawl, cont_ocurrences);

    } else {
        info.ocurrences = 0;
    }

    return info;
}
```

A função *search* é responsável por chamar o *findOcurrences* para contar os prefixos, além de organizar as informações da palavra e retornar com o *info.* Ela também verifica se a entrada é um prefixo e também se a palavra existe na Trie.

Note: A Trie é percorrida de forma a achar o nó que representa a última letra da string de entrada, com essa informação, já é possível chamar o *findOcurrences* e contar as ocorrências do prefixo.

```
//funcao recursiva para deletar a trie
void deleteTrie(struct TrieNode* node) {
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (node->children[i] != nullptr) {
            deleteTrie(node->children[i]);
        } else {
            delete node->children[i];
        }
    }
    if (node->is_root = true) {
        delete node;
    }
}
```

O nome é autoexplicativo, a *deleteTrie* percorre a Trie inteira deletando seus nós (recursivamente), até que em determinado momento reste somente a raiz da árvore, que também é deletada.

2. **A main( ):**

```
131  int main() {
132      struct TrieNode *root = getNode(true);
133
134      string filename;
135      string word = "";
136      string line;
137      string input;
138      int position = 0;
139      int next_word;
140      wordInfo information;
141
142
143      cin >> filename;
144
145      ifstream file;
146      file.open(filename);
```

(Declaração de variáveis, criação da raiz da Trie)

```
cin >> filename;

ifstream file;
file.open(filename);

// Armazena a trie
if (file.is_open()) {
    while (!file.eof()) {  // Leitura do arquivo dict
        getline(file,line);
        for (int i = 1; i < line.length(); i++) {
            if (line[i] == ']')
                break;
            word += line[i];
        }
        insert(root, word, position, line.length());
        position = position + line.length() + 1;
        word = "";
    }

    file.close();

} else {
    throw out_of_range("Não foi possível abrir o arquivo");
}
```

Aqui o algoritmo lê o nome do arquivo e abre-o. Então ele percorre todas as linhas do arquivo procurando por um ‘]’, pois é um padrão e sabe-se que há uma palavra antes dele.

Armazena-se a string da palavra em *word* e ela é inserida dentro da Trie junto com as informações *position* e *length*. (A largura é encontrada por *line.length()* e a posição é encontrada pela *line.length()* da linha anterior + 1.)

```
    // Procura pelas palavras até chegar em 0
    while (true) {
        cin >> input;
        if (input == "0") {
            break;
        } else {
            information = search(root, input);

            if (information.error_prefix == 1) {
                cout << input << " is not prefix" << endl;
            } else {
                cout << input << " is prefix of " << information.ocurrences << " words" << endl;

                if (information.found) {
                    cout << input << " is at (" << information.position << "," << information.length << ")" << endl;
                }

            }

        }

    }

    deleteTrie(root);
    return 0;
}
```

Por fim, há um loop que fica lendo o restante das entradas (agora são palavras que queremos ter informações) e ele só é quebrado quando a entrada for ‘0’.

A função *search* é chamada e ela retorna toda a informação que precisamos através do tipo *wordInfo*. Após isso, esses dados são interpretados por estruturas condicionais que irão dar o output desejado.

Após o *deleteTrie* e o *return 0*, a execução do algoritmo foi concluída.

DIFICULDADES

Não houveram muitas dificuldades durante a execução do trabalho, foram basicamente três:

1. Alguns erros ocorreram pois nós estávamos inicializando variáveis estáticas dentro de *structs*.

2. Não estávamos acostumados a pensar recursivamente, então demorou um tempo até pensarmos no algoritmo. (E também houve um pouco de confusão com o que estava sendo considerado prefixo, pois no contexto do trabalho o significado é diferente do convencional, como por exemplo a palavra ser um prefixo dela mesma).

3. Tínhamos entendido que era necessário tratar as entradas, ou seja, pensamos em um algoritmo que percorria todo o input procurando por espaços em branco e separando as palavras, porém isso causou erros na lógica de execução e então percebemos que o C++ estava tratando isso automaticamente.