UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
INE-DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

INE5408 - ESTRUTURA DE DADOS

PEDRO HENRIQUE TAGLIALENHA (22203674)
MATHEUS FERNANDES BIGOLIN (22200371)

RELATÓRIO PROJETO 2

FLORIANÓPOLIS
2023

Sumário

1 INTRODUÇÃO

A indexação e recuperação eficiente de palavras em grandes arquivos de dicionários são desafios cruciais em diversas aplicações. Neste projeto, abordaremos a construção e utilização de uma estrutura hierárquica conhecida como *trie*, derivada do termo em inglês “retrieval” e também referenciada como árvore de prefixos ou árvore digital. A finalidade principal desta implementação é resolver dois problemas fundamentais. O primeiro consiste na identificação de prefixos, onde a *trie* será construída em memória principal a partir das palavras contidas no arquivo de dicionário. A aplicação deverá então determinar se uma série de palavras, pertencentes ou não ao dicionário, é um prefixo de outras palavras, fornecendo a quantidade de palavras correspondentes. O segundo problema aborda a indexação do arquivo de dicionário, incorporando à *trie* informações sobre a localização e tamanho de cada palavra. Ao receber uma palavra, a aplicação deve identificar sua presença no dicionário e, se for o caso, indicar se é também um prefixo de outras palavras, fornecendo as coordenadas da palavra no arquivo.

Esta abordagem visa otimizar a busca e recuperação de informações em dicionários extensos, proporcionando uma solução eficaz para o processamento e análise de grandes volumes de dados lexicais. A estrutura proposta, baseada na *trie*, oferece uma organização hierárquica que agiliza as operações de busca, proporcionando respostas rápidas e eficientes na identificação de prefixos e indexação de palavras em arquivos de dicionários extensos.

2 CÓDIGO DESENVOLVIDO

Na seguinte seção, será apresentado uma análise detalhada da lógica envolvida nos blocos de comandos do projeto, que foi estruturado em dois arquivos principais: o `main.cpp` e o `dicio.h`. O arquivo `main.cpp` inicia definindo funções essenciais, como a `abrir_arquivo`, responsável por abrir e verificar a existência de um arquivo. A implementação principal ocorre na função de entrada (`main`), na qual ocorre a leitura do arquivo de dicionário, a construção da *trie*, e a verificação de prefixos, seguida pela indexação das palavras. O arquivo `dicio.h` contém a definição da classe `Dicionario`, que implementa a *trie* e fornece métodos para inserção, verificação de prefixos e obtenção de indexação.

A estrutura do código foi organizada buscando eficiência na manipulação da *trie* e garantindo a integridade das operações realizadas. Ao longo desta seção, será detalhado o funcionamento de cada bloco de código, esclarecendo a implementação da *trie*, a inserção de palavras, e as operações de verificação de prefixos e indexação, proporcionando uma compreensão da lógica do projeto.

Figura 1 - Código solução `main.cpp`.

```cpp
// Dupla:
// Matheus Fernandes Bigolin (22200371)
// Pedro Henrique De Sena Trombini Taglialenha (22203674)

#include "dicio.h"

#include <cstdlib>
#include <fstream>
#include <iostream>
#include <stdexcept>
#include <string>

std::ifstream abrir_arquivo(std::string nome_arq) {
  std::ifstream arq(nome_arq, std::ios_base::binary | std::ios_base::in);

  if (!arq.is_open()) {
    throw std::runtime_error("Não foi possível abrir " + nome_arq);
  }

  return arq;
}

int main(void) {
  std::string nome_arq;
  std::cin >> nome_arq;
```

```
  auto arq_dicio{ abrir_arquivo(nome_arq) };

  Dicionario dicio{};

  int tamanho_total{ 0 };
  std::string verbete;
  while (std::getline(arq_dicio, verbete)) {
    // Encontra o início e o fim da palavra sendo definida delimitada por
    // colchetes.
    const std::size_t inicio{ verbete.find('[') };
    const std::size_t fim{ verbete.find(']', inicio + 2) };

    if (inicio != std::string::npos && fim != std::string::npos) {
      // Extração da palavra que está sendo definida.
      std::string palavra{ verbete.substr(inicio + 1, fim - inicio - 1) };

      dicio.inserir(palavra, tamanho_total + inicio + 1, verbete.size());
    } else {
      throw std::runtime_error("Verbete mal-formado.");
    }

    tamanho_total += verbete.size() + 1;
  }

  while (true) {
    std::string palavra{};

    std::cin >> palavra;

    if (palavra == "0") {
      break;
    }

    if (dicio.existe_prefixo(palavra)) {
      std::size_t contagem{ dicio.contar_palavras_prefixadas(palavra) };

      std::cout << palavra << " is prefix of " << contagem << " words"
                << std::endl;

      Dicionario::Indice indice{ dicio.obter_indexacao(palavra) };

      // Verifica se o índice é valido antes de imprimi-lo.
      if (indice.posicao != 0 && indice.tamanho != 0) {
        std::cout << palavra << " is at (" << indice.posicao - 1 << ","
                  << indice.tamanho << ")" << std::endl;
      }
    } else {
      std::cout << palavra << " is not prefix." << std::endl;
    }
  }

  arq_dicio.close();
  return EXIT_SUCCESS;
}
```

Fonte: Desenvolvido pelos autores (2023).

2.1 DETALHAMENTO DE `MAIN.CPP`

O arquivo `main.cpp` desempenha um papel fundamental na execução do projeto, realizando a leitura do arquivo de dicionário, a construção da *trie* e a implementação das funcionalidades de verificação de prefixos e indexação.

2.1.1 Função `abrir_arquivo`

- Esta função recebe o nome de um arquivo como parâmetro e retorna um objeto do tipo `std::ifstream` que representa o arquivo aberto em modo binário para leitura.
- Caso o arquivo não possa ser aberto, uma exceção do tipo `std::runtime_error` é lançada.

2.1.2 Função `main`

- Inicia solicitando ao usuário o nome do arquivo de dicionário.
- Abre o arquivo utilizando a função `abrir_arquivo` e cria uma instância da classe `Dicionario`.
- Utiliza um laço para percorrer cada linha do arquivo de dicionário.
    - Identifica o início e o fim da palavra definida entre colchetes.
    - Insere a palavra na *trie* da classe `Dicionario` juntamente com a posição e tamanho da linha no arquivo.
    - Lança uma exceção se encontrar um verbete mal-formado.
    - Atualiza o tamanho total considerando o tamanho da linha atual.
- Em seguida, entra em outro laço para receber palavras do usuário até que a entrada seja "`0`".
    - Verifica se a palavra é um prefixo existente na *trie*.
    - Se for um prefixo, conta e exibe a quantidade de palavras que têm essa palavra como prefixo.
    - Obtém e exibe a posição e tamanho da palavra no dicionário, se aplicável.

    - Se não for um prefixo, exibe a mensagem indicando que a palavra não é um prefixo.
- Fecha o arquivo e encerra o programa com sucesso.

Figura 2 - Código solução `dicio.h`.

```
// Dupla:
// Matheus Fernandes Bigolin (22200371)
// Pedro Henrique De Sena Trombini Taglialenha (22203674)

#include <cstddef>
#include <string>

// Número de letras no alfabeto empregado.
constexpr auto TAMANHO_ALFABETO{ 'z' - 'a' + 1 };

class Dicionario {
private:
  // Um nó representa uma letra em uma posição específica.
  struct No {
    // Nós que sucedem o nó atual.
    No *filhos[TAMANHO_ALFABETO] = {};

    // Indica se o nó representa a última letra de uma palavra do dicionário.
    bool esta_terminal = false;

    // Posição e tamanho de um verbete no dicionário. Somente válidos se
    // esta_terminal é verdadeiro.
    std::size_t posicao = 0;
    std::size_t tamanho = 0;

    ~No() {
      for (auto filho : filhos) {
        delete filho;
      }
    }
  };

  // Início do dicionário; não representa nenhuma letra.
  No *raiz{ nullptr };

public:
  Dicionario() : raiz(new No()) {
  }

  ~Dicionario() {
    delete raiz;
  }

  void inserir(const std::string &palavra, std::size_t posicao,
               std::size_t tamanho) {
    No *atual{ raiz };

    for (auto letra : palavra) {
```

```
      if (atual->filhos[letra - 'a'] == nullptr) {
        atual->filhos[letra - 'a'] = new No();
      }

      atual = atual->filhos[letra - 'a'];
    }

    atual->esta_terminal = true;
    atual->posicao = posicao;
    atual->tamanho = tamanho;
  }

  bool existe_prefixo(const std::string &prefixo) const {
    return encontrar_prefixo(prefixo) != nullptr;
  }

  std::size_t contar_palavras_prefixadas(const std::string &prefixo) const {
    return contar_palavras(encontrar_prefixo(prefixo));
  }

  // Índice formado pela posição e tamanho de um verbete no dicionário.
  struct Indice {
    std::size_t posicao;
    std::size_t tamanho;
  };

  // Obtém a indexação de uma determinada palavra no dicionário.
  Indice obter_indexacao(const std::string &palavra) const {
    No *no{ encontrar_prefixo(palavra) };

    // Prefixo não existe ou não forma uma palavra.
    if (no == nullptr || !no->esta_terminal) {
      return Indice{ 0, 0 };
    }

    return Indice{ no->posicao, no->tamanho };
  }

private:
  // Encontra a subárvore de um determinado prefixo. Caso esse prefixo não
  // exista, retorna o ponteiro nulo.
  No *encontrar_prefixo(const std::string &prefixo) const {
    No *atual{ raiz };

    for (auto letra : prefixo) {
      if (atual->filhos[letra - 'a'] == nullptr) {
        return nullptr;
      }

      atual = atual->filhos[letra - 'a'];
    }

    return atual;
  }

  // Conta o número de todas as palavras na subárvore.
```

```
  std::size_t contar_palavras(No *no) const {
    if (no == nullptr) {
      return 0;
    }

    std::size_t contagem{ 0 };

    if (no->esta_terminal) {
      contagem++;
    }

    for (No *filho : no->filhos) {
      contagem += contar_palavras(filho);
    }

    return contagem;
  }
};
```

Fonte: Desenvolvido pelos autores (2023).

2.2 DETALHAMENTO DE `DICIO.H`

O arquivo `dicio.h` contém a definição da classe `Dicionario`, que representa a *trie* utilizada para resolver os problemas propostos no projeto. A classe foi projetada para fornecer uma implementação da *trie*, com métodos que permitem inserção, verificação de prefixos e obtenção de indexação.

2.2.1 Constante `TAMANHO_ALFABETO`

Esta constante define o tamanho do alfabeto utilizado no projeto, calculando a diferença entre os códigos ASCII de 'z' e 'a' e adicionando 1. Isso resulta no número total de letras no alfabeto utilizado na implementação da *trie*.

2.2.2 Estrutura `No`

A estrutura `No` é a base para cada nó na *trie*. Cada nó representa uma letra do alfabeto em uma posição específica e armazena informações sobre a palavra que o caminho até esse nó representa. Os membros incluem:

- `filhos[TAMANHO_ALFABETO]`: Um *array* de ponteiros para os nós filhos, representando as letras do alfabeto. Cada índice do *array* corresponde a uma letra, facilitando a navegação pela *trie*.

- `esta_terminal`: Um indicador de se o nó representa a última letra de uma palavra. Se verdadeiro, os membros `posicao` e `tamanho` são válidos.
- `posicao` e `tamanho`: Armazenam a posição inicial e o tamanho da palavra no dicionário.

2.2.3 Classe `Dicionario`

A classe `Dicionario` contém a implementação da *trie*. Seus membros incluem:

- `No *raiz`: Um ponteiro para o nó raiz da *trie*, que não representa nenhuma letra.
- `Dicionario()`: O construtor inicializa a *trie* com um nó raiz vazio.
- `~Dicionario()`: O destrutor libera a memória alocada para a *trie*, chamando o destrutor do nó raiz, que recursivamente libera todos os nós filhos.

2.2.4 Método `inserir`

Este método insere uma palavra na *trie*, atualizando a estrutura da *trie* conforme necessário. Percorre a *trie*, criando nós filhos se não existirem, até atingir o nó correspondente à última letra da palavra. Nesse ponto, marca o nó como terminal e armazena a posição e tamanho da palavra.

2.2.5 Método `existe_prefixo`

Este método verifica se um determinado prefixo existe na *trie*. Ele utiliza o método privado `encontrar_prefixo` para navegar até o nó correspondente ao último caractere do prefixo.

2.2.6 Método `contar_palavras_prefixadas`

Este método conta o número de palavras que têm um determinado prefixo. Ele utiliza o método privado `contar_palavras` para realizar uma contagem recursiva a partir do nó correspondente ao último caractere do prefixo.

2.2.7 Estrutura `Indice`

A estrutura `Indice` representa a posição e o tamanho de um verbete no dicionário. É utilizada para fornecer informações sobre a localização de uma palavra na indexação.

2.2.8 Método `obter_indexacao`

Este método obtém a indexação (posição e tamanho) de uma palavra no dicionário. Utiliza o método privado `encontrar_prefixo` para localizar o nó correspondente à última letra da palavra e retorna a estrutura `Indice` com as informações.

2.2.9 Método privado `encontrar_prefixo`

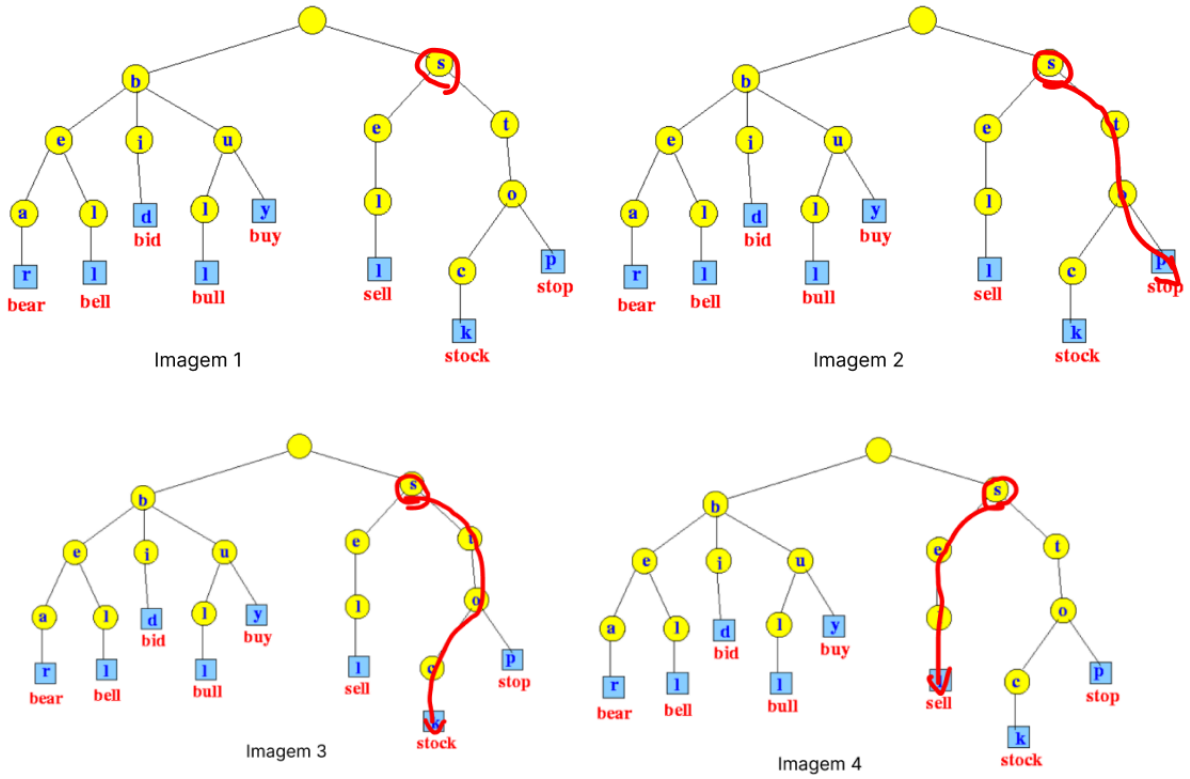
Este método encontra a subárvore correspondente a um determinado prefixo na *trie*. Percorre a *trie* até o nó correspondente ao último caractere do prefixo e retorna o ponteiro nulo se o prefixo não existir.

2.2.10 Método privado `contar_palavras`

Este método conta o número de palavras na subárvore a partir de um determinado nó. Realiza uma contagem recursiva, considerando todos os nós terminais na subárvore.

3 ILUSTRAÇÕES DA SOLUÇÃO

Figura 3 - Contagem do número de palavras prefixadas por “s”.

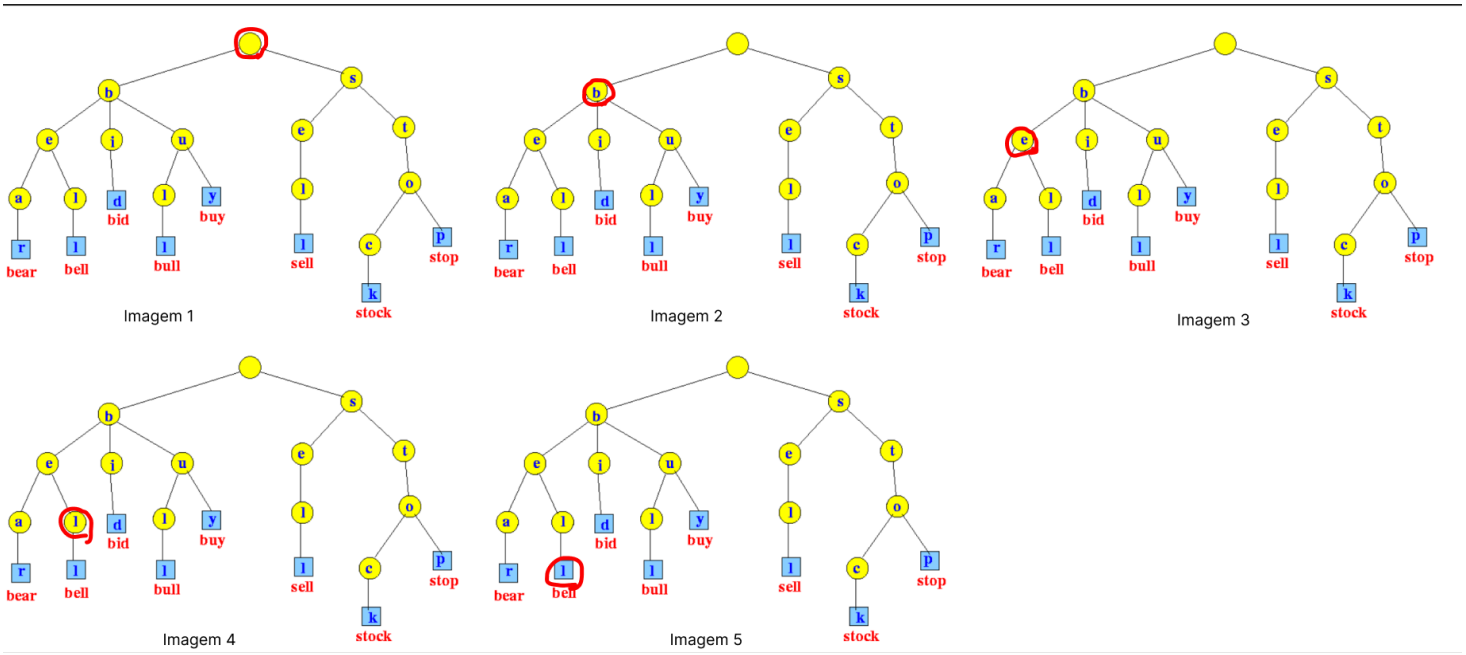


Fonte: Desenvolvido pelos autores (2023).

- Verificação do Primeiro Caractere na Raiz: Inicialmente, partimos da raiz da árvore, buscando a presença do primeiro caractere da string entre seus filhos. No exemplo em questão, o caractere 's' é identificado entre os filhos da raiz.
- Identificação da String Completa e Contagem de Palavras: Dado que 's' corresponde ao prefixo a ser buscado, procedemos à contagem de palavras que compartilham esse prefixo. Este é um passo crucial para determinar a extensão do prefixo na árvore *trie*.
- Exploração da Subárvore à Esquerda de 's': Ao percorrer a árvore à esquerda de 's', identificamos uma única subárvore. O último nó da subárvore, marcado em azul como o caractere 'k', representa o fim de uma palavra. Portanto, incrementamos a contagem de prefixos em 1.

- Exploração da Subárvore à Direita de 's': Exploramos a árvore à direita de 's' e encontramos o nó 'o' com 2 filhos. A busca continua nos filhos em busca de nós que representem o fim de palavras, marcados em azul. Após percorrer os filhos de 'o', identificamos um nó fim de palavra à sua esquerda e outro à sua direita. Incrementamos o contador em 2.
- Resultado Final: Ao concluir a análise, determinamos que 's' é um prefixo para 3 palavras no dicionário representado pela árvore *trie*.

Figura 4 - Funcionamento de busca para a palavra “bell”.



Fonte: Desenvolvido pelos autores (2023).

- Na Imagem 1, observamos o ponto inicial da busca pela palavra "bell" na árvore *trie*. A busca tem início no nó raiz, inicialmente vazio. A primeira letra da palavra é então procurada nas subárvores subsequentes.
- Na Imagem 2, apenas duas subárvores são verificadas, e a primeira letra da palavra desejada, "b", é identificada na subárvore correspondente.
- Na Imagem 3, nas raízes das subárvores abaixo de "b", verificamos qual delas possui a letra "e". No exemplo apresentado, encontramos a letra "e", e assim, inserimos esse radical.
- Na imagem 4, dois nós precisam ser verificados, e a letra “l” foi encontrada.

- Na Imagem 5 apenas um nó precisa ser verificado. Se não corresponder à próxima letra da palavra, indica que a palavra não está no dicionário. No entanto, neste caso, o nó corresponde à última letra da palavra “bell” e é retornado informações sobre sua indexação no dicionário, armazenados na estrutura `No`.

4 DESAFIOS ENCONTRADOS

O principal desafio encontrado para realizar o trabalho foi a implementação da *trie* com todas as operações necessárias. Para isso, a dupla precisou estudar sobre a estrutura de dados, tanto no aspecto teórico, isto é, quais são suas operações, quanto no aspecto prático de sua elaboração. Uma vez que a estrutura *trie* foi implementada em código, a resolução dos problemas propostos provou-se simples e de fácil execução.

5 CONCLUSÃO

Em conclusão, a implementação do projeto, que emprega uma estrutura hierárquica *trie* para indexação e recuperação eficiente de palavras em grandes arquivos de dicionários, demonstra uma solução robusta e organizada. A lógica desenvolvida atende aos requisitos propostos, proporcionando uma resposta eficiente na identificação de prefixos e na indexação das palavras, o que se revela crucial para a manipulação de dicionários extensos. O projeto não apenas cumpre os objetivos estabelecidos, mas também destaca a importância e eficácia da estrutura de dados *trie* em cenários nos quais a rapidez na recuperação de informações é essencial.

6 REFERÊNCIAS

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
https://www.youtube.com/watch?v=MEmLEYhna-I
http://desenvolvendosoftware.com.br/estruturas-de-dados/tries.html