Dictionary

1.0

Gerado por Doxygen 1.13.2

Capítulo 1

Contador de Frequências com Estruturas de Dados Avançadas

Aplicação em C++ para contagem de frequência de palavras em textos, utilizando e comparando o desempenho de quatro diferentes estruturas de dados: Árvore AVL, Árvore Rubro-Negra, Tabela Hash com Encadeamento e Tabela Hash com Endereçamento Aberto.

1.1 Sumário

1.2 Sobre o Projeto

Este repositório contém um projeto completo para a disciplina de Estruturas de Dados Avançadas (QXD0115) da Universidade Federal do Ceará. O objetivo é duplo:

1. **Implementar Estruturas de Dados:** Desenvolver implementações genéricas, robustas e eficientes de dicionários (mapas chave-valor) usando Árvore AVL, Árvore Rubro-Negra, Tabela Hash com Encadeamento e Tabela Hash com Endereçamento Aberto.
2. **Analisar Performance:** Utilizar essas estruturas em uma aplicação prática de contagem de frequência de palavras para coletar métricas (comparações, rotações, colisões) e realizar uma análise empírica do desempenho de cada uma em um cenário real.

O projeto é dividido em duas partes principais, conforme a especificação:

- **Parte 1:** Foco na implementação e teste das estruturas de dados.
- **Parte 2:** Desenvolvimento da aplicação final (contador de frequência) e análise comparativa.
- **Status: Aplicação Finalizada e Pronta para Análise**
- **Tecnologias:** C++20, STL, GoogleTest, Doxygen, Make
- **Objetivo Final:** Fornecer uma ferramenta funcional para análise de texto e, mais importante, um estudo comparativo sobre a performance de estruturas de dados clássicas.

1.3 Estruturas e Funcionalidades

O núcleo do projeto é uma interface de dicionário (`Dictionary<Key, Value>`) que abstrai a implementação subjacente, permitindo que a aplicação principal troque a estrutura de dados dinamicamente.

1.3.1 Interface `Dictionary`

A interface `Dictionary.hpp` define o seguinte contrato para todas as estruturas:

- `insert(const std::pair<Key, Value>&)`: Adiciona um par chave-valor.
- `remove(const Key&)`: Remove um par com base na chave.
- `update(const std::pair<Key, Value>&)`: Atualiza o valor de uma chave existente.
- `contains(const Key&)`: Verifica a existência de uma chave.
- `at(const Key&)`: Busca e retorna uma referência ao valor associado a uma chave.
- `operator[](const Key&)`: Permite acesso ou inserção de um valor (similar ao `std::map`).
- `clear()`: Remove todos os elementos.
- `size()`: Retorna o número de elementos.
- `empty()`: Verifica se o dicionário está vazio.
- `print()`: Imprime o conteúdo do dicionário.
- `forEach(const std::function<...>&)`: Executa uma função para cada par chave-valor.
- `clone()`: Cria uma cópia profunda (deep copy) do dicionário.

1.3.2 Componentes da Aplicação

Além das estruturas de dados, a aplicação conta com os seguintes componentes principais:

- **`DynamicDictionary`**: Uma classe *wrapper* que permite selecionar e usar qualquer uma das implementações de dicionário em tempo de execução.
- **`DictionaryFactory`**: Uma fábrica que simplifica a criação de instâncias de dicionários (`AVLTree`, `RedBlackTree`, etc.) com base em um `DictionaryType`.
- **`TextProcessor`**: Classe responsável por ler um arquivo de texto, normalizar as palavras (convertendo para minúsculas e removendo pontuações) e alimentar o dicionário.

1.3.3 Métricas Coletadas

Um requisito central do projeto é a análise de performance. Para isso, as seguintes métricas são rastreadas dentro de cada estrutura:

Estrutura	Métricas
Árvores (AVL e Rubro-Negra)	`comparações`, `rotações`
Tabelas Hash	`comparações`, `colisões`

Esses dados, juntamente com o tempo de execução, são salvos em arquivos de saída para permitir a análise comparativa.

1.4 Arquitetura e UML

A arquitetura foi projetada para ser modular e extensível. O diagrama abaixo ilustra a relação entre os principais componentes do sistema:

```
classDiagram
    direction LR

    class TextProcessor {
        -file_stream: ifstream
        -normalize(word: string): string
        +toLowerCase(text: string): void
        +processFile(wordHandler: function): void
    }

    class DictionaryFactory {
        «Factory»
        +create_dictionary(type): unique_ptr<Dictionary>
    }

    class Dictionary~Key, Value~ {
        «Interface»
        +insert(pair): void
        +update(pair): void
        +remove(key): void
        +at(key): Value
        +contains(key): bool
        +operator\[](key): Value
        +clear(): void
        +size(): size_t
        +empty(): bool
        +print(): void
        +forEach(func: function):void
        +clone(): unique_ptr<Dictionary>
    }

    class DynamicDictionary {
        -dictionary: unique_ptr<Dictionary>
        -type: DictionaryType
    }

    class AVLTree~Key, Value~ {
        -comparisons: long long
        -rotations: long long
    }
    class RedBlackTree~Key, Value~ {
        -comparisons: long long
        -rotations: long long
    }
```

```
class ChainedHashTable~Key, Value~ {
    -comparisons: long long
    -collisions: long long
}
class OpenHashTable~Key, Value~ {
    -comparisons: long long
    -collisions: long long
}

TextProcessor --> DynamicDictionary : "Usa para contar palavras"
DynamicDictionary o-- Dictionary
DictionaryFactory ..> AVLTree : "Cria"
DictionaryFactory ..> RedBlackTree : "Cria"
DictionaryFactory ..> ChainedHashTable : "Cria"
DictionaryFactory ..> OpenHashTable : "Cria"

Dictionary <|-- AVLTree
Dictionary <|-- RedBlackTree
Dictionary <|-- ChainedHashTable
Dictionary <|-- OpenHashTable
```

Este diagrama UML mostra a relação entre as classes principais do projeto, destacando a interface `Dictionary` e suas implementações concretas. A classe `TextProcessor` é responsável por processar o texto e alimentar o dicionário, enquanto a `DictionaryFactory` facilita a criação das diferentes estruturas de dados.

Diagrama geral do projeto:

1.5 Pré-requisitos

Para compilar e executar este projeto, você precisará de:

- **Compilador C++:** `g++` com suporte a C++20 ou superior.
- **Ferramentas de Build:** `make` e `git`.
- **Documentação:** `Doxygen` (opcional, para gerar a documentação da API).

A biblioteca `googletest` é utilizada para os testes e já está incluída como um submódulo no repositório.

1.6 Instalação e Compilação

Siga os passos abaixo para obter o código e compilá-lo.

1. **Clone o repositório:**

   `bash git clone https://github.com/WillianSilva51/Dictionary.git cd Dictionary`

2. **Inicialize o submódulo do GoogleTest:**

   `bash git submodule update --init --recursive`

3. **Compile o projeto usando o Makefile:** O `makefile` principal oferece vários alvos. Para compilar a aplicação principal e os testes, use `all`.

   `bash @section autotoc_md19 Compila o programa principal make`

   Para compilar em modo *release* (otimizado), use:

   `bash make MODE=release`

1.6.1 Executando o Programa

Após a compilação, você pode executar o contador de frequência a partir da raiz do projeto. O programa espera dois argumentos: o tipo de estrutura de dados e o nome do arquivo de texto (que deve estar no diretório `files/`).

Sintaxe:

```
./build/bin/Dictionary <estrutura> <arquivo.txt>
```

Argumentos:

- `<estrutura>`: O tipo de dicionário a ser usado. Opções:
  - `avl`: Árvore AVL

  - `rbt`: Árvore Rubro-Negra
  - `chash`: Tabela Hash com Encadeamento
  - `ohash`: Tabela Hash com Endereçamento Aberto
  - `all`: Executa e compara todas as quatro estruturas em threads separadas.
- `<arquivo.txt>`: O nome do arquivo de texto localizado na pasta `files/`.

Exemplos:

```
# Executar com a Árvore Rubro-Negra no arquivo bible.txt
./build/bin/Dictionary rbt bible.txt

# Executar e comparar todas as estruturas no arquivo donquijote.txt
./build/bin/Dictionary all donquijote.txt
```

Os resultados, incluindo a contagem de palavras e as métricas de desempenho, serão salvos em um novo arquivo dentro do diretório `out/`. O programa também exibirá um resumo das métricas no console.
Para ver a mensagem de ajuda, execute:

```
./build/bin/Dictionary help
```

1.6.2 Executando os Testes

A validação das estruturas de dados é realizada através de um conjunto de testes unitários com GoogleTest. Para executá-los, use o seguinte comando:

```
make test
```

A saída mostrará os resultados de todos os casos de teste para cada estrutura de dados, garantindo que as operações básicas e os casos de borda estão funcionando como esperado.

1.6.3 Documentação da API

A documentação completa de todas as classes, métodos e da arquitetura do projeto foi gerada com o **Doxygen**. Para consultá-la:

1. **Gere a documentação (requer Doxygen instalado):**

   ```
   bash make docs
   ```

2. **Abra o arquivo principal em seu navegador:** `docs/html/index.html`

A documentação é a melhor fonte de referência para entender os detalhes de implementação de cada método.

1.6.4 Roadmap do Projeto

- ☑ **Parte 1:** Implementação das Estruturas de Dados (AVL, RB, Hash com Encadeamento, Hash com Endereçamento Aberto).
- ☑ **Parte 1:** Inclusão de contadores de métricas de performance (comparações, rotações, colisões).
- ☑ **Parte 1:** Desenvolvimento de testes unitários com GoogleTest para validar as estruturas.
- ☑ **Parte 1:** Criação da documentação da API com Doxygen.
- ☑ **Parte 2:** Implementação da aplicação de contador de frequência (leitura de arquivos, processamento de texto).
- ☑ **Parte 2:** Coleta de dados e análise comparativa de performance entre as estruturas.
- ☐ **Parte 2:** Finalização do relatório e apresentação do projeto.

1.6.5 Contribuição

Contribuições são bem-vindas! Se você tiver sugestões para melhorar o projeto, siga estes passos:

1. Faça um *Fork* deste repositório.
2. Crie uma nova *Branch*: `git checkout -b feature/sua-feature`.
3. Faça o *Commit* de suas mudanças: 'git commit -m 'feat: Descrição da sua feature' .
4. `Faça o *Push* para a *Branch*:` git push origin feature/sua-feature`.
5. Abra um *Pull Request*.

Como alternativa, consulte a documentação do GitHub em `como criar uma solicitação pull`.

1.6.6 Licença

Este projeto está licenciado sob a Licença MIT. Veja o arquivo `LICENSE` para mais detalhes.

1.6.7 Créditos

- **Professor:** Prof. Atílio Gomes Luiz – Universidade Federal do Ceará.
- **Material de Apoio:** Slides e materiais da disciplina de Estruturas de Dados Avançadas.
- **Ferramentas:** `GoogleTest` para os testes unitários e `Doxygen` para a documentação.

Capítulo 2

Índice da hierarquia

2.1 Hierarquia de classes

Esta lista de heranças está organizada, dentro do possível, por ordem alfabética:

Capítulo 3

Índice dos componentes

3.1 Lista de componentes

Lista de classes, estruturas, uniões e interfaces com uma breve descrição:

Capítulo 4

Índice dos ficheiros

4.1 Lista de ficheiros

Lista de todos os ficheiros com uma breve descrição:

Capítulo 5

Documentação da classe

5.1 Referência à classe Template AVLTree< Key, Value >

`#include <AVLTree.hpp>`
Diagrama de heranças da classe AVLTree< Key, Value >

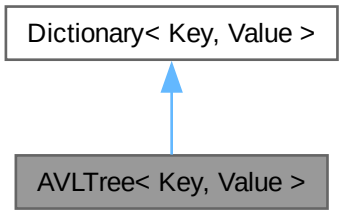
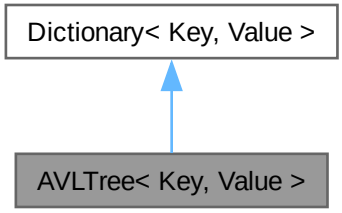


Diagrama de colaboração para AVLTree< Key, Value >:



Membros públicos

- AVLTree ()=default

  Construtor padrão. Cria um conjunto vazio.
- AVLTree (const AVLTree &other)

  Construtor de cópia. Cria um novo conjunto como cópia de `other`.

- AVLTree (std::initializer_list< std::pair< Key, Value > > list)

  Construtor a partir de uma lista inicializadora.
- std::unique_ptr< Dictionary< Key, Value > > clone () const

  Cria uma cópia profunda da árvore AVL.
- ~AVLTree ()

  Destrutor. Libera toda a memória alocada pelos nós da árvore.
- iterator begin () noexcept

  Retorna um iterador para o início do conjunto.
- iterator end () noexcept

  Retorna um iterador para o final do conjunto.
- iterator begin () const noexcept

  Retorna um iterador constante para o início do conjunto.
- iterator end () const noexcept

  Retorna um iterador constante para o início do conjunto.
- iterator cbegin () const noexcept

  Retorna um iterador constante para o início do conjunto.
- iterator cend () const noexcept

  Retorna um iterador constante para o final do conjunto.
- void operator= (const AVLTree &other)

  Operador de atribuição por cópia.
- size_t size () const noexcept

  Retorna o número de elementos no conjunto.
- bool empty () const noexcept

  Verifica se o conjunto está vazio.
- long long getComparisons () const noexcept

  Retorna o número de comparações realizadas durante as operações.
- long long getRotations () const noexcept

  Retorna o número de rotações realizadas durante as operações.
- void clear ()

  Remove todos os elementos do conjunto.
- void swap (AVLTree< Key, Value > &other) noexcept

  Troca o conteúdo deste conjunto com o de `other`.
- void insert (const std::pair< Key, Value > &key)

  Insere uma chave no conjunto.
- Value & at (const Key &key)

  Retorna o nó associado a uma chave.
- Value & operator[ ] (const Key &key)

  Sobrecarga do operador de indexação para acessar o valor associado a uma chave.
- void update (const std::pair< Key, Value > &key)

  Atualiza o valor associado a uma chave existente ou insere uma nova chave.
- void operator= (std::pair< Key, Value > &key)

  Sobrecarga do operador de atribuição para atualizar ou inserir uma chave.
- void remove (const Key &key)

  Remove uma chave do conjunto.
- bool contains (const Key &key)

  Verifica se o conjunto contém uma determinada chave.
- void print () const

  Imprime os elementos do conjunto em ordem crescente (in-order traversal).
- void forEach (const std::function< void(const std::pair< Key, Value > &)> &func) const

  Aplica uma função a cada par chave-valor no conjunto.
- void bshow ()

  Exibe a estrutura da árvore AVL de forma visual no console.

Membros públicos herdados de Dictionary< Key, Value >

- virtual ~Dictionary ()=default

  Destrutor virtual para permitir a destruição correta de classes derivadas.

Amigos

- class IteratorAVL< Key, Value >

  Declaração da classe IteratorAVL como amiga.

5.1.1 Descrição detalhada

template<typename Key, typename Value>
class AVLTree< Key, Value >

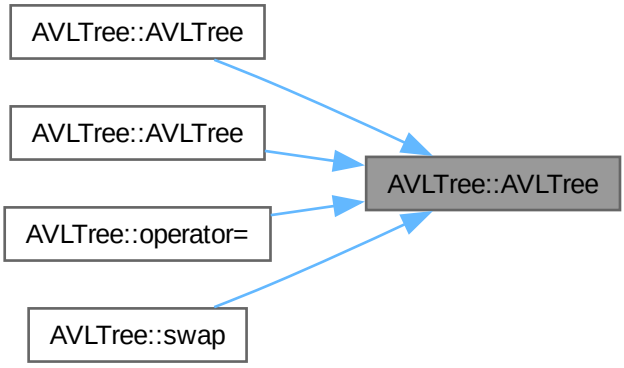
Definido na linha 25 do ficheiro AVLTree.hpp.

5.1.2 Documentação dos Construtores & Destrutor

5.1.2.1 AVLTree() [1/3]

```
template<typename Key, typename Value>
AVLTree< Key, Value >::AVLTree ()  [default]
```

Construtor padrão. Cria um conjunto vazio.
Este é o diagrama das funções que utilizam esta função:



5.1.2.2 AVLTree() [2/3]

```
template<typename Key, typename Value>
AVLTree< Key, Value >::AVLTree (
            const AVLTree< Key, Value > & other)
```

Construtor de cópia. Cria um novo conjunto como cópia de `other`.
Realiza uma cópia profunda dos elementos.

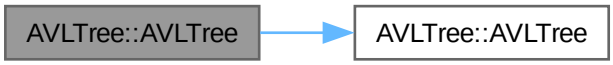
Parâmetros

other	O conjunto a ser copiado.

Definido na linha 516 do ficheiro AVLTree.hpp.

```
00516                                                 : AVLTree()
00517 {
00518     if (other.root != nullptr)
00519     {
00520         root = clone_recursive(other.root);
00521         size_m = other.size_m;
00522         comparisons = other.comparisons;
00523         rotations = other.rotations;
00524     }
00525 }
```

Grafo de chamadas desta função:



5.1.2.3 AVLTree() [3/3]

```
template<typename Key, typename Value>
AVLTree< Key, Value >::AVLTree (
            std::initializer_list< std::pair< Key, Value > > list)
```

Construtor a partir de uma lista inicializadora.
Cria um conjunto e insere todos os elementos da lista.

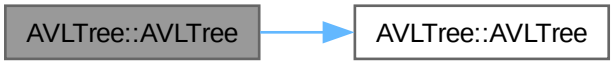
Parâmetros

list	A lista de inicialização (`std::initializer_list<std::pair<Key, Value>>`).

Definido na linha 509 do ficheiro AVLTree.hpp.

```
00509                                                                 : AVLTree()
00510 {
00511     for (const auto &key : list)
00512         insert(key);
00513 }
```

Grafo de chamadas desta função:



5.1.2.4 ~AVLTree()

```
template<typename Key, typename Value>
AVLTree< Key, Value >::~AVLTree ()
```

Destrutor. Liberta toda a memória alocada pelos nós da árvore.

Definido na linha 534 do ficheiro AVLTree.hpp.

```
00535 {
00536     clear();
00537 }
```

5.1.3 Documentação das funções

5.1.3.1 at()

```
template<typename Key, typename Value>
Value & AVLTree< Key, Value >::at (
            const Key & key)  [inline], [virtual]
```

Retorna o nó associado a uma chave.
Se a chave não existir, lança uma exceção.

Parâmetros

key	A chave a ser buscada.

Retorna

NodePtr Ponteiro para o nó associado à chave.

Implementa Dictionary< Key, Value >.
Definido na linha 428 do ficheiro AVLTree.hpp.

```
00428 { return at(root, key); };
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.1.3.2 begin() [1/2]

```
template<typename Key, typename Value>
iterator AVLTree< Key, Value >::begin () const  [inline], [noexcept]
```

Retorna um iterador constante para o início do conjunto.
O iterador aponta para o menor elemento da árvore (in-order traversal).

Retorna

IteratorAVL<Key, Value> Um iterador constante para o início do conjunto.

Definido na linha 322 do ficheiro AVLTree.hpp.

```
00322 { return iterator(root); }
```

5.1.3.3 begin() [2/2]

```
template<typename Key, typename Value>
iterator AVLTree< Key, Value >::begin ()  [inline], [noexcept]
```

Retorna um iterador para o início do conjunto.
O iterador aponta para o menor elemento da árvore (in-order traversal).

Retorna

IteratorAVL<Key, Value> Um iterador para o início do conjunto.

Definido na linha 304 do ficheiro AVLTree.hpp.

```
00304 { return iterator(root); }
```

5.1.3.4 bshow()

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::bshow ()
```

Exibe a estrutura da árvore AVL de forma visual no console.
Útil para depuração e visualização do balanceamento da árvore.
Definido na linha 931 do ficheiro AVLTree.hpp.

```
00932 {
00933     bshow(root, "");
00934 }
```

5.1.3.5 cbegin()

```
template<typename Key, typename Value>
iterator AVLTree< Key, Value >::cbegin () const  [inline], [noexcept]
```

Retorna um iterador constante para o início do conjunto.
O iterador aponta para o menor elemento da árvore (in-order traversal).

Retorna

IteratorAVL<Key, Value> Um iterador constante para o início do conjunto.

Definido na linha 340 do ficheiro AVLTree.hpp.

```
00340 { return iterator(root); }
```

5.1.3.6 cend()

```
template<typename Key, typename Value>
iterator AVLTree< Key, Value >::cend () const  [inline], [noexcept]
```

Retorna um iterador constante para o final do conjunto.
O iterador aponta para o elemento após o maior elemento da árvore.

Retorna

IteratorAVL<Key, Value> Um iterador constante para o final do conjunto.

Definido na linha 349 do ficheiro AVLTree.hpp.

```
00349 { return iterator(); }
```

5.1.3.7 clear()

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::clear ()  [virtual]
```

Remove todos os elementos do conjunto.
Após esta operação, `size()` retornará 0 e `empty()` retornará `true`.
Implementa Dictionary< Key, Value >.
Definido na linha 583 do ficheiro AVLTree.hpp.

```
00584 {
00585     root = clear(root);
00586     size_m = 0;
00587 }
```

5.1.3.8 clone()

```
template<typename Key, typename Value>
std::unique_ptr< Dictionary< Key, Value > > AVLTree< Key, Value >::clone () const  [virtual]
```

Cria uma cópia profunda da árvore AVL.
Retorna um ponteiro inteligente para uma nova instância da árvore AVL que contém os mesmos elementos que a árvore atual.

Retorna

std::unique_ptr<AVLTree<Key, Value>> Um ponteiro inteligente para a nova árvore AVL.

Implementa Dictionary< Key, Value >.
Definido na linha 528 do ficheiro AVLTree.hpp.

```
00529 {
00530     return std::make_unique<AVLTree<Key, Value>>(*this);
00531 }
```

5.1.3.9 contains()

```
template<typename Key, typename Value>
bool AVLTree< Key, Value >::contains (
            const Key & key)  [virtual]
```

Verifica se o conjunto contém uma determinada chave.

Parâmetros

key	A chave a ser procurada.

Retorna

true Se a chave estiver presente no conjunto.

false Caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 898 do ficheiro AVLTree.hpp.

```
00899 {
00900     return contains(root, key);
00901 }
```

5.1.3.10 empty()

```
template<typename Key, typename Value>
bool AVLTree< Key, Value >::empty () const  [virtual], [noexcept]
```

Verifica se o conjunto está vazio.

Retorna

true Se o conjunto não contiver elementos.

false Caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 562 do ficheiro AVLTree.hpp.

```
00563 {
00564     return root == nullptr;
00565 }
```

5.1.3.11 end() [1/2]

```
template<typename Key, typename Value>
iterator AVLTree< Key, Value >::end () const  [inline], [noexcept]
```

Retorna um iterador constante para o início do conjunto.
O iterador aponta para o menor elemento da árvore (in-order traversal).

Retorna

IteratorAVL<Key, Value> Um iterador constante para o início do conjunto.

Definido na linha 331 do ficheiro AVLTree.hpp.

```
00331 { return iterator(); }
```

5.1.3.12 end() [2/2]

```
template<typename Key, typename Value>
iterator AVLTree< Key, Value >::end ()  [inline], [noexcept]
```

Retorna um iterador para o final do conjunto.
O iterador aponta para o elemento após o maior elemento da árvore.

Retorna

IteratorAVL<Key, Value> Um iterador para o final do conjunto.

Definido na linha 313 do ficheiro AVLTree.hpp.

```
00313 { return iterator(); }
```

5.1.3.13 forEach()

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::forEach (
            const std::function< void(const std::pair< Key, Value > &)> & func) const  [virtual]
```

Aplica uma função a cada par chave-valor no conjunto.
Permite iterar sobre todos os elementos do conjunto e aplicar uma função personalizada a cada um deles.

Parâmetros

func	A função a ser aplicada a cada par chave-valor.

Implementa Dictionary< Key, Value >.
Definido na linha 924 do ficheiro AVLTree.hpp.

```
00925 {
00926     for (const auto &key : *this)
00927         func(key);
00928 }
```

5.1.3.14 getComparisons()

```
template<typename Key, typename Value>
long long AVLTree< Key, Value >::getComparisons () const  [inline], [noexcept]
```

Retorna o número de comparações realizadas durante as operações.
Útil para medir a eficiência das operações na árvore.

Retorna

long long O número de comparações realizadas.

Definido na linha 383 do ficheiro AVLTree.hpp.

```
00383 { return comparisons; }
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.1.3.15 getRotations()

```
template<typename Key, typename Value>
long long AVLTree< Key, Value >::getRotations () const  [inline], [noexcept]
```

Retorna o número de rotações realizadas durante as operações.
Útil para medir a quantidade de balanceamento necessário na árvore.

Retorna

long long O número de rotações realizadas.

Definido na linha 392 do ficheiro AVLTree.hpp.

```
00392 { return rotations; }
```

5.1.3.16 insert()

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::insert (
            const std::pair< Key, Value > & key)  [virtual]
```

Insere uma chave no conjunto.
Se a chave já existir, o conjunto não é modificado. A árvore é balanceada após a inserção, se necessário.

Parâmetros

key	A chave a ser inserida.

Implementa Dictionary< Key, Value >.
Definido na linha 652 do ficheiro AVLTree.hpp.

```
00653 {
00654     root = insert(root, key);
00655 }
```

5.1.3.17 operator=() [1/2]

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::operator= (
            const AVLTree< Key, Value > & other)
```

Operador de atribuição por cópia.
Substitui o conteúdo do conjunto atual pelo conteúdo de `other`. Garante a autotribuição segura e libera a memória antiga antes de copiar.

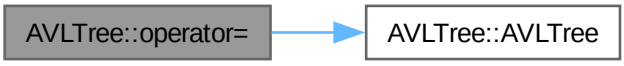
Parâmetros

other	O conjunto a ser copiado.

Definido na linha 540 do ficheiro AVLTree.hpp.

```
00541 {
00542     if (this != &other)
00543     {
00544         clear(); // Limpa a árvore atual
00545         if (other.root != nullptr)
00546         {
00547             root = clone_recursive(other.root);
00548             size_m = other.size_m;
00549             comparisons = other.comparisons;
00550             rotations = other.rotations;
00551         }
00552     }
00553 }
```

Grafo de chamadas desta função:



5.1.3.18 operator=() [2/2]

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::operator= (
            std::pair< Key, Value > & key)  [inline]
```

Sobrecarga do operador de atribuição para atualizar ou inserir uma chave.
Permite usar a sintaxe `tree = {key, value}` para atualizar ou inserir uma chave.

Parâmetros

key	O par chave-valor a ser atualizado ou inserido.

Definido na linha 460 do ficheiro AVLTree.hpp.

```
00460 { root = update(root, key); };
```

5.1.3.19 operator[]()

```
template<typename Key, typename Value>
Value & AVLTree< Key, Value >::operator[] (
            const Key & key)  [virtual]
```

Sobrecarga do operador de indexação para acessar o valor associado a uma chave.
Permite usar a sintaxe `tree[key]` para acessar o valor associado à chave. Se a chave não existir, cria uma nova entrada com valor padrão.

Parâmetros

key	A chave a ser buscada.

Retorna

Value O valor associado à chave.

Implementa Dictionary< Key, Value >.
Definido na linha 762 do ficheiro AVLTree.hpp.

```
00763 {
00764     NodePtr aux = root;
00765 
00766     while (aux != nullptr)
00767     {
00768         comparisons++;
00769         if (key == aux->key.first)
00770             return aux->key.second;
00771 
00772         comparisons++;
00773         if (key < aux->key.first)
00774             aux = aux->left;
00775         else
00776             aux = aux->right;
00777     }
00778     // Se a chave não for encontrada, insere um novo nó com valor padrão
00779     root = insert(root, {key, Value()});
00780 
00781     return at(root, key); // Retorna o valor associado à nova chave
00782 }
```

5.1.3.20 print()

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::print () const  [virtual]
```

Imprime os elementos do conjunto em ordem crescente (in-order traversal).
Implementa Dictionary< Key, Value >.
Definido na linha 904 do ficheiro AVLTree.hpp.

```
00905 {
00906     printInOrder(root);
00907 }
```

5.1.3.21 remove()

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::remove (
            const Key & key)  [virtual]
```

Remove uma chave do conjunto.
Se a chave não existir, o conjunto não é modificado. A árvore é balanceada após a remoção, se necessário.

Parâmetros

key	A chave a ser removida.

Implementa Dictionary< Key, Value >.
Definido na linha 658 do ficheiro AVLTree.hpp.

```
00659 {
00660     root = m_remove(root, key);
00661 }
```

5.1.3.22 size()

```
template<typename Key, typename Value>
size_t AVLTree< Key, Value >::size () const  [virtual], [noexcept]
```

Retorna o número de elementos no conjunto.

Retorna

size_t O número de elementos.

Implementa Dictionary< Key, Value >.
Definido na linha 556 do ficheiro AVLTree.hpp.

```
00557 {
00558     return size_m;
00559 }
```

5.1.3.23 swap()

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::swap (
            AVLTree< Key, Value > & other)  [noexcept]
```

Troca o conteúdo deste conjunto com o de `other`.
Operação eficiente que apenas troca os ponteiros raiz e os tamanhos.

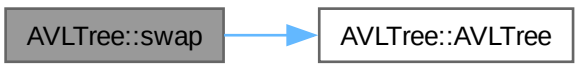
Parâmetros

other	O outro conjunto com o qual trocar o conteúdo.

Definido na linha 590 do ficheiro AVLTree.hpp.

```
00591 {
00592     std::swap(root, other.root);
00593     std::swap(size_m, other.size_m);
00594     std::swap(comparisons, other.comparisons);
00595     std::swap(rotations, other.rotations);
00596 }
```

Grafo de chamadas desta função:



5.1.3.24 update()

```
template<typename Key, typename Value>
void AVLTree< Key, Value >::update (
            const std::pair< Key, Value > & key)  [inline], [virtual]
```

Atualiza o valor associado a uma chave existente ou insere uma nova chave.
Se a chave já existir, atualiza seu valor. Caso contrário, insere a nova chave. A árvore é balanceada após a atualização, se necessário.

Parâmetros

key	O par chave-valor a ser atualizado ou inserido.

Excepções

std::out_of_range	Se a chave não existir.

Implementa Dictionary< Key, Value >.
Definido na linha 451 do ficheiro AVLTree.hpp.

```
00451 { root = update(root, key); };
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.1.4 Documentação dos símbolos amigos e relacionados

5.1.4.1 IteratorAVL< Key, Value >

```
template<typename Key, typename Value>
friend class IteratorAVL< Key, Value >  [friend]
```

Declaração da classe IteratorAVL como amiga.
Permite que a classe IteratorAVL acesse membros privados e protegidos da classe AVLTree, facilitando a implementação de iteração sobre a árvore.
Definido na linha 937 do ficheiro AVLTree.hpp.
A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/avl_tree/AVLTree.hpp

5.2 Referência à classe AVLTreeSpecificTest

Diagrama de heranças da classe AVLTreeSpecificTest

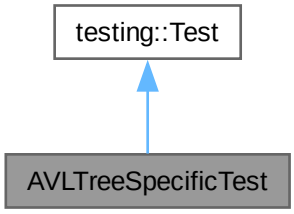
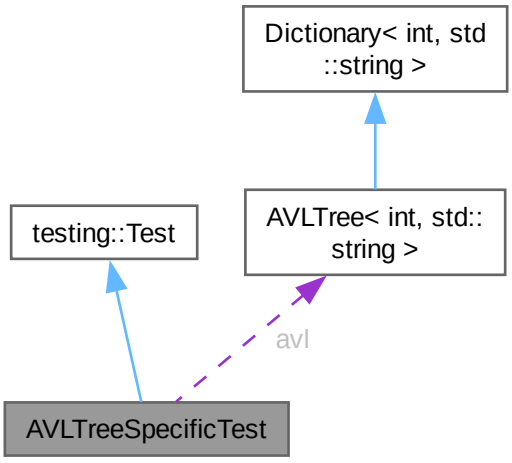


Diagrama de colaboração para AVLTreeSpecificTest:



Atributos Protegidos

- AVLTree< int, std::string > avl

5.2.1 Descrição detalhada

Definido na linha 223 do ficheiro Tests.cpp.

5.2.2 Documentação dos dados membro

5.2.2.1 avl

```
AVLTree<int, std::string> AVLTreeSpecificTest::avl  [protected]
```

Definido na linha 226 do ficheiro Tests.cpp.

A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- tests/Tests.cpp

5.3 Referência à classe Template ChainedHashTable< Key, Value, Hash >

`#include <ChainedHashTable.hpp>`

Diagrama de heranças da classe ChainedHashTable< Key, Value, Hash >

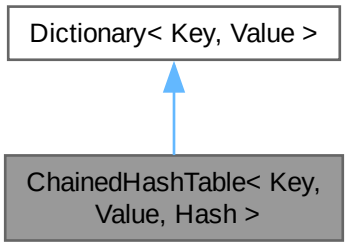
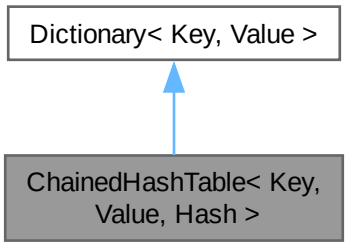


Diagrama de colaboração para ChainedHashTable< Key, Value, Hash >:



Membros públicos

- ChainedHashTable (const size_t &tableSize=19, const float &load_factor=1.0f)

  Construtor padrão. Cria uma tabela hash vazia.
- ChainedHashTable (const std::initializer_list< std::pair< Key, Value > > &list, const size_t &tableSize=19, const float &load_factor=1.0f)

  Construtor que inicializa a tabela com uma lista de elementos.
- std::unique_ptr< Dictionary< Key, Value > > clone () const

  Cria e retorna uma cópia profunda (deep copy) da tabela hash.
- long long getComparisons () const noexcept

  Retorna o número de comparações realizadas durante as operações.
- long long getCollisions () const noexcept

  Retorna o número de colisões ocorridas durante as operações.
- size_t size () const noexcept

Retorna o número de pares chave-valor na tabela.

- bool empty () const noexcept

  Verifica se a tabela está vazia.
- size_t bucket_count () const noexcept

  Retorna o número de "buckets" (slots) na tabela hash.
- size_t bucket_size (size_t n) const

  Retorna o número de elementos em um "bucket" específico.
- size_t bucket (const Key &k) const

  Retorna o índice do "bucket" onde um elemento com a chave `k` seria armazenado.
- void clear ()

  Remove todos os elementos da tabela, deixando-a com tamanho 0.
- float load_factor () const noexcept

  Retorna o fator de carga atual da tabela. O fator de carga é a razão entre o número de elementos e o número de "buckets".
- float max_load_factor () const noexcept

  Retorna o fator de carga máximo permitido. Se `load_factor()` exceder este valor, um rehash é acionado.
- ~ChainedHashTable ()=default

  Destrutor. Libera todos os recursos.
- void insert (const std::pair< Key, Value > &key_value)

  Insere um novo par chave-valor na tabela.
- void update (const std::pair< Key, Value > &key_value)

  Atualiza o valor associado a uma chave existente.
- bool contains (const Key &k)

  Verifica se a tabela contém um elemento com a chave especificada.
- Value & at (const Key &k)

  Acessa o valor associado a uma chave.
- const Value & at (const Key &k) const

  Acessa o valor associado a uma chave (versão const).
- void rehash (size_t m)

  Solicita uma alteração no número de "buckets" da tabela.
- void remove (const Key &k)

  Remove um elemento da tabela pela chave.
- void reserve (size_t n) noexcept

  Reserva espaço para pelo menos `n` elementos.
- void set_max_load_factor (float lf)

  Define o fator de carga máximo.
- Value & operator[ ] (const Key &k)

  Acessa ou insere um elemento.
- const Value & operator[ ] (const Key &k) const

  Acessa um elemento (versão const).
- void print () const

  Imprime o conteúdo da tabela no formato [chave1:valor1, chave2:valor2, ...].
- void forEach (const std::function< void(const std::pair< Key, Value > &)> &func) const

  Aplica uma função a cada par chave-valor na tabela.

Membros públicos herdados de Dictionary< Key, Value >

- virtual ~Dictionary ()=default

  Destrutor virtual para permitir a destruição correta de classes derivadas.

5.3.1 Descrição detalhada

template<typename Key, typename Value, typename Hash = std::hash<Key>>
class ChainedHashTable< Key, Value, Hash >

Definido na linha 29 do ficheiro ChainedHashTable.hpp.

5.3.2 Documentação dos Construtores & Destrutor

5.3.2.1 ChainedHashTable() [1/2]

```
template<typename Key, typename Value, typename Hash>
ChainedHashTable< Key, Value, Hash >::ChainedHashTable (
            const size_t & tableSize = 19,
            const float & load_factor = 1.0f)
```

Construtor padrão. Cria uma tabela hash vazia.

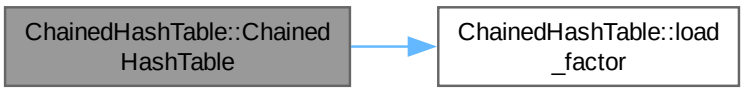
Parâmetros

tableSize	O número inicial de "buckets" (slots) na tabela. Será ajustado para o próximo número primo maior ou igual.
load_factor	O fator de carga máximo permitido antes de um rehash.

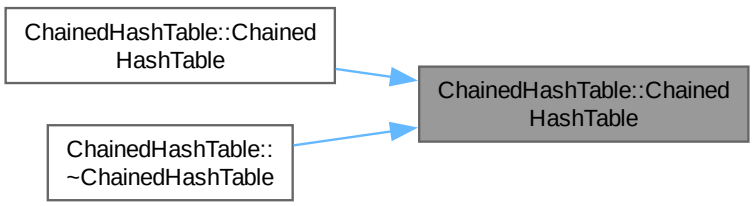
Definido na linha 358 do ficheiro ChainedHashTable.hpp.

```
00358
      : m_number_of_elements(0), m_table_size(tableSize)
00359 {
00360     m_table.resize(m_table_size);
00361
00362     if (load_factor <= 0)
00363         m_max_load_factor = 1.0f;
00364     else
00365         m_max_load_factor = load_factor;
00366 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



5.3.2.2 ChainedHashTable() [2/2]

```
template<typename Key, typename Value, typename Hash>
ChainedHashTable< Key, Value, Hash >::ChainedHashTable (
            const std::initializer_list< std::pair< Key, Value > > & list,
            const size_t & tableSize = 19,
            const float & load_factor = 1.0f)
```

Construtor que inicializa a tabela com uma lista de elementos.

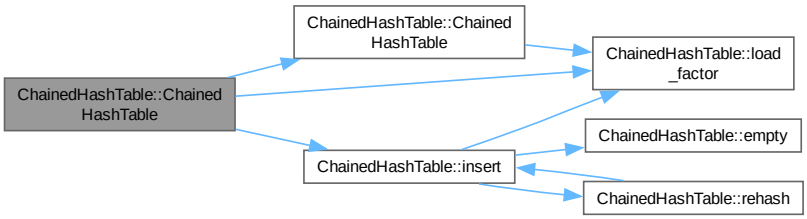
Parâmetros

list	Uma `std::initializer_list` de pares chave-valor para inserir na tabela.
tableSize	O número inicial de "buckets" na tabela.
load_factor	O fator de carga máximo.

Definido na linha 369 do ficheiro ChainedHashTable.hpp.

```
00369
      : ChainedHashTable(tableSize, load_factor)
00370 {
00371     for (const auto &pair : list)
00372         insert(pair);
00373 }
```

Grafo de chamadas desta função:

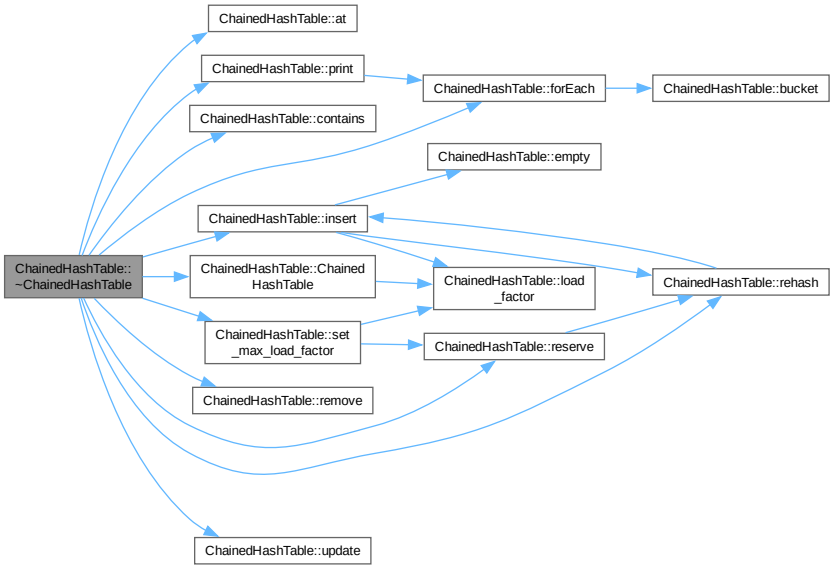


5.3.2.3 ~ChainedHashTable()

```
template<typename Key, typename Value, typename Hash = std::hash<Key>>
ChainedHashTable< Key, Value, Hash >::~ChainedHashTable ()  [default]
```

Destrutor. Libera todos os recursos.

Grafo de chamadas desta função:



5.3.3 Documentação das funções

5.3.3.1 at() [1/2]

```
template<typename Key, typename Value, typename Hash>
Value & ChainedHashTable< Key, Value, Hash >::at (
            const Key & k)  [virtual]
```

Acessa o valor associado a uma chave.
Retorna uma referência ao valor correspondente à chave `k`.

Parâmetros

k	A chave do elemento a ser acessado.

Retorna

Value& Uma referência ao valor.

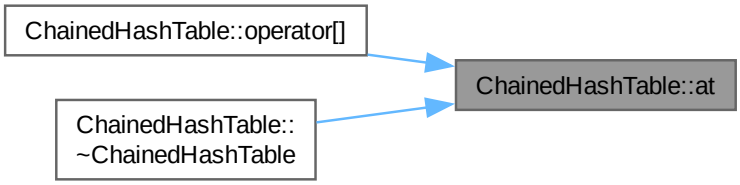
Excepções

std::out_of_range	se a chave `k` não for encontrada na tabela.

Implementa Dictionary< Key, Value >.
Definido na linha 488 do ficheiro ChainedHashTable.hpp.

```
00489 {
00490     size_t hash_index = hash_code(k);
00491 
00492     for (auto &pair : m_table[hash_index])
00493     {
00494         comparisons++;
00495         if (pair.first == k)
00496             return pair.second; // retorna o valor associado a chave
00497     }
00498 
00499     throw std::out_of_range("Key not found in the hash table");
00500 }
```

Este é o diagrama das funções que utilizam esta função:



5.3.3.2 at() [2/2]

```
template<typename Key, typename Value, typename Hash>
const Value & ChainedHashTable< Key, Value, Hash >::at (
            const Key & k) const
```

Acessa o valor associado a uma chave (versão const).
Retorna uma referência constante ao valor correspondente à chave `k`.

Parâmetros

k	A chave do elemento a ser acessado.

Retorna

const Value& Uma referência constante ao valor.

Excepções

std::out_of_range	se a chave `k` não for encontrada na tabela.

Definido na linha 503 do ficheiro ChainedHashTable.hpp.

```
00504 {
00505     size_t hash_index = hash_code(k);
00506
00507     for (const auto &pair : m_table[hash_index])
00508     {
00509         comparisons++;
00510         if (pair.first == k)
00511             return pair.second; // retorna o valor associado a chave
00512     }
00513
00514     throw std::out_of_range("Key not found in the hash table");
00515 }
```

5.3.3.3 bucket()

```
template<typename Key, typename Value, typename Hash>
size_t ChainedHashTable< Key, Value, Hash >::bucket (
            const Key & k) const
```

Retorna o índice do "bucket" onde um elemento com a chave `k` seria armazenado.

Parâmetros

k	A chave a ser localizada.

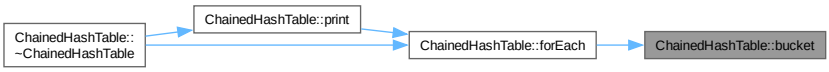
Retorna

size_t O índice do "bucket" correspondente.

Definido na linha 409 do ficheiro ChainedHashTable.hpp.

```
00410 {
00411     return hash_code(k);
00412 }
```

Este é o diagrama das funções que utilizam esta função:



5.3.3.4 bucket_count()

```
template<typename Key, typename Value, typename Hash>
size_t ChainedHashTable< Key, Value, Hash >::bucket_count () const  [noexcept]
```

Retorna o número de "buckets" (slots) na tabela hash.

Retorna

size_t O tamanho da tabela interna (número de listas de encadeamento).

Definido na linha 394 do ficheiro ChainedHashTable.hpp.

```
00395 {
00396     return m_table_size;
00397 }
```

5.3.3.5 bucket_size()

```
template<typename Key, typename Value, typename Hash>
size_t ChainedHashTable< Key, Value, Hash >::bucket_size (
            size_t n) const
```

Retorna o número de elementos em um "bucket" específico.

Parâmetros

n	O índice do "bucket" (deve estar em [0, bucket_count() - 1]).

Retorna

size_t O número de elementos no "bucket" `n`.

Excepções

std::out_of_range	se `n` for um índice inválido.

Definido na linha 400 do ficheiro ChainedHashTable.hpp.

```
00401 {
00402     if (n >= m_table_size)
00403         throw std::out_of_range("invalid index");
00404
00405     return m_table[n].size();
00406 }
```

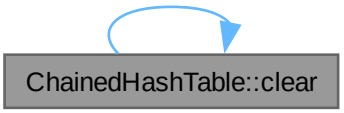
5.3.3.6 clear()

```
template<typename Key, typename Value, typename Hash>
void ChainedHashTable< Key, Value, Hash >::clear ()  [virtual]
```

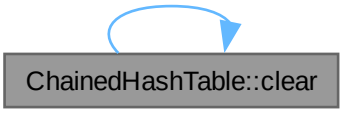
Remove todos os elementos da tabela, deixando-a com tamanho 0.
Implementa Dictionary< Key, Value >.
Definido na linha 427 do ficheiro ChainedHashTable.hpp.

```
00428 {
00429     for (size_t i = 0; i < m_table_size; ++i)
00430         m_table[i].clear();
00431
00432     m_number_of_elements = 0;
00433 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



5.3.3.7 clone()

```
template<typename Key, typename Value, typename Hash>
std::unique_ptr< Dictionary< Key, Value > > ChainedHashTable< Key, Value, Hash >::clone ()
const  [virtual]
```

Cria e retorna uma cópia profunda (deep copy) da tabela hash.

Retorna

std::unique_ptr<Dictionary<Key, Value>> Um ponteiro para a nova instância clonada.

Implementa Dictionary< Key, Value >.
Definido na linha 376 do ficheiro ChainedHashTable.hpp.

```
00377 {
00378     return std::make_unique<ChainedHashTable<Key, Value, Hash>>(*this);
00379 }
```

5.3.3.8 contains()

```
template<typename Key, typename Value, typename Hash>
bool ChainedHashTable< Key, Value, Hash >::contains (
            const Key & k)  [virtual]
```

Verifica se a tabela contém um elemento com a chave especificada.

Parâmetros

k	A chave a ser procurada.

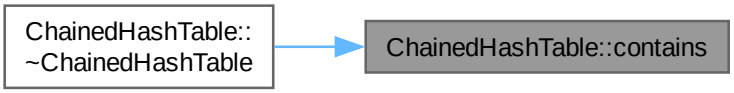
Retorna

true se um elemento com a chave `k` existir, false caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 476 do ficheiro ChainedHashTable.hpp.

```
00477 {
00478     for (auto &i : m_table[hash_code(k)])
00479     {
00480         comparisons++;
00481         if (i.first == k)
00482             return true;
00483     }
00484     return false;
00485 }
```

Este é o diagrama das funções que utilizam esta função:



5.3.3.9 empty()

```
template<typename Key, typename Value, typename Hash>
bool ChainedHashTable< Key, Value, Hash >::empty () const  [virtual], [noexcept]
```

Verifica se a tabela está vazia.

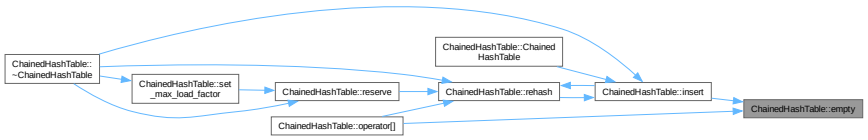
Retorna

true se a tabela não contém elementos, false caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 388 do ficheiro ChainedHashTable.hpp.

```
00389 {
00390     return m_number_of_elements == 0;
00391 }
```

Este é o diagrama das funções que utilizam esta função:



5.3.3.10 forEach()

```
template<typename Key, typename Value, typename Hash>
void ChainedHashTable< Key, Value, Hash >::forEach (
            const std::function< void(const std::pair< Key, Value > &)> & func) const  [virtual]
```

Aplica uma função a cada par chave-valor na tabela.
Itera sobre todos os elementos da tabela e executa a função `func` para cada um. A ordem de iteração não é garantida.

Parâmetros

func	A função a ser aplicada. Deve aceitar um `const std::pair<Key, Value>&`.

Implementa Dictionary< Key, Value >.
Definido na linha 611 do ficheiro ChainedHashTable.hpp.

```
00612 {
00613     for (const auto &bucket : m_table)
00614         for (const auto &pair : bucket)
00615             func(pair); // aplica a funcao a cada par chave-valor
00616 }
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.3.3.11 getCollisions()

```
template<typename Key, typename Value, typename Hash = std::hash<Key>>
long long ChainedHashTable< Key, Value, Hash >::getCollisions () const  [inline], [noexcept]
```

Retorna o número de colisões ocorridas durante as operações.
Este método é útil para análise de desempenho, permitindo verificar quantas colisões ocorreram ao longo das operações de inserção.

Retorna

O número total de colisões ocorridas.

Definido na linha 124 do ficheiro ChainedHashTable.hpp.

```
00124 { return collisions; }
```

5.3.3.12 getComparisons()

```
template<typename Key, typename Value, typename Hash = std::hash<Key>>
long long ChainedHashTable< Key, Value, Hash >::getComparisons () const  [inline], [noexcept]
```

Retorna o número de comparações realizadas durante as operações.
Este método é útil para análise de desempenho, permitindo verificar quantas comparações foram feitas ao longo das operações de inserção, busca e remoção.

Retorna

long long O número total de comparações realizadas.

Definido na linha 114 do ficheiro ChainedHashTable.hpp.

```
00114 { return comparisons; }
```

5.3.3.13 insert()

```
template<typename Key, typename Value, typename Hash>
void ChainedHashTable< Key, Value, Hash >::insert (
            const std::pair< Key, Value > & key_value)  [virtual]
```

Insere um novo par chave-valor na tabela.
A inserção só ocorre se a chave ainda não existir na tabela. Se a inserção fizer com que o fator de carga exceda o `max_load_factor`, um rehash é executado para aumentar o tamanho da tabela.

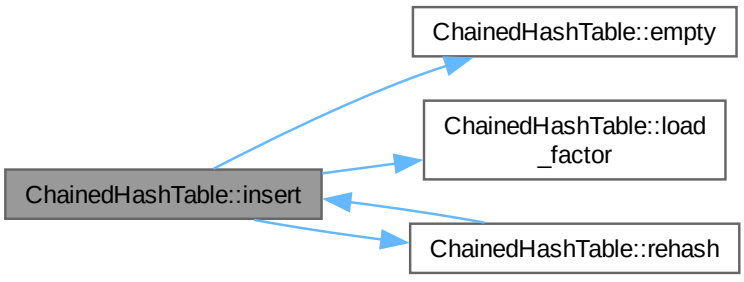
Parâmetros

key_value	O par `std::pair<Key, Value>` a ser inserido.

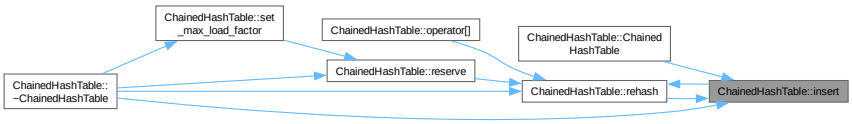
Implementa Dictionary< Key, Value >.
Definido na linha 436 do ficheiro ChainedHashTable.hpp.

```
00437 {
00438     if (load_factor() >= m_max_load_factor)
00439         rehash(m_table_size * 2);
00440
00441     size_t hash_index = hash_code(key_value.first);
00442
00443     for (const auto &pair : m_table[hash_index])
00444     {
00445         comparisons++;
00446         if (pair.first == key_value.first)
00447             return; // chave ja existe, nao adiciona
00448     }
00449
00450     if (!m_table[hash_index].empty())
00451         collisions++;
00452
00453     m_table[hash_index].push_back(key_value);
00454     m_number_of_elements++;
00455 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



5.3.3.14 load_factor()

```
template<typename Key, typename Value, typename Hash>
float ChainedHashTable< Key, Value, Hash >::load_factor () const  [noexcept]
```

Retorna o fator de carga atual da tabela. O fator de carga é a razão entre o número de elementos e o número de "buckets".

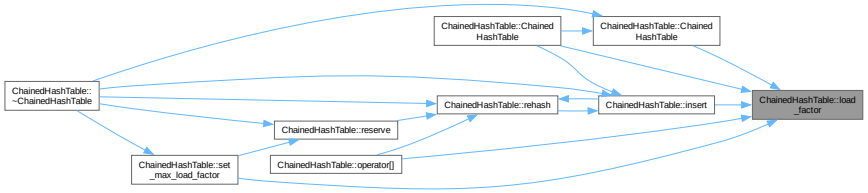
Retorna

float O fator de carga atual.

Definido na linha 415 do ficheiro ChainedHashTable.hpp.

```
00416 {
00417     return static_cast<float>(m_number_of_elements) / m_table_size;
00418 }
```

Este é o diagrama das funções que utilizam esta função:



5.3.3.15 max_load_factor()

```
template<typename Key, typename Value, typename Hash>
float ChainedHashTable< Key, Value, Hash >::max_load_factor () const  [noexcept]
```

Retorna o fator de carga máximo permitido. Se `load_factor()` exceder este valor, um rehash é acionado.

Retorna

float O fator de carga máximo.

Definido na linha 421 do ficheiro ChainedHashTable.hpp.

```
00422 {
00423     return m_max_load_factor;
00424 }
```

5.3.3.16 operator[]() [1/2]

```
template<typename Key, typename Value, typename Hash>
Value & ChainedHashTable< Key, Value, Hash >::operator[] (
            const Key & k)  [virtual]
```

Acessa ou insere um elemento.

Se a chave `k` existir, retorna uma referência ao seu valor. Se não existir, insere um novo elemento com a chave `k` (usando o construtor padrão de `Value`) e retorna uma referência ao novo valor.

Parâmetros

k	A chave do elemento a ser acessado ou inserido.

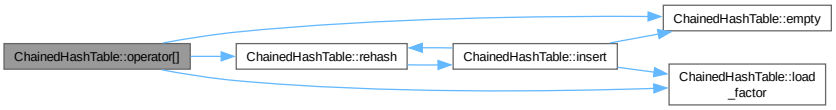
Retorna

Value& Uma referência ao valor do elemento.

Implementa Dictionary< Key, Value >.
Definido na linha 575 do ficheiro ChainedHashTable.hpp.

```
00576 {
00577     if (load_factor() >= m_max_load_factor)
00578         rehash(2 * m_table_size);
00579
00580     size_t slot = hash_code(k);
00581
00582     for (auto &pair : m_table[slot])
00583     {
00584         comparisons++;
00585         if (pair.first == k)
00586             return pair.second; // retorna o valor associado a chave
00587     }
00588
00589     if (!m_table[slot].empty())
00590         collisions++;
00591
00592     m_table[slot].push_back({k, Value()}); // insere um novo elemento com valor padrão
00593     m_number_of_elements++;
00594     return m_table[slot].back().second; // retorna o valor associado a chave
00595 }
```

Grafo de chamadas desta função:



5.3.3.17 operator[]() [2/2]

```
template<typename Key, typename Value, typename Hash>
const Value & ChainedHashTable< Key, Value, Hash >::operator[] (
            const Key & k) const
```

Acessa um elemento (versão const).
Se a chave `k` existir, retorna uma referência constante ao seu valor.

Parâmetros

k	A chave do elemento a ser acessado.

Retorna

const Value& Uma referência constante ao valor do elemento.

Excepções

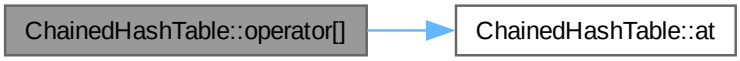
std::out_of_range	se a chave `k` não for encontrada.

Definido na linha 598 do ficheiro ChainedHashTable.hpp.

```
00599 {
00600     return at(k); // chama a funcao at para obter o valor associado a chave
```

```
00601 }
```

Grafo de chamadas desta função:



5.3.3.18 print()

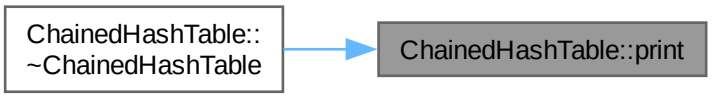
```
template<typename Key, typename Value, typename Hash>
void ChainedHashTable< Key, Value, Hash >::print () const  [virtual]
```

Imprime o conteúdo da tabela no formato [chave1:valor1, chave2:valor2, ...].
Útil para depuração. A ordem dos elementos não é garantida.
Implementa Dictionary< Key, Value >.
Definido na linha 604 do ficheiro ChainedHashTable.hpp.

```
00605 {
00606     forEach([](const std::pair<Key, Value> &par)
00607             { std::cout « "[" « par.first « ", " « par.second « "]" « std::endl; });
00608 }
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:



5.3.3.19 rehash()

```
template<typename Key, typename Value, typename Hash>
void ChainedHashTable< Key, Value, Hash >::rehash (
            size_t m)
```

Solicita uma alteração no número de "buckets" da tabela.
Se `m` for maior que o `bucket_count()` atual, a tabela é recriada (rehash) com um tamanho de pelo menos `m` "buckets". Caso contrário, a chamada não tem efeito.

Parâmetros

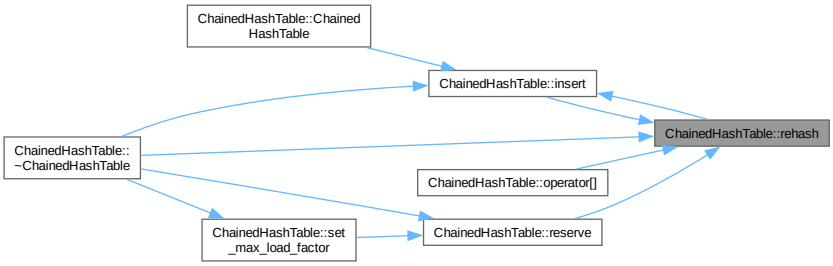
m	O número mínimo desejado de "buckets".

Definido na linha 518 do ficheiro ChainedHashTable.hpp.

```
00519 {
00520     size_t new_table_size = get_next_prime(m);
00521
00522     if (new_table_size > m_table_size)
00523     {
00524         std::vector<std::list<std::pair<Key, Value>» aux;
00525         m_table.swap(aux);
00526         m_table.resize(new_table_size);
00527
00528         m_table_size = new_table_size;
00529         m_number_of_elements = 0;
00530
00531         for (auto &vec : aux)
00532             for (auto &listas : vec)
00533                 insert({listas.first, listas.second});
00534     }
00535 }
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:



5.3.3.20 remove()

```
template<typename Key, typename Value, typename Hash>
void ChainedHashTable< Key, Value, Hash >::remove (
            const Key & k)  [virtual]
```

Remove um elemento da tabela pela chave.

Se um elemento com a chave `k` existir, ele é removido da tabela e o número de elementos é decrementado. Se a chave não for encontrada, a função não realiza nenhuma operação.

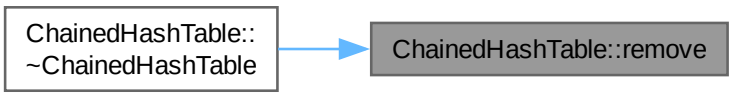
Parâmetros

k	A chave do elemento a ser removido.

Implementa Dictionary< Key, Value >.
Definido na linha 538 do ficheiro ChainedHashTable.hpp.

```
00539 {
00540     size_t slot = hash_code(k); // calcula o slot em que estaria a chave
00541     for (auto it = m_table[slot].begin(); it != m_table[slot].end(); ++it)
00542     {
00543         comparisons++;
00544         if (it->first == k)
00545         {
00546             m_table[slot].erase(it); // se encontrar, deleta
00547             m_number_of_elements--;
00548             return; // sai da funcao apos remover
00549         }
00550     }
00551 }
```

Este é o diagrama das funções que utilizam esta função:



5.3.3.21 reserve()

```
template<typename Key, typename Value, typename Hash>
void ChainedHashTable< Key, Value, Hash >::reserve (
            size_t n)  [noexcept]
```

Reserva espaço para pelo menos `n` elementos.
Se a capacidade atual não for suficiente para `n` elementos (considerando o `max_load_factor`), a tabela é redimensionada (rehash) para acomodá-los. A verificação é `n > bucket_count() * max_load_factor()`.

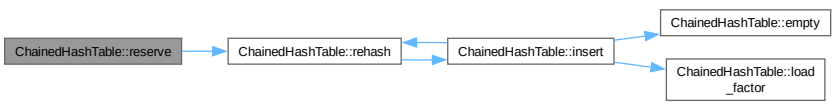
Parâmetros

n	O número mínimo de elementos que a tabela deve ser capaz de conter.

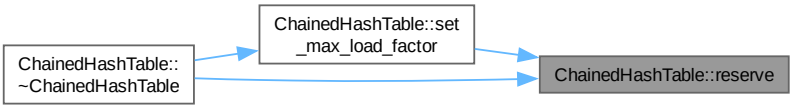
Definido na linha 554 do ficheiro ChainedHashTable.hpp.

```
00555 {
00556     if (n > m_table_size * m_max_load_factor)
00557         rehash(n / m_max_load_factor);
00558 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



5.3.3.22 set_max_load_factor()

```
template<typename Key, typename Value, typename Hash>
void ChainedHashTable< Key, Value, Hash >::set_max_load_factor (
            float lf)
```

Define o fator de carga máximo.

Altera o fator de carga máximo para `lf`. Após a alteração, a tabela pode ser redimensionada se o fator de carga atual exceder o novo máximo.

Parâmetros

lf	O novo valor para o fator de carga máximo (deve ser > 0).

Excepções

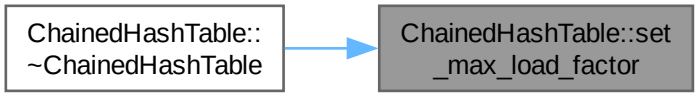
std::out_of_range	se `lf` não for positivo.

Definido na linha 561 do ficheiro ChainedHashTable.hpp.

```
00562 {
00563     if (lf <= 0)
00564         throw std::out_of_range("max load factor must be greater than 0");
00565 
00566     m_max_load_factor = lf;
00567 
00568     // Se o novo fator de carga for menor que o atual,
00569     // podemos precisar redimensionar a tabela.
00570     if (load_factor() > m_max_load_factor)
00571         reserve(m_number_of_elements);
00572 }
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:



5.3.3.23 size()

```
template<typename Key, typename Value, typename Hash>
size_t ChainedHashTable< Key, Value, Hash >::size () const  [virtual], [noexcept]
```

Retorna o número de pares chave-valor na tabela.

Retorna

size_t O número de elementos.

Implementa Dictionary< Key, Value >.
Definido na linha 382 do ficheiro ChainedHashTable.hpp.

```
00383 {
00384     return m_number_of_elements;
00385 }
```

5.3.3.24 update()

```
template<typename Key, typename Value, typename Hash>
void ChainedHashTable< Key, Value, Hash >::update (
            const std::pair< Key, Value > & key_value)  [virtual]
```

Atualiza o valor associado a uma chave existente.
Se a chave `key_value.first` for encontrada na tabela, seu valor correspondente é atualizado para `key_value.second`. Se a chave não existir, a função não realiza nenhuma operação.

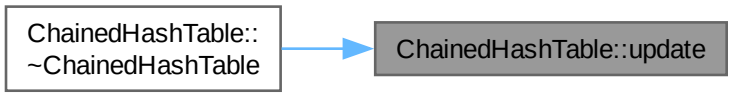
Parâmetros

key_value	O par `std::pair<Key, Value>` contendo a chave a ser encontrada e o novo valor.

Implementa Dictionary< Key, Value >.
Definido na linha 458 do ficheiro ChainedHashTable.hpp.

```
00459 {
00460     size_t hash_index = hash_code(key_value.first);
00461
00462     for (auto &pair : m_table[hash_index])
00463     {
00464         comparisons++;
00465         if (pair.first == key_value.first)
00466         {
00467             pair.second = key_value.second; // atualiza o valor associado a chave
00468             return;
00469         }
00470     }
00471
00472     throw std::out_of_range("Key not found in the hash table");
00473 }
```

Este é o diagrama das funções que utilizam esta função:

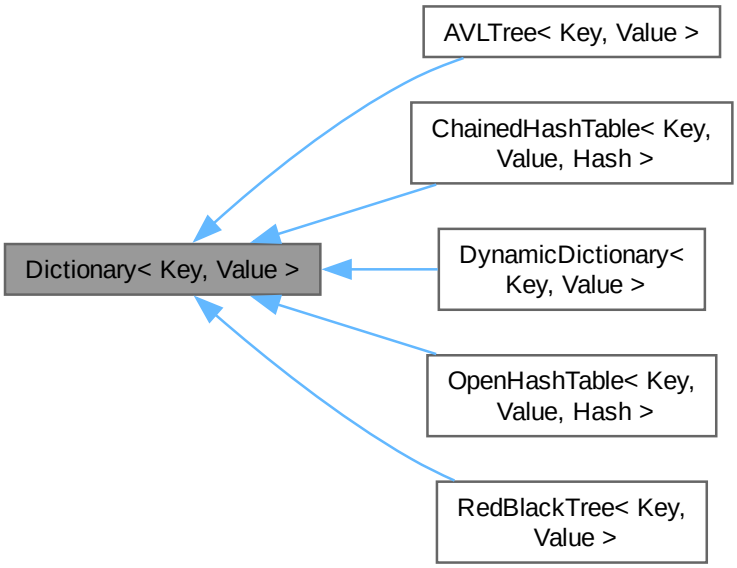


A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/hash_table_c/ChainedHashTable.hpp

5.4 Referência à classe Template Dictionary< Key, Value >

Define a interface para uma estrutura de dados de dicionário (ou mapa).
`#include <Dictionary.hpp>`
Diagrama de heranças da classe Dictionary< Key, Value >



Membros públicos

- virtual ~Dictionary ()=default

  Destrutor virtual para permitir a destruição correta de classes derivadas.
- virtual void insert (const std::pair< Key, Value > &key_value)=0

  Insere um par chave-valor no dicionário.
- virtual void remove (const Key &key)=0

  Remove um elemento do dicionário pela chave.

- virtual void update (const std::pair< Key, Value > &key_value)=0

  Atualiza o valor associado a uma chave existente no dicionário.
- virtual bool contains (const Key &key)=0

  Verifica se uma chave está presente no dicionário.
- virtual Value & at (const Key &key)=0

  Obtém o valor associado a uma chave.
- virtual Value & operator[ ] (const Key &key)=0

  Sobrecarga do operador de indexação para acessar o valor associado a uma chave.
- virtual void clear ()=0

  Limpa todos os elementos do dicionário.
- virtual size_t size () const noexcept=0

  Obtém o número de elementos no dicionário.
- virtual bool empty () const noexcept=0

  Verifica se o dicionário está vazio.
- virtual void print () const =0

  Imprime o conteúdo do dicionário.
- virtual void forEach (const std::function< void(const std::pair< Key, Value > &)> &func) const =0

  Itera sobre todos os pares chave-valor no dicionário e aplica uma função a cada um.
- virtual std::unique_ptr< Dictionary< Key, Value > > clone () const =0

  Clona o dicionário atual.

5.4.1 Descrição detalhada

template<typename Key, typename Value>
class Dictionary< Key, Value >

Define a interface para uma estrutura de dados de dicionário (ou mapa).
Um dicionário é uma coleção de pares chave-valor, onde cada chave é única. Esta classe de interface pura (abstrata) estabelece o contrato que todas as implementações de dicionário devem seguir.

Parâmetros de template

Key	O tipo da chave. Deve ser único para cada elemento no dicionário.
Value	O tipo do valor associado a uma chave.

Definido na linha 17 do ficheiro Dictionary.hpp.

5.4.2 Documentação dos Construtores & Destrutor

5.4.2.1 ~Dictionary()

```
template<typename Key, typename Value>
virtual Dictionary< Key, Value >::~Dictionary ()  [virtual], [default]
```

Destrutor virtual para permitir a destruição correta de classes derivadas.

5.4.3 Documentação das funções

5.4.3.1 at()

```
template<typename Key, typename Value>
virtual Value & Dictionary< Key, Value >::at (
            const Key & key)  [pure virtual]
```

Obtém o valor associado a uma chave.

Parâmetros

key	A chave cujo valor será obtido.

Retorna

O valor associado à chave.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.

5.4.3.2 clear()

```
template<typename Key, typename Value>
virtual void Dictionary< Key, Value >::clear ()  [pure virtual]
```

Limpa todos os elementos do dicionário.
Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.

5.4.3.3 clone()

```
template<typename Key, typename Value>
virtual std::unique_ptr< Dictionary< Key, Value > > Dictionary< Key, Value >::clone () const
[pure virtual]
```

Clona o dicionário atual.

Retorna

Um ponteiro único para uma nova instância do dicionário com os mesmos elementos.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.
Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.4.3.4 contains()

```
template<typename Key, typename Value>
virtual bool Dictionary< Key, Value >::contains (
            const Key & key)  [pure virtual]
```

Verifica se uma chave está presente no dicionário.

Parâmetros

key	A chave a ser verificada.

Retorna

true Se a chave estiver presente.

false Se a chave não estiver presente.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.

5.4.3.5 empty()

```
template<typename Key, typename Value>
virtual bool Dictionary< Key, Value >::empty () const  [pure virtual], [noexcept]
```

Verifica se o dicionário está vazio.

Retorna

true Se o dicionário não contiver elementos.

false Se o dicionário contiver pelo menos um elemento.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.
Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.4.3.6 forEach()

```
template<typename Key, typename Value>
virtual void Dictionary< Key, Value >::forEach (
            const std::function< void(const std::pair< Key, Value > &)> & func) const  [pure
virtual]
```

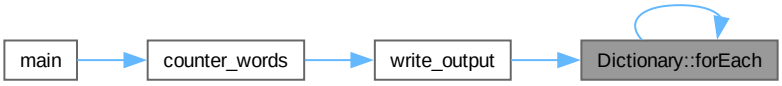
Itera sobre todos os pares chave-valor no dicionário e aplica uma função a cada um.

Parâmetros

func	A função a ser aplicada a cada par chave-valor.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.
Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:



5.4.3.7 insert()

```
template<typename Key, typename Value>
virtual void Dictionary< Key, Value >::insert (
            const std::pair< Key, Value > & key_value)  [pure virtual]
```

Insere um par chave-valor no dicionário.

Parâmetros

key_value	O par chave-valor a ser inserido.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.

5.4.3.8 operator[]()

```
template<typename Key, typename Value>
virtual Value & Dictionary< Key, Value >::operator[] (
            const Key & key)  [pure virtual]
```

Sobrecarga do operador de indexação para acessar o valor associado a uma chave.

Parâmetros

key	A chave cujo valor será acessado.

Retorna

O valor associado à chave.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.

5.4.3.9 print()

```
template<typename Key, typename Value>
virtual void Dictionary< Key, Value >::print () const  [pure virtual]
```

Imprime o conteúdo do dicionário.
Esta função deve ser implementada para exibir os pares chave-valor de forma legível, dependendo da implementação específica do dicionário.
Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.
Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.4.3.10 remove()

```
template<typename Key, typename Value>
virtual void Dictionary< Key, Value >::remove (
            const Key & key)  [pure virtual]
```

Remove um elemento do dicionário pela chave.

Parâmetros

key	A chave do elemento a ser removido.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.

5.4.3.11 size()

```
template<typename Key, typename Value>
virtual size_t Dictionary< Key, Value >::size () const  [pure virtual], [noexcept]
```

Obtém o número de elementos no dicionário.

Retorna

O número de elementos no dicionário.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.
Este é o diagrama das funções que utilizam esta função:

5.4.3.12 update()

```
template<typename Key, typename Value>
virtual void Dictionary< Key, Value >::update (
            const std::pair< Key, Value > & key_value)  [pure virtual]
```

Atualiza o valor associado a uma chave existente no dicionário.

Parâmetros

key_value	O par chave-valor a ser atualizado.

Implementado em AVLTree< Key, Value >, AVLTree< int, std::string >, ChainedHashTable< Key, Value, Hash >, DynamicDictionary< Key, Value >, OpenHashTable< Key, Value, Hash > e RedBlackTree< Key, Value >.
A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/Dictionary.hpp

5.5 Referência à classe Template DictionaryTest< T >

Fixture genérico para a interface Dictionary.

Diagrama de heranças da classe DictionaryTest< T >

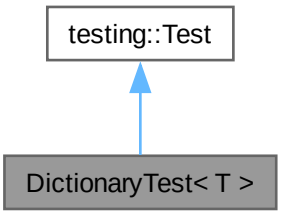
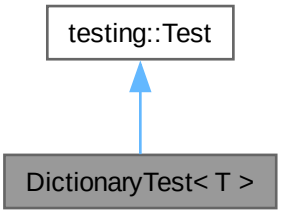


Diagrama de colaboração para DictionaryTest< T >:



Membros protegidos

- void SetUp () override
- void TearDown () override

Atributos Protegidos

- std::unique_ptr< Dictionary< int, std::string > > dict

5.5.1 Descrição detalhada

template<typename T>
class DictionaryTest< T >

Fixture genérico para a interface Dictionary.
Definido na linha 16 do ficheiro Tests.cpp.

5.5.2 Documentação das funções

5.5.2.1 SetUp()

```
template<typename T>
void DictionaryTest< T >::SetUp ()  [inline], [override], [protected]
```

Definido na linha 21 do ficheiro Tests.cpp.

```
00022     {
00023         // Cria uma nova instância da implementação de dicionário para cada teste
```

```
00024         dict = std::make_unique<T>();
00025     }
```

5.5.2.2 TearDown()

```
template<typename T>
void DictionaryTest< T >::TearDown ()  [inline], [override], [protected]
```

Definido na linha 27 do ficheiro Tests.cpp.

```
00028     {
00029         // unique_ptr lida com a desalocação automaticamente
00030     }
```

5.5.3 Documentação dos dados membro

5.5.3.1 dict

```
template<typename T>
std::unique_ptr<Dictionary<int, std::string> > DictionaryTest< T >::dict  [protected]
```

Definido na linha 19 do ficheiro Tests.cpp.

A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- tests/Tests.cpp

5.6 Referência à classe Template DynamicDictionary< Key, Value >

Uma classe de dicionário dinâmico que atua como um wrapper (invólucro).

`#include <DynamicDictionary.hpp>`

Diagrama de heranças da classe DynamicDictionary< Key, Value >

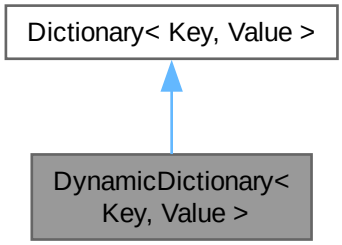
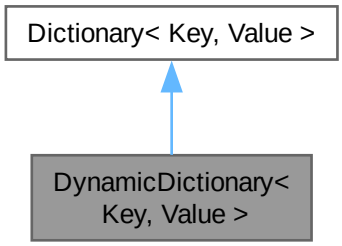


Diagrama de colaboração para DynamicDictionary< Key, Value >:



Membros públicos

- DynamicDictionary (DictionaryType type=DictionaryType::RBTREE)

  Construtor que cria um dicionário do tipo especificado.
- DynamicDictionary (const DynamicDictionary &other)

  Construtor de cópia.
- DynamicDictionary (const std::initializer_list< std::pair< Key, Value > > list, DictionaryType type=DictionaryType::RBTREE)

  Construtor que inicializa o dicionário com uma lista de pares chave-valor.
- std::unique_ptr< Dictionary< Key, Value > > clone () const

  Cria e retorna um clone (cópia profunda) do objeto DynamicDictionary atual.
- void insert (const std::pair< Key, Value > &key_value)

  Insere um novo par chave-valor no dicionário.
- void remove (const Key &key)

  Remove um elemento do dicionário com base na chave.
- void update (const std::pair< Key, Value > &key_value)

  Atualiza o valor de uma chave existente.
- bool contains (const Key &key)

  Verifica se o dicionário contém um elemento com a chave especificada.
- Value & at (const Key &key)

  Acessa o valor associado a uma chave.
- DynamicDictionary & operator= (const DynamicDictionary &other)

  Operador de atribuição de cópia.
- Value & operator[ ] (const Key &key)

  Acessa o valor associado a uma chave.
- void clear ()

  Remove todos os elementos do dicionário.
- size_t size () const noexcept

  Retorna o número de elementos no dicionário.
- bool empty () const noexcept

  Verifica se o dicionário está vazio.
- void print () const

  Imprime o conteúdo do dicionário na saída padrão.
- void forEach (const std::function< void(const std::pair< Key, Value > &)> &func) const

  Aplica uma função a cada par chave-valor no dicionário.
- Dictionary< Key, Value > & get_dictionary () const

  Obtém uma referência para a implementação do dicionário subjacente.

Membros públicos herdados de Dictionary< Key, Value >

- virtual ~Dictionary ()=default

  Destrutor virtual para permitir a destruição correta de classes derivadas.

5.6.1 Descrição detalhada

template<typename Key, typename Value>
class DynamicDictionary< Key, Value >

Uma classe de dicionário dinâmico que atua como um wrapper (invólucro).
Esta classe permite a criação de diferentes tipos de dicionários (como Árvore Rubro-Negra, Tabela Hash, etc.) em tempo de execução, com base no `DictionaryType` fornecido. Ela delega todas as operações para a implementação de dicionário subjacente que ela encapsula, fornecendo uma interface uniforme.

Parâmetros de template

Key	O tipo da chave dos elementos no dicionário.
Value	O tipo do valor associado a cada chave.

Definido na linha 17 do ficheiro DynamicDictionary.hpp.

5.6.2 Documentação dos Construtores & Destrutor

5.6.2.1 DynamicDictionary() [1/3]

```
template<typename Key, typename Value>
DynamicDictionary< Key, Value >::DynamicDictionary (
            DictionaryType type = DictionaryType::RBTREE)  [inline]
```

Construtor que cria um dicionário do tipo especificado.

Parâmetros

type	O tipo de dicionário a ser criado (por exemplo, RBTREE, HASHTABLE). O padrão é RBTREE.

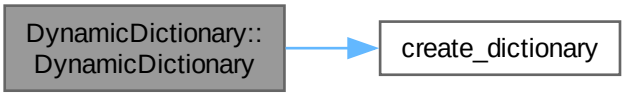
Excepções

std::runtime_error	se o tipo de dicionário for inválido e a criação falhar.

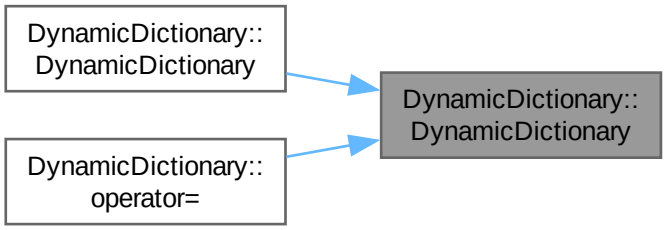
Definido na linha 39 do ficheiro DynamicDictionary.hpp.

```
00039                                                                   : type(type),
      dictionary(create_dictionary<Key, Value>(type))
00040     {
00041         check_dictionary();
00042     }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



5.6.2.2 DynamicDictionary() [2/3]

```
template<typename Key, typename Value>
DynamicDictionary< Key, Value >::DynamicDictionary (
            const DynamicDictionary< Key, Value > & other)  [inline]
```

Construtor de cópia.
Cria uma cópia profunda do outro dicionário, clonando sua implementação subjacente através do método `clone()`.

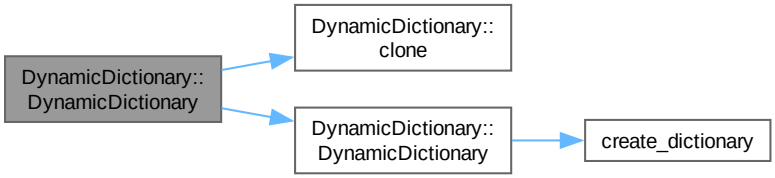
Parâmetros

other	O `DynamicDictionary` a ser copiado.

Definido na linha 52 do ficheiro DynamicDictionary.hpp.

```
00052                                                   : type(other.type),
      dictionary(other.dictionary->clone())
00053     {
00054         check_dictionary();
00055     }
```

Grafo de chamadas desta função:



5.6.2.3 DynamicDictionary() [3/3]

```
template<typename Key, typename Value>
DynamicDictionary< Key, Value >::DynamicDictionary (
            const std::initializer_list< std::pair< Key, Value > > list,
            DictionaryType type = DictionaryType::RBTREE)  [inline]
```

Construtor que inicializa o dicionário com uma lista de pares chave-valor.

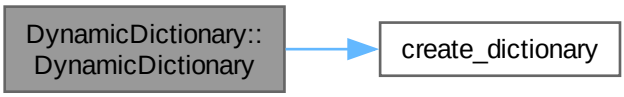
Parâmetros

list	A `std::initializer_list` contendo os pares chave-valor para popular o dicionário.
type	O tipo de dicionário a ser criado. O padrão é RBTREE.

Definido na linha 62 do ficheiro DynamicDictionary.hpp.

```
00063         : type(type), dictionary(create_dictionary<Key, Value>(type, list))
00064     {
00065         check_dictionary();
00066     }
```

Grafo de chamadas desta função:



5.6.3 Documentação das funções

5.6.3.1 at()

```
template<typename Key, typename Value>
Value & DynamicDictionary< Key, Value >::at (
             const Key & key)  [inline], [virtual]
```

Acessa o valor associado a uma chave.
Lança uma exceção se a chave não for encontrada.

Parâmetros

key	A chave do valor a ser acessado.

Retorna

Uma referência ao valor associado à chave.

Excepções

std::out_of_range	(ou similar, dependendo da implementação subjacente) se a chave não for encontrada.

Implementa Dictionary< Key, Value >.
Definido na linha 132 do ficheiro DynamicDictionary.hpp.

```
00133     {
00134         return dictionary->at(key);
00135     }
```

5.6.3.2 clear()

```
template<typename Key, typename Value>
void DynamicDictionary< Key, Value >::clear ()  [inline], [virtual]
```

Remove todos os elementos do dicionário.
Implementa Dictionary< Key, Value >.
Definido na linha 171 do ficheiro DynamicDictionary.hpp.

```
00172     {
00173         dictionary->clear();
00174     }
```

5.6.3.3 clone()

```
template<typename Key, typename Value>
std::unique_ptr< Dictionary< Key, Value > > DynamicDictionary< Key, Value >::clone () const
[inline], [virtual]
```

Cria e retorna um clone (cópia profunda) do objeto DynamicDictionary atual.

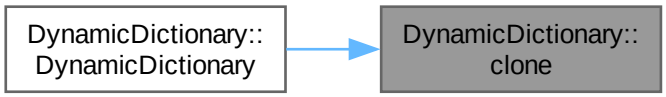
Retorna

Um `std::unique_ptr<Dictionary<Key, Value>>` para o novo dicionário clonado.

Implementa Dictionary< Key, Value >.
Definido na linha 72 do ficheiro DynamicDictionary.hpp.

```
00073     {
00074         return std::make_unique<DynamicDictionary<Key, Value>>(*this);
00075     }
```

Este é o diagrama das funções que utilizam esta função:



5.6.3.4 contains()

```
template<typename Key, typename Value>
bool DynamicDictionary< Key, Value >::contains (
            const Key & key)  [inline], [virtual]
```

Verifica se o dicionário contém um elemento com a chave especificada.

Parâmetros

key	A chave a ser procurada.

Retorna

`true` se a chave existir, `false` caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 118 do ficheiro DynamicDictionary.hpp.

```
00119     {
00120         return dictionary->contains(key);
00121     }
```

5.6.3.5 empty()

```
template<typename Key, typename Value>
bool DynamicDictionary< Key, Value >::empty () const  [inline], [virtual], [noexcept]
```

Verifica se o dicionário está vazio.

Retorna

`true` se o dicionário estiver vazio, `false` caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 189 do ficheiro DynamicDictionary.hpp.

```
00190     {
00191         return dictionary->empty();
00192     }
```

5.6.3.6 forEach()

```
template<typename Key, typename Value>
void DynamicDictionary< Key, Value >::forEach (
            const std::function< void(const std::pair< Key, Value > &)> & func) const  [inline],
[virtual]
```

Aplica uma função a cada par chave-valor no dicionário.

Parâmetros

func	A função (`std::function`) a ser executada para cada elemento. A função recebe um `const std::pair<Key, Value>&`.

Implementa Dictionary< Key, Value >.
Definido na linha 208 do ficheiro DynamicDictionary.hpp.

```
00209     {
00210         dictionary->forEach(func);
00211     }
```

5.6.3.7 get_dictionary()

```
template<typename Key, typename Value>
Dictionary< Key, Value > & DynamicDictionary< Key, Value >::get_dictionary () const  [inline]
```

Obtém uma referência para a implementação do dicionário subjacente.

Retorna

Uma referência ao objeto `Dictionary<Key, Value>` encapsulado.

Definido na linha 217 do ficheiro DynamicDictionary.hpp.

```
00218     {
00219         return *dictionary;
00220     }
```

5.6.3.8 insert()

```
template<typename Key, typename Value>
void DynamicDictionary< Key, Value >::insert (
            const std::pair< Key, Value > & key_value)  [inline], [virtual]
```

Insere um novo par chave-valor no dicionário.
A operação é delegada para a implementação do dicionário subjacente.

Parâmetros

key_value	O `std::pair<Key, Value>` a ser inserido.

Implementa Dictionary< Key, Value >.
Definido na linha 84 do ficheiro DynamicDictionary.hpp.

```
00085     {
00086         dictionary->insert(key_value);
00087     }
```

5.6.3.9 operator=()

```
template<typename Key, typename Value>
DynamicDictionary & DynamicDictionary< Key, Value >::operator= (
            const DynamicDictionary< Key, Value > & other)  [inline]
```

Operador de atribuição de cópia.
Substitui o conteúdo do dicionário atual por uma cópia profunda do outro dicionário.

Parâmetros

other	O outro `DynamicDictionary` a ser copiado.

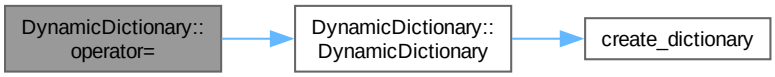
Retorna

Uma referência a este objeto (`*this`).

Definido na linha 145 do ficheiro DynamicDictionary.hpp.

```
00146     {
00147         if (this != &other)
00148         {
00149             dictionary = other.dictionary->clone;
00150         }
00151 
00152         return *this;
00153     }
```

Grafo de chamadas desta função:



5.6.3.10 operator[]()

```
template<typename Key, typename Value>
Value & DynamicDictionary< Key, Value >::operator[] (
            const Key & key)  [inline], [virtual]
```

Acessa o valor associado a uma chave.
Se a chave não existir, ela é inserida com um valor padrão e uma referência a esse novo valor é retornada.

Parâmetros

key	A chave do elemento a ser acessado ou inserido.

Retorna

Uma referência ao valor associado à chave.

Implementa Dictionary< Key, Value >.
Definido na linha 163 do ficheiro DynamicDictionary.hpp.

```
00164     {
00165         return dictionary->operator[](key);
00166     }
```

5.6.3.11 print()

```
template<typename Key, typename Value>
void DynamicDictionary< Key, Value >::print () const  [inline], [virtual]
```

Imprime o conteúdo do dicionário na saída padrão.
O formato da impressão depende da implementação do dicionário subjacente.
Implementa Dictionary< Key, Value >.
Definido na linha 199 do ficheiro DynamicDictionary.hpp.

```
00200     {
00201         dictionary->print();
00202     }
```

5.6.3.12 remove()

```
template<typename Key, typename Value>
void DynamicDictionary< Key, Value >::remove (
            const Key & key)  [inline], [virtual]
```

Remove um elemento do dicionário com base na chave.
A operação é delegada para a implementação do dicionário subjacente.

Parâmetros

key	A chave do elemento a ser removido.

Implementa Dictionary< Key, Value >.
Definido na linha 96 do ficheiro DynamicDictionary.hpp.

```
00097     {
00098         dictionary->remove(key);
00099     }
```

5.6.3.13 size()

```
template<typename Key, typename Value>
size_t DynamicDictionary< Key, Value >::size () const  [inline], [virtual], [noexcept]
```

Retorna o número de elementos no dicionário.

Retorna

O número de pares chave-valor.

Implementa Dictionary< Key, Value >.
Definido na linha 180 do ficheiro DynamicDictionary.hpp.

```
00181     {
00182         return dictionary->size();
00183     }
```

5.6.3.14 update()

```
template<typename Key, typename Value>
void DynamicDictionary< Key, Value >::update (
            const std::pair< Key, Value > & key_value)  [inline], [virtual]
```

Atualiza o valor de uma chave existente.
A operação é delegada para a implementação do dicionário subjacente.

Parâmetros

key_value	O `std::pair<Key, Value>` contendo a chave a ser encontrada e o novo valor.

Implementa Dictionary< Key, Value >.
Definido na linha 108 do ficheiro DynamicDictionary.hpp.

```
00109     {
00110         dictionary->update(key_value);
00111     }
```

A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/DynamicDictionary.hpp

5.7 Referência à classe Template GeneralStressTest< T >

Diagrama de heranças da classe GeneralStressTest< T >

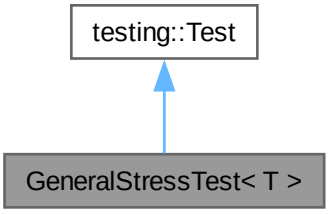
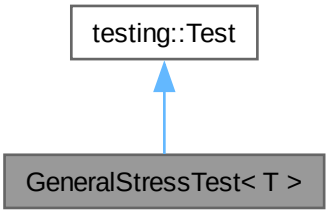


Diagrama de colaboração para GeneralStressTest< T >:



Membros protegidos

- void SetUp () override

Atributos Protegidos

- std::unique_ptr< Dictionary< int, std::string > > dict

5.7.1 Descrição detalhada

template<typename T>
class GeneralStressTest< T >

Definido na linha 426 do ficheiro Tests.cpp.

5.7.2 Documentação das funções

5.7.2.1 SetUp()

```
template<typename T>
void GeneralStressTest< T >::SetUp ()  [inline], [override], [protected]
```

Definido na linha 431 do ficheiro Tests.cpp.

```
00432     {
00433         dict = std::make_unique<T>();
00434     }
```

5.7.3 Documentação dos dados membro

5.7.3.1 dict

```
template<typename T>
std::unique_ptr<Dictionary<int, std::string> > GeneralStressTest< T >::dict  [protected]
```

Definido na linha 429 do ficheiro Tests.cpp.
A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- tests/Tests.cpp

5.8 Referência à classe Template HashTableStressTest< T >

Usamos novamente um teste tipado para aplicar os mesmos testes de estresse para ChainedHashTable e OpenHashTable.
Diagrama de heranças da classe HashTableStressTest< T >

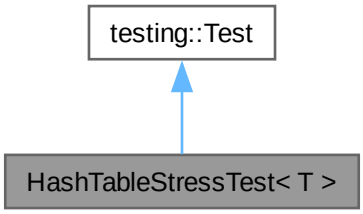
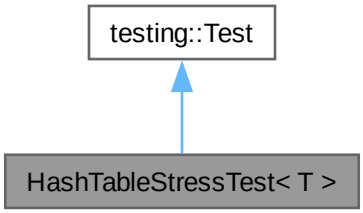


Diagrama de colaboração para HashTableStressTest< T >:



Membros protegidos

- void SetUp () override

Atributos Protegidos

- std::unique_ptr< Dictionary< int, std::string > > hashTable

5.8.1 Descrição detalhada

template<typename T>
class HashTableStressTest< T >

Usamos novamente um teste tipado para aplicar os mesmos testes de estresse para ChainedHashTable e OpenHashTable.
Definido na linha 320 do ficheiro Tests.cpp.

5.8.2 Documentação das funções

5.8.2.1 SetUp()

```
template<typename T>
void HashTableStressTest< T >::SetUp ()  [inline], [override], [protected]
```

Definido na linha 325 do ficheiro Tests.cpp.

```
00326     {
00327         hashTable = std::make_unique<T>();
00328     }
```

5.8.3 Documentação dos dados membro

5.8.3.1 hashTable

```
template<typename T>
std::unique_ptr<Dictionary<int, std::string> > HashTableStressTest< T >::hashTable  [protected]
```

Definido na linha 323 do ficheiro Tests.cpp.
A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- tests/Tests.cpp

5.9 Referência à classe Template IteratorAVL< Key, Value >

Um iterador para a árvore AVL.
`#include <IteratorAVL.hpp>`

Tipos Públicos

- using iterator_category = std::input_iterator_tag

  Categoria do iterador, indica que é um iterador de entrada.
- using value_type = std::pair<Key, Value>

  O tipo do valor apontado pelo iterador (um par chave-valor).
- using difference_type = std::ptrdiff_t

  Tipo para representar a diferença entre dois iteradores.
- using pointer = value_type *

  Ponteiro para o tipo de valor.
- using reference = value_type &

  Referência para o tipo de valor.
- using const_pointer = const value_type *

  Ponteiro constante para o tipo de valor.
- using const_reference = const value_type &

  Referência constante para o tipo de valor.
- using NodeType = Node<Key, Value>

  Tipo do nó da árvore AVL.
- using NodePtrType = NodePtr

  Tipo do ponteiro para o nó da árvore AVL.

Membros públicos

- IteratorAVL ()

  Construtor padrão.
- IteratorAVL (NodePtr root)

  Construtor que inicializa o iterador a partir da raiz da árvore.
- reference operator∗ () const

  Operador de derreferência.
- pointer operator-> () const

  Operador de acesso a membro.
- IteratorAVL & operator++ ()

  Operador de incremento (pré-fixado).
- IteratorAVL operator++ (int)

  Operador de incremento (pós-fixado).
- bool operator== (const IteratorAVL &other) const

  Operador de igualdade.
- bool operator!= (const IteratorAVL &other) const

  Operador de desigualdade.

5.9.1 Descrição detalhada

template<typename Key, typename Value>
class IteratorAVL< Key, Value >

Um iterador para a árvore AVL.
Esta classe fornece funcionalidade de iterador para percorrer uma árvore AVL em ordem (in-order traversal).

Parâmetros de template

Key	O tipo da chave armazenada nos nós da árvore.
Value	O tipo do valor associado à chave nos nós da árvore.

Definido na linha 20 do ficheiro IteratorAVL.hpp.

5.9.2 Documentação das definições de tipo

5.9.2.1 const_pointer

```
template<typename Key, typename Value>
using IteratorAVL< Key, Value >::const_pointer = const value_type *
```

Ponteiro constante para o tipo de valor.
Definido na linha 40 do ficheiro IteratorAVL.hpp.

5.9.2.2 const_reference

```
template<typename Key, typename Value>
using IteratorAVL< Key, Value >::const_reference = const value_type &
```

Referência constante para o tipo de valor.
Definido na linha 42 do ficheiro IteratorAVL.hpp.

5.9.2.3 difference_type

```
template<typename Key, typename Value>
using IteratorAVL< Key, Value >::difference_type = std::ptrdiff_t
```

Tipo para representar a diferença entre dois iteradores.
Definido na linha 34 do ficheiro IteratorAVL.hpp.

5.9.2.4 iterator_category

```
template<typename Key, typename Value>
using IteratorAVL< Key, Value >::iterator_category = std::input_iterator_tag
```

Categoria do iterador, indica que é um iterador de entrada.
Definido na linha 30 do ficheiro IteratorAVL.hpp.

5.9.2.5 NodePtrType

```
template<typename Key, typename Value>
using IteratorAVL< Key, Value >::NodePtrType = NodePtr
```

Tipo do ponteiro para o nó da árvore AVL.
Definido na linha 46 do ficheiro IteratorAVL.hpp.

5.9.2.6 NodeType

```
template<typename Key, typename Value>
using IteratorAVL< Key, Value >::NodeType = Node<Key, Value>
```

Tipo do nó da árvore AVL.
Definido na linha 44 do ficheiro IteratorAVL.hpp.

5.9.2.7 pointer

```
template<typename Key, typename Value>
using IteratorAVL< Key, Value >::pointer = value_type *
```

Ponteiro para o tipo de valor.
Definido na linha 36 do ficheiro IteratorAVL.hpp.

5.9.2.8 reference

```
template<typename Key, typename Value>
using IteratorAVL< Key, Value >::reference = value_type &
```

Referência para o tipo de valor.
Definido na linha 38 do ficheiro IteratorAVL.hpp.

5.9.2.9 value_type

```
template<typename Key, typename Value>
using IteratorAVL< Key, Value >::value_type = std::pair<Key, Value>
```

O tipo do valor apontado pelo iterador (um par chave-valor).
Definido na linha 32 do ficheiro IteratorAVL.hpp.

5.9.3 Documentação dos Construtores & Destrutor

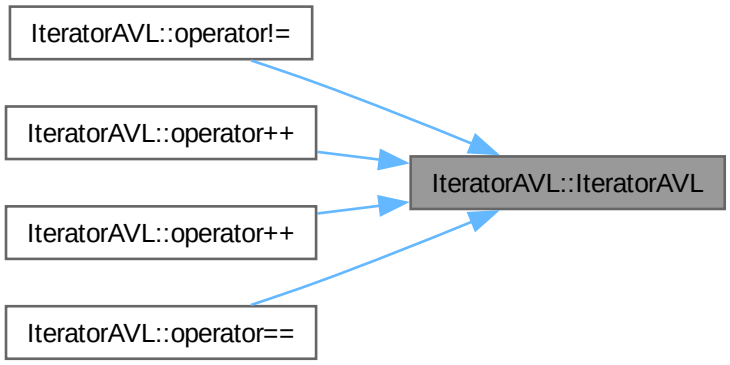
5.9.3.1 IteratorAVL() [1/2]

```
template<typename Key, typename Value>
IteratorAVL< Key, Value >::IteratorAVL ()  [inline]
```

Construtor padrão.
Cria um iterador inválido (geralmente usado para representar o fim de uma coleção).
Definido na linha 53 do ficheiro IteratorAVL.hpp.

```
00053 {}
```

Este é o diagrama das funções que utilizam esta função:



5.9.3.2 IteratorAVL() [2/2]

```
template<typename Key, typename Value>
IteratorAVL< Key, Value >::IteratorAVL (
            NodePtr root)  [inline]
```

Construtor que inicializa o iterador a partir da raiz da árvore.
O iterador é posicionado no primeiro elemento da travessia em ordem (o nó mais à esquerda).

Parâmetros

root	Ponteiro para o nó raiz da árvore AVL.

Definido na linha 63 do ficheiro IteratorAVL.hpp.

```
00064     {
00065         NodePtr current = root;
00066         while (current != nullptr)
00067         {
00068             path.push(current);
00069             current = current->left;
00070         }
00071     }
```

5.9.4 Documentação das funções

5.9.4.1 operator"!=()

```
template<typename Key, typename Value>
bool IteratorAVL< Key, Value >::operator!= (
            const IteratorAVL< Key, Value > & other) const  [inline]
```

Operador de desigualdade.
Compara este iterador com outro iterador para verificar se apontam para nós diferentes.

Parâmetros

other	O outro iterador a ser comparado.

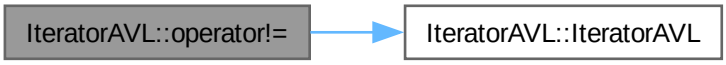
Retorna

`true` se os iteradores são diferentes, `false` caso contrário.

Definido na linha 170 do ficheiro IteratorAVL.hpp.

```
00171     {
00172         return !(*this == other);
00173     }
```

Grafo de chamadas desta função:



5.9.4.2 operator∗()

```
template<typename Key, typename Value>
reference IteratorAVL< Key, Value >::operator* () const  [inline]
```

Operador de derreferência.
Retorna uma referência para o par chave-valor do nó atual.

Retorna

Referência para o par chave-valor do nó atual.

Definido na linha 80 do ficheiro IteratorAVL.hpp.

```
00081     {
00082         return path.top()->key;
00083     }
```

5.9.4.3 operator++() [1/2]

```
template<typename Key, typename Value>
IteratorAVL & IteratorAVL< Key, Value >::operator++ ()  [inline]
```

Operador de incremento (pré-fixado).
Avança o iterador para o próximo nó na travessia em ordem.

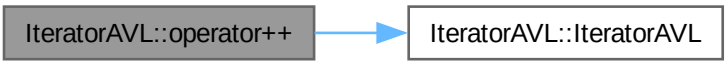
Retorna

Referência para o iterador atualizado.

Definido na linha 104 do ficheiro IteratorAVL.hpp.

```
00105     {
00106         if (path.empty())
00107             return *this;
00108 
00109         NodePtr node = path.top();
00110         path.pop();
00111 
00112         if (node->right != nullptr)
00113         {
00114             NodePtr current = node->right;
00115 
00116             while (current != nullptr)
00117             {
00118                 path.push(current);
00119                 current = current->left;
00120             }
00121         }
00122 
00123         return *(this);
00124     }
```

Grafo de chamadas desta função:



5.9.4.4 operator++() [2/2]

```
template<typename Key, typename Value>
IteratorAVL IteratorAVL< Key, Value >::operator++ (
            int )  [inline]
```

Operador de incremento (pós-fixado).
Avança o iterador para o próximo nó na travessia em ordem e retorna uma cópia do iterador antes do incremento.

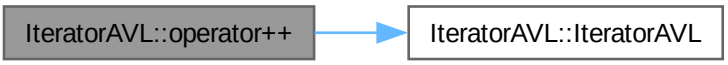
Retorna

Uma cópia do iterador antes do incremento.

Definido na linha 134 do ficheiro IteratorAVL.hpp.

```
00135     {
00136         IteratorAVL temp = *this;
00137         ++(*this);
00138         return temp;
00139     }
```

Grafo de chamadas desta função:



5.9.4.5 operator->()

```
template<typename Key, typename Value>
pointer IteratorAVL< Key, Value >::operator-> () const  [inline]
```

Operador de acesso a membro.
Retorna um ponteiro para o par chave-valor do nó atual.

Retorna

Ponteiro para o par chave-valor do nó atual.

Definido na linha 92 do ficheiro IteratorAVL.hpp.

```
00093     {
00094         return &(path.top()->key);
00095     }
```

5.9.4.6 operator==()

```
template<typename Key, typename Value>
bool IteratorAVL< Key, Value >::operator== (
            const IteratorAVL< Key, Value > & other) const  [inline]
```

Operador de igualdade.
Compara este iterador com outro iterador para verificar se apontam para o mesmo nó.

Parâmetros

other	O outro iterador a ser comparado.

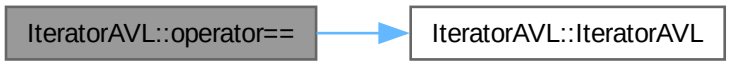
Retorna

`true` se os iteradores são iguais, `false` caso contrário.

Definido na linha 150 do ficheiro IteratorAVL.hpp.

```
00151     {
00152         if (path.empty() && other.path.empty())
00153             return true;
00154 
00155         if (path.empty() || other.path.empty())
00156             return false;
00157 
00158         return path.top() == other.path.top();
00159     }
```

Grafo de chamadas desta função:



A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/avl_tree/IteratorAVL.hpp

5.10 Referência à classe Template IteratorRB< Key, Value >

Classe de iterador para a Red-Black Tree.
`#include <IteratorRB.hpp>`

Tipos Públicos

- using iterator_category = std::input_iterator_tag

  Categoria do iterador, indica que é um iterador de entrada.
- using value_type = std::pair<Key, Value>

  O tipo do valor apontado pelo iterador (um par chave-valor).
- using difference_type = std::ptrdiff_t

  Tipo para representar a diferença entre dois iteradores.
- using pointer = value_type *

  Ponteiro para o tipo de valor.
- using reference = value_type &

  Referência para o tipo de valor.

- using const_pointer = const value_type *

  Ponteiro constante para o tipo de valor.
- using const_reference = const value_type &

  Referência constante para o tipo de valor.
- using NodeType = NodeRB<Key, Value>

  Tipo do nó da árvore Red-Black Tree.
- using NodePtrType = NodePtr

  Tipo do ponteiro para o nó da árvore Red-Black Tree.

Membros públicos

- IteratorRB ()=default

  Construtor padrão.
- IteratorRB (NodePtr root, NodePtr nil)

  Construtor que inicializa o iterador a partir da raiz da árvore.
- reference operator* () const

  Operador de derreferência.
- pointer operator-> () const

  Operador de acesso a membro.
- IteratorRB & operator++ ()

  Operador de incremento (pré-fixado).
- IteratorRB operator++ (int)

  Operador de incremento (pós-fixado).
- bool operator== (const IteratorRB &other) const

  Operador de igualdade.
- bool operator!= (const IteratorRB &other) const

  Operador de desigualdade.

5.10.1 Descrição detalhada

template<typename Key, typename Value>
class IteratorRB< Key, Value >

Classe de iterador para a Red-Black Tree.
Este iterador permite percorrer os nós da árvore Red-Black Tree em ordem crescente (in-order traversal), retornando pares chave-valor.

Parâmetros de template

Key	Tipo da chave dos nós da árvore.
Value	Tipo do valor associado às chaves.

Definido na linha 20 do ficheiro IteratorRB.hpp.

5.10.2 Documentação das definições de tipo

5.10.2.1 const_pointer

```
template<typename Key, typename Value>
using IteratorRB< Key, Value >::const_pointer = const value_type *
```

Ponteiro constante para o tipo de valor.
Definido na linha 44 do ficheiro IteratorRB.hpp.

5.10.2.2 const_reference

```
template<typename Key, typename Value>
using IteratorRB< Key, Value >::const_reference = const value_type &
```

Referência constante para o tipo de valor.
Definido na linha 46 do ficheiro IteratorRB.hpp.

5.10.2.3 difference_type

```
template<typename Key, typename Value>
using IteratorRB< Key, Value >::difference_type = std::ptrdiff_t
```

Tipo para representar a diferença entre dois iteradores.
Definido na linha 38 do ficheiro IteratorRB.hpp.

5.10.2.4 iterator_category

```
template<typename Key, typename Value>
using IteratorRB< Key, Value >::iterator_category = std::input_iterator_tag
```

Categoria do iterador, indica que é um iterador de entrada.
Definido na linha 34 do ficheiro IteratorRB.hpp.

5.10.2.5 NodePtrType

```
template<typename Key, typename Value>
using IteratorRB< Key, Value >::NodePtrType = NodePtr
```

Tipo do ponteiro para o nó da árvore Red-Black Tree.
Definido na linha 50 do ficheiro IteratorRB.hpp.

5.10.2.6 NodeType

```
template<typename Key, typename Value>
using IteratorRB< Key, Value >::NodeType = NodeRB<Key, Value>
```

Tipo do nó da árvore Red-Black Tree.
Definido na linha 48 do ficheiro IteratorRB.hpp.

5.10.2.7 pointer

```
template<typename Key, typename Value>
using IteratorRB< Key, Value >::pointer = value_type *
```

Ponteiro para o tipo de valor.
Definido na linha 40 do ficheiro IteratorRB.hpp.

5.10.2.8 reference

```
template<typename Key, typename Value>
using IteratorRB< Key, Value >::reference = value_type &
```

Referência para o tipo de valor.
Definido na linha 42 do ficheiro IteratorRB.hpp.

5.10.2.9 value_type

```
template<typename Key, typename Value>
using IteratorRB< Key, Value >::value_type = std::pair<Key, Value>
```

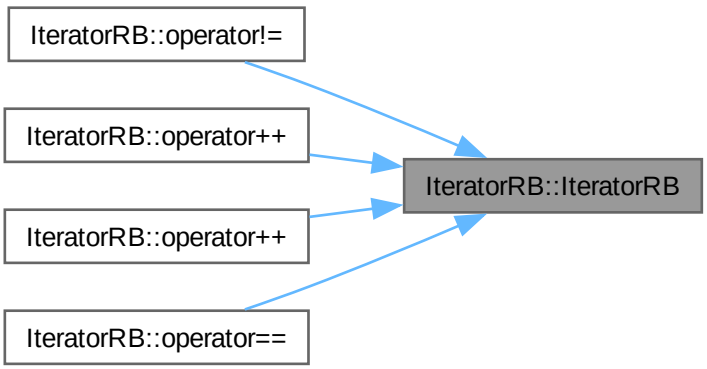
O tipo do valor apontado pelo iterador (um par chave-valor).
Definido na linha 36 do ficheiro IteratorRB.hpp.

5.10.3 Documentação dos Construtores & Destrutor

5.10.3.1 IteratorRB() [1/2]

```
template<typename Key, typename Value>
IteratorRB< Key, Value >::IteratorRB ()  [default]
```

Construtor padrão.
Cria um iterador inválido (geralmente usado para representar o fim de uma coleção). Este é o diagrama das funções que utilizam esta função:



5.10.3.2 IteratorRB() [2/2]

```
template<typename Key, typename Value>
IteratorRB< Key, Value >::IteratorRB (
            NodePtr root,
            NodePtr nil)  [inline]
```

Construtor que inicializa o iterador a partir da raiz da árvore.
O iterador é posicionado no primeiro elemento da travessia em ordem (o nó mais à esquerda).

Parâmetros

root	Ponteiro para o nó raiz da árvore Red-Black Tree.

Definido na linha 67 do ficheiro IteratorRB.hpp.

```
00067                                           : nil(nil)
00068     {
00069         NodePtr current = root;
00070         while (current != nil)
00071         {
00072             path.push(current);
00073             current = current->left;
00074         }
00075     }
```

5.10.4 Documentação das funções

5.10.4.1 operator"!=()

```
template<typename Key, typename Value>
bool IteratorRB< Key, Value >::operator!= (
            const IteratorRB< Key, Value > & other) const  [inline]
```

Operador de desigualdade.
Compara este iterador com outro iterador para verificar se apontam para nós diferentes.

Parâmetros

other	O outro iterador a ser comparado.

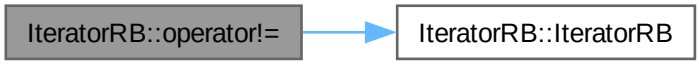
Retorna

`true` se os iteradores são diferentes, `false` caso contrário.

Definido na linha 174 do ficheiro IteratorRB.hpp.

```
00175     {
00176         return !(*this == other);
00177     }
```

Grafo de chamadas desta função:



5.10.4.2 operator∗()

```
template<typename Key, typename Value>
reference IteratorRB< Key, Value >::operator* () const  [inline]
```

Operador de derreferência.
Retorna uma referência para o par chave-valor do nó atual.

Retorna

Referência para o par chave-valor do nó atual.

Definido na linha 84 do ficheiro IteratorRB.hpp.

```
00085     {
00086         return path.top()->key;
00087     }
```

5.10.4.3 operator++() [1/2]

```
template<typename Key, typename Value>
IteratorRB & IteratorRB< Key, Value >::operator++ ()  [inline]
```

Operador de incremento (pré-fixado).
Avança o iterador para o próximo nó na travessia em ordem.

Retorna

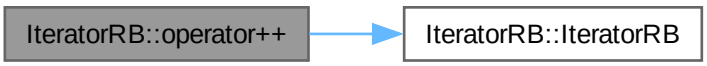
Referência para o iterador atualizado.

Definido na linha 108 do ficheiro IteratorRB.hpp.

```
00109     {
00110         if (path.empty())
00111             return *this;
00112 
00113         NodePtr node = path.top();
00114         path.pop();
00115 
00116         if (node->right != nil)
00117         {
```

```
00118             NodePtr current = node->right;
00119
00120             while (current != nil)
00121             {
00122                 path.push(current);
00123                 current = current->left;
00124             }
00125         }
00126
00127         return *(this);
00128     }
```

Grafo de chamadas desta função:



5.10.4.4 operator++() [2/2]

```
template<typename Key, typename Value>
IteratorRB IteratorRB< Key, Value >::operator++ (
            int )  [inline]
```

Operador de incremento (pós-fixado).
Avança o iterador para o próximo nó na travessia em ordem e retorna uma cópia do iterador antes do incremento.

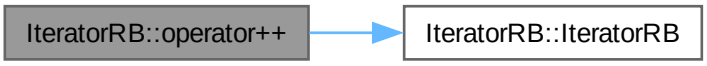
Retorna

Uma cópia do iterador antes do incremento.

Definido na linha 138 do ficheiro IteratorRB.hpp.

```
00139     {
00140         IteratorRB temp = *this;
00141         ++(*this);
00142         return temp;
00143     }
```

Grafo de chamadas desta função:



5.10.4.5 operator->()

```
template<typename Key, typename Value>
pointer IteratorRB< Key, Value >::operator-> () const  [inline]
```

Operador de acesso a membro.
Retorna um ponteiro para o par chave-valor do nó atual.

Retorna

Ponteiro para o par chave-valor do nó atual.

Definido na linha 96 do ficheiro IteratorRB.hpp.

```
00097     {
00098         return &(path.top()->key);
00099     }
```

5.10.4.6 operator==()

```
template<typename Key, typename Value>
bool IteratorRB< Key, Value >::operator== (
             const IteratorRB< Key, Value > & other) const  [inline]
```

Operador de igualdade.
Compara este iterador com outro iterador para verificar se apontam para o mesmo nó.

Parâmetros

other	O outro iterador a ser comparado.

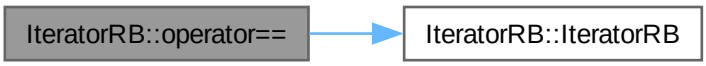
Retorna

`true` se os iteradores são iguais, `false` caso contrário.

Definido na linha 154 do ficheiro IteratorRB.hpp.

```
00155     {
00156         if (path.empty() && other.path.empty())
00157             return true;
00158
00159         if (path.empty() || other.path.empty())
00160             return false;
00161
00162         return path.top() == other.path.top();
00163     }
```

Grafo de chamadas desta função:



A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/rb_tree/IteratorRB.hpp

5.11 Referência à estrutura Template Node< Key, Value >

Estrutura que representa um nó em uma árvore binária, comumente utilizada em árvores AVL.
`#include <Node.hpp>`

Membros públicos

- Node (const std::pair< Key, Value > &key, const int &height=1, Node< Key, Value > *left=nullptr, Node< Key, Value > *right=nullptr)

Atributos Públicos

- std::pair< Key, Value > key
- int height
- Node< Key, Value > * left
- Node< Key, Value > * right

5.11.1 Descrição detalhada

template<typename Key, typename Value>
struct Node< Key, Value >

Estrutura que representa um nó em uma árvore binária, comumente utilizada em árvores AVL.
Esta estrutura genérica (template) `Node` é projetada para encapsular os dados de um nó individual dentro de uma estrutura de árvore. Cada nó contém uma chave (o valor armazenado), a altura do nó na árvore (fundamental para algoritmos de balanceamento, como os empregados em árvores AVL), e ponteiros para seus nós filhos, esquerdo e direito.

Parâmetros de template

Key	O tipo de dado da chave a ser armazenada no nó. Este tipo deve suportar operações de comparação se o nó for utilizado em árvores de busca ordenadas.
Value	O tipo de dado do valor associado à chave. Este tipo pode se qualquer tipo que seja necessário armazenar junto com a chave, como um valor de dados, uma estrutura ou um objeto.

5.11.1.0.1 Membros:

- `key` (do tipo `std::pair<Key, Value>`): Armazena o valor principal ou a chave de identificação do nó. É um par
- `height` (do tipo `int`): Representa a altura do nó dentro da árvore. A altura é definida como a maior distância (número de arestas) deste nó até uma folha em sua subárvore. Por convenção, um nó folha tem altura 1. A altura de um subárvore vazia (representada por um ponteiro `nullptr`) é frequentemente considerada 0 para simplificar os cálculos de balanceamento.
- `left` (ponteiro para `Node<Key, Value>`): Aponta para o nó filho à esquerda. Se o nó não possuir um filho esquerdo, este ponteiro será `nullptr`.
- `right` (ponteiro para `Node<Key, Value>`): Aponta para o nó filho à direita. Se o nó não possuir um filho direito, este ponteiro será `nullptr`.

5.11.1.0.2 Construtor:

`Node(const std::pair<Key, Value> &key, const int &height = 1, Node<Key, Value> *left = nullptr, Node<Key, Value> *right = nullptr)`
Constrói uma nova instância de `Node`.

Parâmetros

key	Referência constante para a chave que será armazenada no nó.
height	(Opcional) Valor inteiro para a altura inicial do nó. O valor padrão é `1`, assumindo que um novo nó é, inicialmente, uma folha.
left	(Opcional) Ponteiro para o nó filho esquerdo. O valor padrão é `nullptr`.
right	(Opcional) Ponteiro para o nó filho direito. O valor padrão é `nullptr`.

O construtor utiliza uma lista de inicialização de membros para definir os valores `key`, `height`, `left` e `right` com os parâmetros fornecidos.
Definido na linha 61 do ficheiro Node.hpp.

5.11.2 Documentação dos Construtores & Destrutor

5.11.2.1 Node()

```
template<typename Key, typename Value>
Node< Key, Value >::Node (
            const std::pair< Key, Value > & key,
            const int & height = 1,
            Node< Key, Value > * left = nullptr,
            Node< Key, Value > * right = nullptr)  [inline]
```

Definido na linha 68 do ficheiro Node.hpp.

```
00069         : key(key), height(height), left(left), right(right) {}
```

5.11.3 Documentação dos dados membro

5.11.3.1 height

```
template<typename Key, typename Value>
int Node< Key, Value >::height
```

Definido na linha 64 do ficheiro Node.hpp.

5.11.3.2 key

```
template<typename Key, typename Value>
std::pair<Key, Value> Node< Key, Value >::key
```

Definido na linha 63 do ficheiro Node.hpp.

5.11.3.3 left

```
template<typename Key, typename Value>
Node<Key, Value>* Node< Key, Value >::left
```

Definido na linha 65 do ficheiro Node.hpp.

5.11.3.4 right

```
template<typename Key, typename Value>
Node<Key, Value>* Node< Key, Value >::right
```

Definido na linha 66 do ficheiro Node.hpp.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- include/dictionary/avl_tree/Node.hpp

5.12 Referência à estrutura Template NodeRB< Key, Value >

Estrutura que representa um nó em uma Árvore Rubro-Negra (Red-Black Tree).

`#include <NodeRB.hpp>`

Membros públicos

- NodeRB (const std::pair< Key, Value > &key, const bool &color, NodeRB< Key, Value > *parent, NodeRB< Key, Value > *left, NodeRB< Key, Value > *right)

  Construtor para um nó da Árvore Rubro-Negra.

Atributos Públicos

- std::pair< Key, Value > key

  Par chave-valor armazenado no nó.

- bool color {RED}

  Cor do nó (VERMELHO ou PRETO). Utiliza as constantes `RED` ou `BLACK`.

- NodeRB< Key, Value > * parent {nullptr}

  Ponteiro para o nó pai. Inicializado como `nullptr` por padrão.
- NodeRB< Key, Value > * left {nullptr}

  Ponteiro para o filho esquerdo. Inicializado como `nullptr` por padrão.
- NodeRB< Key, Value > * right {nullptr}

  Ponteiro para o filho direito. Inicializado como `nullptr` por padrão.

Atributos Públicos Estáticos

- static constexpr bool BLACK = false

  Constante estática que representa a cor PRETA para um nó. O valor é `false`.
- static constexpr bool RED = true

  Constante estática que representa a cor VERMELHA para um nó. O valor é `true`.

5.12.1 Descrição detalhada

template<typename Key, typename Value>
struct NodeRB< Key, Value >

Estrutura que representa um nó em uma Árvore Rubro-Negra (Red-Black Tree).
Esta estrutura armazena um par chave-valor, a cor do nó (VERMELHO ou PRETO), e ponteiros para o nó pai, filho esquerdo e filho direito.

Parâmetros de template

Key	Tipo da chave armazenada no nó.
Value	Tipo do valor associado à chave.

Nota

A cor do nó é representada por um booleano, onde `true` indica VERMELHO e `false` indica PRETO. As constantes estáticas `RED` e `BLACK` são usadas para melhorar a legibilidade do código.

Esta estrutura é utilizada internamente na implementação de uma Árvore Rubro-Negra, que é uma estrutura de dados de árvore balanceada. A cor dos nós é usada para garantir que a árvore mantenha suas propriedades de balanceamento, o que permite operações de inserção, remoção e busca eficientes.
Definido na linha 25 do ficheiro NodeRB.hpp.

5.12.2 Documentação dos Construtores & Destrutor

5.12.2.1 NodeRB()

```
template<typename Key, typename Value>
NodeRB< Key, Value >::NodeRB (
            const std::pair< Key, Value > & key,
            const bool & color,
            NodeRB< Key, Value > * parent,
            NodeRB< Key, Value > * left,
            NodeRB< Key, Value > * right)  [inline]
```

Construtor para um nó da Árvore Rubro-Negra.

Parâmetros

key	O par chave-valor a ser armazenado no nó.
color	A cor do nó. O padrão é `RED`.
parent	Ponteiro para o nó pai.
left	Ponteiro para o filho esquerdo.

right	Ponteiro para o filho direito.

Definido na linha 81 do ficheiro NodeRB.hpp.

```
00082         : key(key), color(color), parent(parent), left(left), right(right) {}
```

5.12.3 Documentação dos dados membro

5.12.3.1 BLACK

```
template<typename Key, typename Value>
bool NodeRB< Key, Value >::BLACK = false  [static], [constexpr]
```

Constante estática que representa a cor PRETA para um nó. O valor é `false`.
Definido na linha 31 do ficheiro NodeRB.hpp.

5.12.3.2 color

```
template<typename Key, typename Value>
bool NodeRB< Key, Value >::color {RED}
```

Cor do nó (VERMELHO ou PRETO). Utiliza as constantes `RED` ou `BLACK`.
A cor é usada para manter as propriedades de balanceamento da Árvore Rubro-Negra. Por padrão, os nós são inicializados como VERMELHOS (`RED`).
Definido na linha 52 do ficheiro NodeRB.hpp.

```
00052 {RED};
```

5.12.3.3 key

```
template<typename Key, typename Value>
std::pair<Key, Value> NodeRB< Key, Value >::key
```

Par chave-valor armazenado no nó.
Definido na linha 42 do ficheiro NodeRB.hpp.

5.12.3.4 left

```
template<typename Key, typename Value>
NodeRB<Key, Value>* NodeRB< Key, Value >::left {nullptr}
```

Ponteiro para o filho esquerdo. Inicializado como `nullptr` por padrão.
Definido na linha 64 do ficheiro NodeRB.hpp.

```
00064 {nullptr};
```

5.12.3.5 parent

```
template<typename Key, typename Value>
NodeRB<Key, Value>* NodeRB< Key, Value >::parent {nullptr}
```

Ponteiro para o nó pai. Inicializado como `nullptr` por padrão.
Definido na linha 58 do ficheiro NodeRB.hpp.

```
00058 {nullptr};
```

5.12.3.6 RED

```
template<typename Key, typename Value>
bool NodeRB< Key, Value >::RED = true  [static], [constexpr]
```

Constante estática que representa a cor VERMELHA para um nó. O valor é `true`.
Definido na linha 37 do ficheiro NodeRB.hpp.

5.12.3.7 right

```
template<typename Key, typename Value>
NodeRB<Key, Value>* NodeRB< Key, Value >::right {nullptr}
```

Ponteiro para o filho direito. Inicializado como `nullptr` por padrão.
Definido na linha 70 do ficheiro NodeRB.hpp.

```
00070 {nullptr};
```

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- include/dictionary/rb_tree/NodeRB.hpp

5.13 Referência à classe Template OpenHashTable< Key, Value, Hash >

`#include <OpenHashTable.hpp>`
Diagrama de heranças da classe OpenHashTable< Key, Value, Hash >

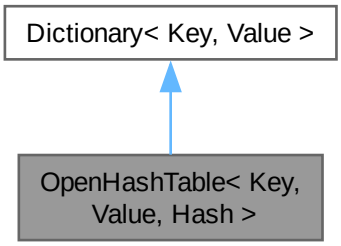
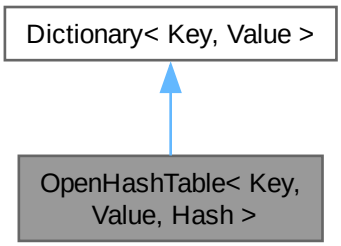


Diagrama de colaboração para OpenHashTable< Key, Value, Hash >:



Membros públicos

- OpenHashTable (const size_t &tableSize=19, const float &load_factor=0.5f)

  Construtor padrão. Cria uma tabela hash vazia.
- OpenHashTable (const std::initializer_list< std::pair< Key, Value > > &list, const size_t &tableSize=19, const float &load_factor=0.5f)

  Construtor que inicializa a tabela com uma lista de elementos.
- std::unique_ptr< Dictionary< Key, Value > > clone () const

  Cria e retorna uma cópia profunda (deep copy) da tabela hash.
- long long getComparisons () const noexcept

  Retorna o número de comparações realizadas durante as operações.

- long long getCollisions () const noexcept

  Retorna o número de colisões ocorridas durante as operações.
- size_t size () const noexcept

  Retorna o número de pares chave-valor na tabela.
- bool empty () const noexcept

  Verifica se a tabela está vazia.
- size_t bucket_count () const noexcept

  Retorna o número de "buckets" (slots) na tabela hash.
- size_t bucket (const Key &k) const

  Retorna o índice do "bucket" onde um elemento com a chave `k` seria armazenado.
- void clear ()

  Remove todos os elementos da tabela, deixando-a com tamanho 0.
- float load_factor () const noexcept

  Retorna o fator de carga atual da tabela. O fator de carga é a razão entre o número de elementos e o número de "buckets".
- float max_load_factor () const noexcept

  Retorna o fator de carga máximo permitido. Se `load_factor()` exceder este valor, um rehash é acionado.
- ~OpenHashTable ()=default

  Destrutor. Libera todos os recursos.
- void insert (const std::pair< Key, Value > &key_value)

  Insere um novo par chave-valor na tabela.
- void update (const std::pair< Key, Value > &key_value)

  Atualiza o valor associado a uma chave existente.
- bool contains (const Key &k)

  Verifica se a tabela contém um elemento com a chave especificada.
- Value & at (const Key &k)

  Acessa o valor associado a uma chave.
- const Value & at (const Key &k) const

  Acessa o valor associado a uma chave (versão const).
- void rehash (size_t m)

  Redimensiona a tabela hash para um novo tamanho.
- void remove (const Key &k)

  Remove um elemento da tabela pela chave.
- void reserve (size_t n) noexcept

  Reserva espaço para pelo menos `n` elementos.
- void set_max_load_factor (float lf)

  Define o fator de carga máximo.
- Value & operator[ ] (const Key &k)

  Acessa ou insere um elemento.
- const Value & operator[ ] (const Key &k) const

  Acessa um elemento (versão const).
- void print () const

  Imprime o conteúdo da tabela no formato [chave1:valor1, chave2:valor2, ...].
- void forEach (const std::function< void(const std::pair< Key, Value > &)> &func) const

  Aplica uma função a cada par chave-valor na tabela.

Membros públicos herdados de Dictionary< Key, Value >

- virtual ~Dictionary ()=default

  Destrutor virtual para permitir a destruição correta de classes derivadas.

5.13.1 Descrição detalhada

template<typename Key, typename Value, typename Hash = std::hash<Key>>
class OpenHashTable< Key, Value, Hash >

Definido na linha 32 do ficheiro OpenHashTable.hpp.

5.13.2 Documentação dos Construtores & Destrutor

5.13.2.1 OpenHashTable() [1/2]

```
template<typename Key, typename Value, typename Hash>
OpenHashTable< Key, Value, Hash >::OpenHashTable (
            const size_t & tableSize = 19,
            const float & load_factor = 0.5f)
```

Construtor padrão. Cria uma tabela hash vazia.

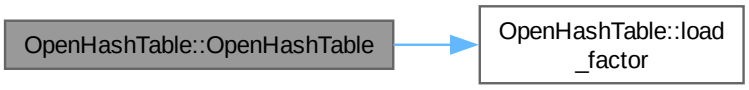
Parâmetros

tableSize	O número inicial de "buckets" (slots) na tabela. Será ajustado para o próximo número primo maior ou igual.
load_factor	O fator de carga máximo permitido antes de um rehash.

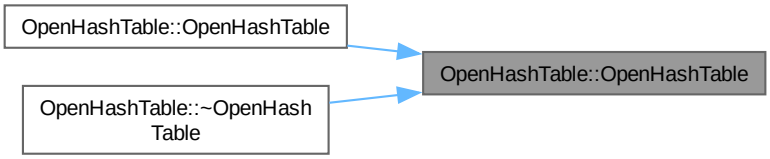
Definido na linha 372 do ficheiro OpenHashTable.hpp.

```
00372                                                                                                    :
      m_number_of_elements(0), m_table_size(tableSize)
00373 {
00374     m_table.resize(m_table_size);
00375
00376     if (load_factor <= 0)
00377         m_max_load_factor = 0.5f;
00378     else
00379         m_max_load_factor = load_factor;
00380 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



5.13.2.2 OpenHashTable() [2/2]

```
template<typename Key, typename Value, typename Hash>
OpenHashTable< Key, Value, Hash >::OpenHashTable (
            const std::initializer_list< std::pair< Key, Value > > & list,
            const size_t & tableSize = 19,
            const float & load_factor = 0.5f)
```

Construtor que inicializa a tabela com uma lista de elementos.

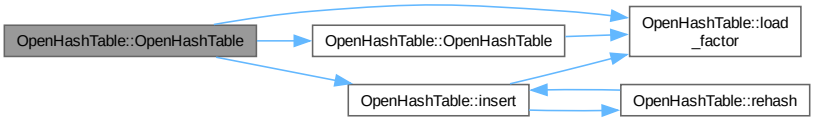
Parâmetros

list	Uma `std::initializer_list` de pares chave-valor para inserir na tabela.
tableSize	O número inicial de "buckets" na tabela.
load_factor	O fator de carga máximo.

Definido na linha 383 do ficheiro OpenHashTable.hpp.

```
00383
      : OpenHashTable(tableSize, load_factor)
00384 {
00385     for (const auto &pair : list)
00386         insert(pair);
00387 }
```

Grafo de chamadas desta função:

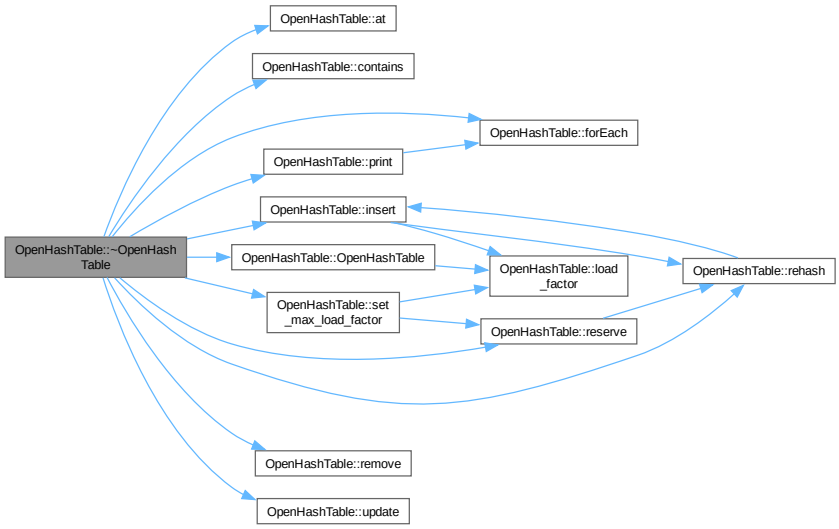


5.13.2.3 ~OpenHashTable()

```
template<typename Key, typename Value, typename Hash = std::hash<Key>>
OpenHashTable< Key, Value, Hash >::~OpenHashTable ()  [default]
```

Destrutor. Libera todos os recursos.

Grafo de chamadas desta função:



5.13.3 Documentação das funções

5.13.3.1 at() [1/2]

```
template<typename Key, typename Value, typename Hash>
Value & OpenHashTable< Key, Value, Hash >::at (
            const Key & k)  [virtual]
```

Acessa o valor associado a uma chave.
Retorna uma referência ao valor correspondente à chave `k`.

Parâmetros

k	A chave do elemento a ser acessado.

Retorna

Value& Uma referência ao valor.

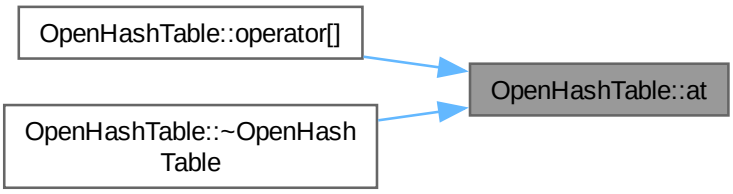
Excepções

std::out_of_range	se a chave `k` não for encontrada na tabela.

Implementa Dictionary< Key, Value >.
Definido na linha 523 do ficheiro OpenHashTable.hpp.

```
00524 {
00525     size_t hash_index = findIndex(k);
00526
00527     if (hash_index != (size_t)-1)
00528         return m_table[hash_index].data.second;
00529     else
00530         throw std::out_of_range("Key not found in the hash table");
00531 }
```

Este é o diagrama das funções que utilizam esta função:



5.13.3.2 at() [2/2]

```
template<typename Key, typename Value, typename Hash>
const Value & OpenHashTable< Key, Value, Hash >::at (
            const Key & k) const
```

Acessa o valor associado a uma chave (versão const).
Retorna uma referência constante ao valor correspondente à chave `k`.

Parâmetros

k	A chave do elemento a ser acessado.

Retorna

const Value& Uma referência constante ao valor.

Excepções

std::out_of_range	se a chave `k` não for encontrada na tabela.

Definido na linha 534 do ficheiro OpenHashTable.hpp.

```
00535 {
00536     size_t hash_index = findIndex(k);
00537
00538     if (hash_index != (size_t)-1)
00539         return m_table[hash_index].data.second;
00540     else
00541         throw std::out_of_range("Key not found in the hash table");
00542 }
```

5.13.3.3 bucket()

```
template<typename Key, typename Value, typename Hash>
size_t OpenHashTable< Key, Value, Hash >::bucket (
            const Key & k) const
```

Retorna o índice do "bucket" onde um elemento com a chave `k` seria armazenado.

Parâmetros

k	A chave a ser localizada.

Retorna

size_t O índice do "bucket" correspondente.

Definido na linha 414 do ficheiro OpenHashTable.hpp.

```
00415 {
00416     return hash_code(k);
00417 }
```

5.13.3.4 bucket_count()

```
template<typename Key, typename Value, typename Hash>
size_t OpenHashTable< Key, Value, Hash >::bucket_count () const  [noexcept]
```

Retorna o número de "buckets" (slots) na tabela hash.

Retorna

size_t O tamanho da tabela interna (número de listas de encadeamento).

Definido na linha 408 do ficheiro OpenHashTable.hpp.

```
00409 {
00410     return m_table_size;
00411 }
```

5.13.3.5 clear()

```
template<typename Key, typename Value, typename Hash>
void OpenHashTable< Key, Value, Hash >::clear ()  [virtual]
```

Remove todos os elementos da tabela, deixando-a com tamanho 0.

Implementa Dictionary< Key, Value >.

Definido na linha 432 do ficheiro OpenHashTable.hpp.

```
00433 {
00434     m_table.clear();
00435     m_table.resize(m_table_size);
00436
00437     m_number_of_elements = 0;
00438 }
```

5.13.3.6 clone()

```
template<typename Key, typename Value, typename Hash>
std::unique_ptr< Dictionary< Key, Value > > OpenHashTable< Key, Value, Hash >::clone ()
const  [virtual]
```

Cria e retorna uma cópia profunda (deep copy) da tabela hash.

Retorna

std::unique_ptr<Dictionary<Key, Value>> Um ponteiro para a nova instância clonada.

Implementa Dictionary< Key, Value >.

Definido na linha 390 do ficheiro OpenHashTable.hpp.

```
00391 {
00392     return std::make_unique<OpenHashTable<Key, Value, Hash>>(*this);
00393 }
```

5.13.3.7 contains()

```
template<typename Key, typename Value, typename Hash>
bool OpenHashTable< Key, Value, Hash >::contains (
            const Key & k)  [virtual]
```

Verifica se a tabela contém um elemento com a chave especificada.

Parâmetros

k	A chave a ser procurada.

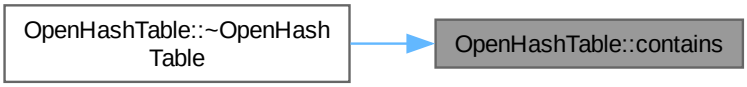
Retorna

true se um elemento com a chave `k` existir, false caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 517 do ficheiro OpenHashTable.hpp.

```
00518 {
00519     return findIndex(k) != (size_t)-1;
00520 }
```

Este é o diagrama das funções que utilizam esta função:



5.13.3.8 empty()

```
template<typename Key, typename Value, typename Hash>
bool OpenHashTable< Key, Value, Hash >::empty () const  [virtual], [noexcept]
```

Verifica se a tabela está vazia.

Retorna

true se a tabela não contém elementos, false caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 402 do ficheiro OpenHashTable.hpp.

```
00403 {
00404     return m_number_of_elements == 0;
00405 }
```

5.13.3.9 forEach()

```
template<typename Key, typename Value, typename Hash>
void OpenHashTable< Key, Value, Hash >::forEach (
            const std::function< void(const std::pair< Key, Value > &)> & func) const  [virtual]
```

Aplica uma função a cada par chave-valor na tabela.
Itera sobre todos os elementos da tabela e executa a função `func` para cada um. A ordem de iteração não é garantida.

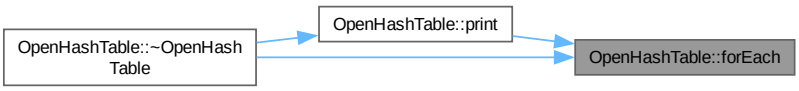
Parâmetros

func	A função a ser aplicada. Deve aceitar um `const std::pair<Key, Value>&`.

Implementa Dictionary< Key, Value >.
Definido na linha 624 do ficheiro OpenHashTable.hpp.

```
00625 {
00626     for (const auto &slot : m_table)
00627         if (slot.is_active()) // verifica se o slot esta ativo
00628             func(slot.data);  // aplica a funcao a cada par chave-valor
00629 }
```

Este é o diagrama das funções que utilizam esta função:



5.13.3.10 getCollisions()

```
template<typename Key, typename Value, typename Hash = std::hash<Key>>
long long OpenHashTable< Key, Value, Hash >::getCollisions () const  [inline], [noexcept]
```

Retorna o número de colisões ocorridas durante as operações.
Este método é útil para análise de desempenho, permitindo verificar quantas colisões ocorreram ao longo das operações de inserção.

Retorna

long long O número total de colisões ocorridas.

Definido na linha 143 do ficheiro OpenHashTable.hpp.

```
00143 { return collisions; }
```

5.13.3.11 getComparisons()

```
template<typename Key, typename Value, typename Hash = std::hash<Key>>
long long OpenHashTable< Key, Value, Hash >::getComparisons () const  [inline], [noexcept]
```

Retorna o número de comparações realizadas durante as operações.
Este método é útil para análise de desempenho, permitindo verificar quantas comparações foram feitas ao longo das operações de inserção, busca e remoção.

Retorna

long long O número total de comparações realizadas.

Definido na linha 133 do ficheiro OpenHashTable.hpp.

```
00133 { return comparisons; }
```

5.13.3.12 insert()

```
template<typename Key, typename Value, typename Hash>
void OpenHashTable< Key, Value, Hash >::insert (
            const std::pair< Key, Value > & key_value)  [virtual]
```

Insere um novo par chave-valor na tabela.
A inserção só ocorre se a chave ainda não existir na tabela. Se a inserção fizer com que o fator de carga exceda o `max_load_factor`, um rehash é executado para aumentar o tamanho da tabela.

Parâmetros

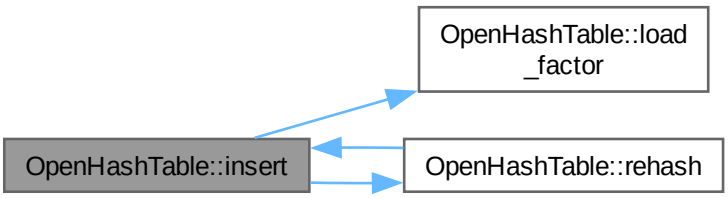
key_value	O par `std::pair<Key, Value>` a ser inserido.

Implementa Dictionary< Key, Value >.
Definido na linha 441 do ficheiro OpenHashTable.hpp.

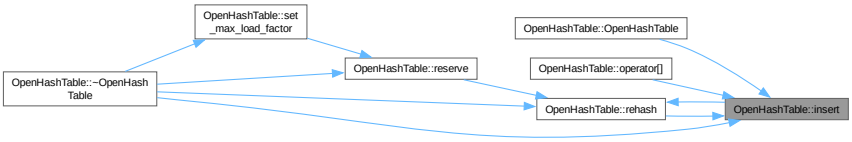
```
00442 {
00443     if (load_factor() >= m_max_load_factor)
00444         rehash(m_table_size * 2);
00445
00446     size_t hash_index{(size_t)-1};
00447     size_t first_deleted_index{(size_t)-1};
```

```
00448
00449     for (size_t i = 0; i < m_table_size; i++)
00450     {
00451         size_t current_index = hash_code(key_value.first, i);
00452
00453         if (m_table[current_index].is_empty())
00454         {
00455             hash_index = current_index;
00456             break;
00457         }
00458         else if (m_table[current_index].is_active())
00459         {
00460             comparisons++;
00461             if (m_table[current_index].data.first == key_value.first)
00462                 return;
00463             collisions++;
00464         }
00465         else
00466         {
00467             if (first_deleted_index == (size_t)-1)
00468                 first_deleted_index = current_index;
00469         }
00470     }
00471
00472     if (first_deleted_index != (size_t)-1)
00473         hash_index = first_deleted_index;
00474
00475     if (hash_index == (size_t)-1)
00476         throw std::out_of_range("Hash table is full, cannot insert new element");
00477
00478     m_table[hash_index] = Slot<Key, Value>(key_value);
00479     m_number_of_elements++;
00480 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



5.13.3.13 load_factor()

```
template<typename Key, typename Value, typename Hash>
float OpenHashTable< Key, Value, Hash >::load_factor () const  [noexcept]
```

Retorna o fator de carga atual da tabela. O fator de carga é a razão entre o número de elementos e o número de "buckets".

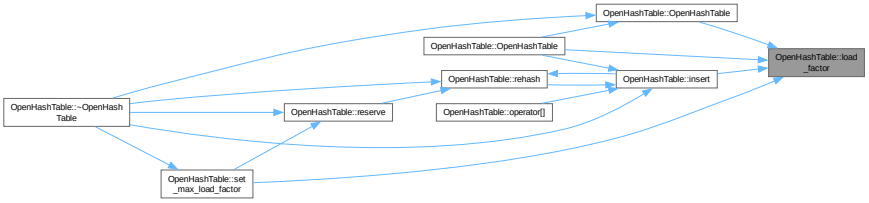
Retorna

float O fator de carga atual.

Definido na linha 420 do ficheiro OpenHashTable.hpp.

```
00421 {
00422     return static_cast<float>(m_number_of_elements) / m_table_size;
00423 }
```

Este é o diagrama das funções que utilizam esta função:



5.13.3.14 max_load_factor()

```
template<typename Key, typename Value, typename Hash>
float OpenHashTable< Key, Value, Hash >::max_load_factor () const  [noexcept]
```

Retorna o fator de carga máximo permitido. Se `load_factor()` exceder este valor, um rehash é acionado.

Retorna

float O fator de carga máximo.

Definido na linha 426 do ficheiro OpenHashTable.hpp.

```
00427 {
00428     return m_max_load_factor;
00429 }
```

5.13.3.15 operator[]() [1/2]

```
template<typename Key, typename Value, typename Hash>
Value & OpenHashTable< Key, Value, Hash >::operator[] (
            const Key & k)  [virtual]
```

Acessa ou insere um elemento.

Se a chave `k` existir, retorna uma referência ao seu valor. Se não existir, insere um novo elemento com a chave `k` (usando o construtor padrão de `Value`) e retorna uma referência ao novo valor.

Parâmetros

k	A chave do elemento a ser acessado ou inserido.

Retorna

Value& Uma referência ao valor do elemento.

Implementa Dictionary< Key, Value >.

Definido na linha 598 do ficheiro OpenHashTable.hpp.

```
00599 {
00600     size_t hash_index = findIndex(k);
00601
00602     if (hash_index != (size_t)-1)
00603         return m_table[hash_index].data.second;
00604     else
00605         insert({k, Value{}}); // insere um novo elemento com valor padrão
00606
00607     return m_table[findIndex(k)].data.second; // retorna o valor associado a chave
00608 }
```

Grafo de chamadas desta função:

5.13.3.16 operator[]() [2/2]

```
template<typename Key, typename Value, typename Hash>
const Value & OpenHashTable< Key, Value, Hash >::operator[] (
            const Key & k) const
```

Acessa um elemento (versão const).
Se a chave `k` existir, retorna uma referência constante ao seu valor.

Parâmetros

k	A chave do elemento a ser acessado.

Retorna

const Value& Uma referência constante ao valor do elemento.

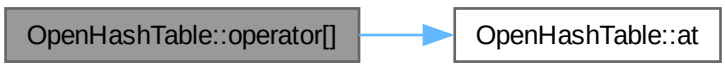
Excepções

std::out_of_range	se a chave `k` não for encontrada.

Definido na linha 611 do ficheiro OpenHashTable.hpp.

```
00612 {
00613     return at(k); // chama a funcao at para obter o valor associado a chave
00614 }
```

Grafo de chamadas desta função:



5.13.3.17 print()

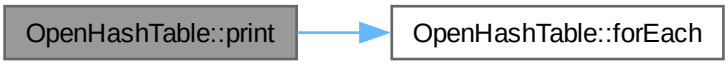
```
template<typename Key, typename Value, typename Hash>
void OpenHashTable< Key, Value, Hash >::print () const  [virtual]
```

Imprime o conteúdo da tabela no formato [chave1:valor1, chave2:valor2, ...].
Útil para depuração. A ordem dos elementos não é garantida.
Implementa Dictionary< Key, Value >.

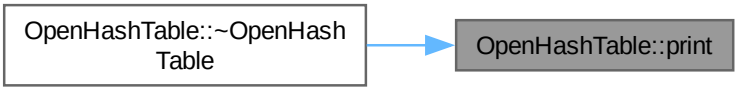
Definido na linha 617 do ficheiro OpenHashTable.hpp.

```
00618 {
00619     forEach([](const std::pair<Key, Value> &par)
00620             { std::cout « "[" « par.first « ", " « par.second « "]" « std::endl; });
00621 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



5.13.3.18 rehash()

```
template<typename Key, typename Value, typename Hash>
void OpenHashTable< Key, Value, Hash >::rehash (
            size_t m)
```

Redimensiona a tabela hash para um novo tamanho.
O tamanho da tabela é ajustado para o próximo número primo maior ou igual a `m`. Se `m` for menor que o tamanho atual, a tabela não é redimensionada. Se `m` for maior, a tabela é redimensionada e todos os elementos existentes são re-hashados para o novo tamanho.

Parâmetros

m	O novo tamanho desejado para a tabela.

Nota

O tamanho da tabela deve ser um número primo para melhor distribuição dos elementos e evitar colisões.

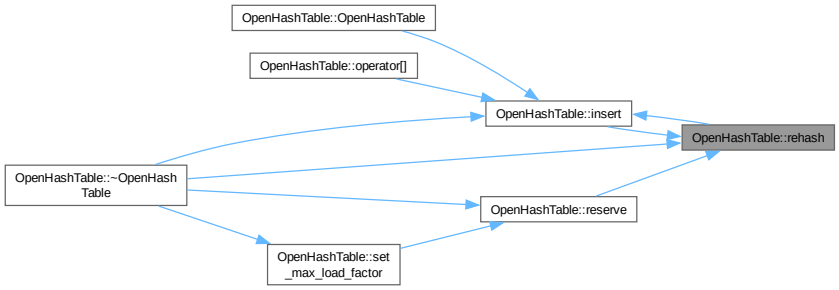
Definido na linha 545 do ficheiro OpenHashTable.hpp.

```
00546 {
00547     size_t new_table_size = get_next_prime(m);
00548
00549     if (new_table_size > m_table_size)
00550     {
00551         std::vector<Slot<Key, Value>> aux;
00552         m_table.swap(aux);
00553         m_table.resize(new_table_size);
00554
00555         m_table_size = new_table_size;
00556         m_number_of_elements = 0;
00557
00558         for (auto &slot : aux)
00559             if (slot.is_active())
```

```
00560                 insert({slot.data.first, slot.data.second});
00561     }
00562 }
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:



5.13.3.19 remove()

```
template<typename Key, typename Value, typename Hash>
void OpenHashTable< Key, Value, Hash >::remove (
            const Key & k)  [virtual]
```

Remove um elemento da tabela pela chave.
Se um elemento com a chave `k` existir, ele é removido da tabela e o número de elementos é decrementado. Se a chave não for encontrada, a função não realiza nenhuma operação.

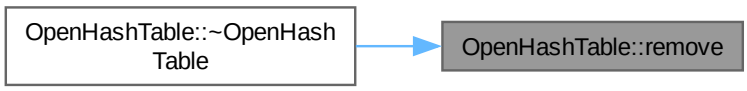
Parâmetros

k	A chave do elemento a ser removido.

Implementa Dictionary< Key, Value >.
Definido na linha 565 do ficheiro OpenHashTable.hpp.

```
00566 {
00567     size_t slot = findIndex(k); // calcula o slot em que estaria a chave
00568
00569     if (slot != (size_t)-1)
00570     {
00571         m_number_of_elements--;
00572         m_table[slot].status = HashTableStatus::DELETED;
00573     }
00574 }
```

Este é o diagrama das funções que utilizam esta função:



5.13.3.20 reserve()

```
template<typename Key, typename Value, typename Hash>
void OpenHashTable< Key, Value, Hash >::reserve (
            size_t n)  [noexcept]
```

Reserva espaço para pelo menos `n` elementos.
Se a capacidade atual não for suficiente para `n` elementos (considerando o `max_load_factor`), a tabela é redimensionada (rehash) para acomodá-los. A verificação é `n > bucket_count() * max_load_factor()`.

Parâmetros

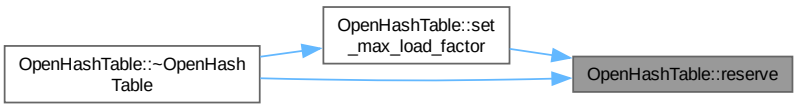
n	O número mínimo de elementos que a tabela deve ser capaz de conter.

Definido na linha 577 do ficheiro OpenHashTable.hpp.

```
00578 {
00579     if (n > m_table_size * m_max_load_factor)
00580         rehash(n / m_max_load_factor);
00581 }
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:



5.13.3.21 set_max_load_factor()

```
template<typename Key, typename Value, typename Hash>
void OpenHashTable< Key, Value, Hash >::set_max_load_factor (
            float lf)
```

Define o fator de carga máximo.

Altera o fator de carga máximo para `lf`. Após a alteração, a tabela pode ser redimensionada se o fator de carga atual exceder o novo máximo.

Parâmetros

lf	O novo valor para o fator de carga máximo (deve ser > 0).

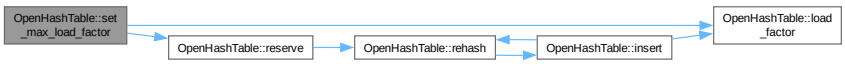
Excepções

std::out_of_range	se `lf` não for positivo.

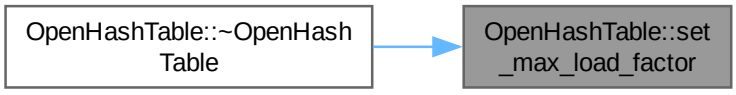
Definido na linha 584 do ficheiro OpenHashTable.hpp.

```
00585 {
00586     if (lf <= 0)
00587         throw std::out_of_range("max load factor must be greater than 0");
00588 
00589     m_max_load_factor = lf;
00590 
00591     // Se o novo fator de carga for menor que o atual,
00592     // podemos precisar redimensionar a tabela.
00593     if (load_factor() > m_max_load_factor)
00594         reserve(m_number_of_elements);
00595 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



5.13.3.22 size()

```
template<typename Key, typename Value, typename Hash>
size_t OpenHashTable< Key, Value, Hash >::size () const  [virtual], [noexcept]
```

Retorna o número de pares chave-valor na tabela.

Retorna

size_t O número de elementos.

Implementa Dictionary< Key, Value >.
Definido na linha 396 do ficheiro OpenHashTable.hpp.

```
00397 {
00398     return m_number_of_elements;
00399 }
```

5.13.3.23 update()

```
template<typename Key, typename Value, typename Hash>
void OpenHashTable< Key, Value, Hash >::update (
            const std::pair< Key, Value > & key_value)  [virtual]
```

Atualiza o valor associado a uma chave existente.
Se a chave `key_value.first` for encontrada na tabela, seu valor correspondente é atualizado para `key_value.second`. Se a chave não existir, a função não realiza nenhuma operação.

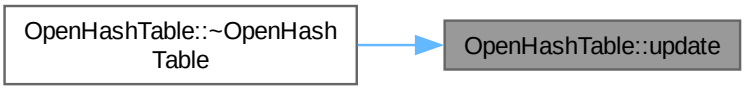
Parâmetros

key_value	O par `std::pair<Key, Value>` contendo a chave a ser encontrada e o novo valor.

Implementa Dictionary< Key, Value >.
Definido na linha 483 do ficheiro OpenHashTable.hpp.

```
00484 {
00485     size_t hash_index = findIndex(key_value.first);
00486
00487     if (hash_index != (size_t)-1)
00488         m_table[hash_index].data.second = key_value.second;
00489     else
00490         throw std::out_of_range("Key not found in the hash table");
00491 }
```

Este é o diagrama das funções que utilizam esta função:



A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/hash_table_o/OpenHashTable.hpp

5.14 Referência à classe Template RBTree< Key, Value >

5.14.1 Descrição detalhada

template<typename Key, typename Value>
class RBTree< Key, Value >

Definido na linha 8 do ficheiro IteratorRB.hpp.
A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/rb_tree/IteratorRB.hpp

5.15 Referência à classe Template RedBlackTree< Key, Value >

`#include <RedBlackTree.hpp>`

Diagrama de heranças da classe RedBlackTree< Key, Value >

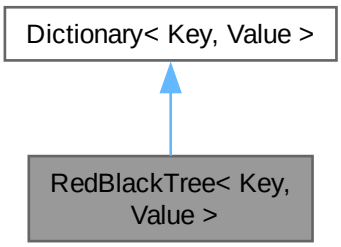
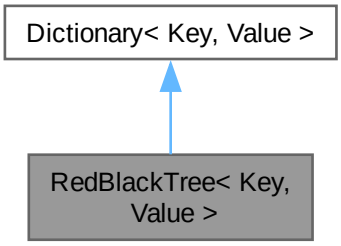


Diagrama de colaboração para RedBlackTree< Key, Value >:



Membros públicos

- RedBlackTree ()

  Construtor padrão. Cria um conjunto vazio.
- RedBlackTree (const RedBlackTree &other)

  Construtor de cópia. Cria um novo conjunto como cópia de `other`.
- RedBlackTree (std::initializer_list< std::pair< Key, Value > > list)

  Construtor a partir de uma lista inicializadora.
- std::unique_ptr< Dictionary< Key, Value > > clone () const

  Cria uma cópia profunda da árvore Rubro-Negra.
- ~RedBlackTree ()

  Destrutor. Libera toda a memória alocada pelos nós da árvore.
- iterator begin () noexcept

  Retorna um iterador para o início do conjunto.
- iterator end () noexcept

  Retorna um iterador para o final do conjunto.
- iterator begin () const noexcept

  Retorna um iterador constante para o início do conjunto.
- iterator end () const noexcept

  Retorna um iterador constante para o final do conjunto.

- iterator cbegin () const noexcept

  Retorna um iterador constante para o início do conjunto.
- iterator cend () const noexcept

  Retorna um iterador constante para o final do conjunto.
- void operator= (const RedBlackTree &other)

  Operador de atribuição por cópia.
- size_t size () const noexcept

  Retorna o número de elementos no conjunto.
- bool empty () const noexcept

  Verifica se o conjunto está vazio.
- long long getComparisons () const noexcept

  Retorna o número de comparações realizadas durante as operações.
- long long getRotations () const noexcept

  Retorna o número de rotações realizadas durante as operações.
- void clear ()

  Remove todos os elementos do conjunto.
- void swap (RedBlackTree< Key, Value > &other) noexcept

  Troca o conteúdo deste conjunto com o de `other`.
- void insert (const std::pair< Key, Value > &key)

  Insere um par chave-valor no conjunto.
- Value & at (const Key &key)

  Acessa o valor associado a uma chave, com verificação de limites.
- Value & operator[ ] (const Key &key)

  Sobrecarga do operador de indexação para acessar ou inserir um elemento.
- void update (const std::pair< Key, Value > &key)

  Atualiza o valor de uma chave existente ou insere um novo par chave-valor.
- void operator= (std::pair< Key, Value > &key)

  Operador de atribuição para atualizar ou inserir um par chave-valor.
- void remove (const Key &key)

  Remove um elemento do conjunto pela chave.
- bool contains (const Key &key)

  Verifica se o conjunto contém uma determinada chave.
- void print () const

  Imprime os elementos do conjunto em ordem crescente (travessia in-order).
- void forEach (const std::function< void(const std::pair< Key, Value > &)> &func) const

  Aplica uma função a cada par chave-valor no conjunto.
- void bshow ()

  Exibe a estrutura da Árvore Rubro-Negra de forma visual no console.

Membros públicos herdados de Dictionary< Key, Value >

- virtual ~Dictionary ()=default

  Destrutor virtual para permitir a destruição correta de classes derivadas.

Amigos

- class IteratorRB< Key, Value >

  Declaração da classe IteratorRB como amiga.

5.15.1 Descrição detalhada

template<typename Key, typename Value>
class RedBlackTree< Key, Value >

Definido na linha 26 do ficheiro RedBlackTree.hpp.

5.15.2 Documentação dos Construtores & Destrutor

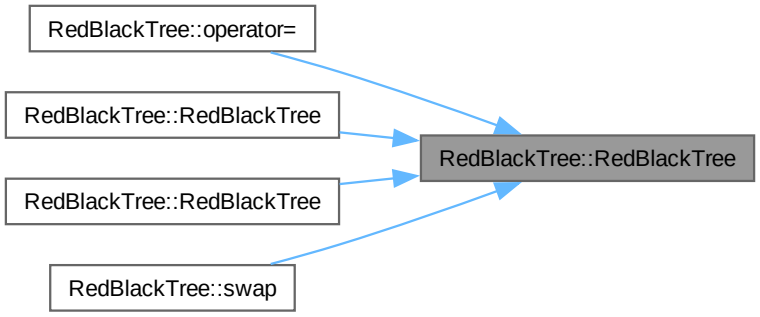
5.15.2.1 RedBlackTree() [1/3]

```
template<typename Key, typename Value>
RedBlackTree< Key, Value >::RedBlackTree ()
```

Construtor padrão. Cria um conjunto vazio.
Inicializa a árvore com um nó sentinela `nil` e a raiz apontando para `nil`. O tamanho é inicializado como 0.
Definido na linha 485 do ficheiro RedBlackTree.hpp.

```
00486 {
00487     nil = new NodeRB<Key, Value>({Key(), Value()}, NodeRB<Key, Value>::BLACK, nullptr, nullptr,
      nullptr);
00488     nil->left = nil->right = nil->parent = nil;
00489     root = nil;
00490     root->parent = nil;
00491 }
```

Este é o diagrama das funções que utilizam esta função:



5.15.2.2 RedBlackTree() [2/3]

```
template<typename Key, typename Value>
RedBlackTree< Key, Value >::RedBlackTree (
            const RedBlackTree< Key, Value > & other)
```

Construtor de cópia. Cria um novo conjunto como cópia de `other`.
Realiza uma cópia profunda de todos os nós da árvore `other`.

Parâmetros

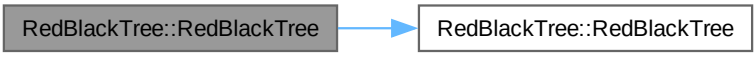
other	O conjunto a ser copiado.

Definido na linha 501 do ficheiro RedBlackTree.hpp.

```
00501                                                                 : RedBlackTree()
00502 {
00503     if (other.root != other.nil)
00504     {
00505         root = clone_recursive(nil, other.root, other.nil);
00506         size_m = other.size_m;
00507         comparisons = other.comparisons;
```

```
00508         rotations = other.rotations;
00509     }
00510 }
```

Grafo de chamadas desta função:



5.15.2.3 RedBlackTree() [3/3]

```
template<typename Key, typename Value>
RedBlackTree< Key, Value >::RedBlackTree (
            std::initializer_list< std::pair< Key, Value > > list)
```

Construtor a partir de uma lista inicializadora.
Cria um conjunto e insere todos os elementos da lista.

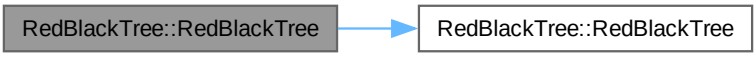
Parâmetros

list	A lista de inicialização (`std::initializer_list<std::pair<Key, Value>>`).

Definido na linha 494 do ficheiro RedBlackTree.hpp.

```
00494                                                                    : RedBlackTree()
00495 {
00496     for (const auto &key : list)
00497         insert(key);
00498 }
```

Grafo de chamadas desta função:



5.15.2.4 ~RedBlackTree()

```
template<typename Key, typename Value>
RedBlackTree< Key, Value >::~RedBlackTree ()
```

Destrutor. Libera toda a memória alocada pelos nós da árvore.
Definido na linha 521 do ficheiro RedBlackTree.hpp.

```
00522 {
00523     clear();
00524     delete nil; // Libera o nó nil (sentinela) após limpar a árvore
00525     nil = nullptr;
00526 }
```

5.15.3 Documentação das funções

5.15.3.1 at()

```
template<typename Key, typename Value>
```

```
Value & RedBlackTree< Key, Value >::at (
            const Key & key)  [inline], [virtual]
```

Acessa o valor associado a uma chave, com verificação de limites.
Se a chave não existir no mapa, uma exceção `std::out_of_range` é lançada.

Parâmetros

key	A chave a ser buscada.

Retorna

Value& Uma referência ao valor associado à chave.

Implementa Dictionary< Key, Value >.
Definido na linha 405 do ficheiro RedBlackTree.hpp.

```
00405 { return at(root, key); };
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.15.3.2 begin() [1/2]

```
template<typename Key, typename Value>
iterator RedBlackTree< Key, Value >::begin () const  [inline], [noexcept]
```

Retorna um iterador constante para o início do conjunto.
O iterador aponta para o menor elemento da árvore (travessia in-order).

Retorna

iterator Um iterador constante para o início do conjunto.

Definido na linha 298 do ficheiro RedBlackTree.hpp.

```
00298 { return iterator(root, nil); }
```

5.15.3.3 begin() [2/2]

```
template<typename Key, typename Value>
iterator RedBlackTree< Key, Value >::begin ()  [inline], [noexcept]
```

Retorna um iterador para o início do conjunto.
O iterador aponta para o menor elemento da árvore (travessia in-order).

Retorna

iterator Um iterador para o início do conjunto.

Definido na linha 280 do ficheiro RedBlackTree.hpp.

```
00280 { return iterator(root, nil); }
```

5.15.3.4 bshow()

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::bshow ()
```

Exibe a estrutura da Árvore Rubro-Negra de forma visual no console.
Útil para depuração e visualização da estrutura e cores dos nós.
Definido na linha 1053 do ficheiro RedBlackTree.hpp.

```
01054 {
01055     bshow(root, "");
01056 }
```

5.15.3.5 cbegin()

```
template<typename Key, typename Value>
iterator RedBlackTree< Key, Value >::cbegin () const  [inline], [noexcept]
```

Retorna um iterador constante para o início do conjunto.
O iterador aponta para o menor elemento da árvore (travessia in-order).

Retorna

iterator Um iterador constante para o início do conjunto.

Definido na linha 316 do ficheiro RedBlackTree.hpp.

```
00316 { return iterator(root, nil); }
```

5.15.3.6 cend()

```
template<typename Key, typename Value>
iterator RedBlackTree< Key, Value >::cend () const  [inline], [noexcept]
```

Retorna um iterador constante para o final do conjunto.
O iterador aponta para a posição após o último elemento (o nó sentinela `nil`).

Retorna

iterator Um iterador constante para o final do conjunto.

Definido na linha 325 do ficheiro RedBlackTree.hpp.

```
00325 { return iterator(nil, nil); }
```

5.15.3.7 clear()

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::clear ()  [virtual]
```

Remove todos os elementos do conjunto.
Após esta operação, `size()` retornará 0 e a árvore conterá apenas a raiz apontando para o nó sentinela `nil`.
Implementa Dictionary< Key, Value >.
Definido na linha 569 do ficheiro RedBlackTree.hpp.

```
00570 {
00571     root = clear(root);
00572     size_m = 0;
00573 }
```

5.15.3.8 clone()

```
template<typename Key, typename Value>
std::unique_ptr< Dictionary< Key, Value > > RedBlackTree< Key, Value >::clone () const  [virtual]
```

Cria uma cópia profunda da árvore Rubro-Negra.
Retorna um ponteiro inteligente para uma nova instância da árvore, contendo os mesmos elementos que a árvore atual.

Retorna

std::unique_ptr<RedBlackTree<Key, Value>> Um ponteiro inteligente para a nova árvore.

Implementa Dictionary< Key, Value >.
Definido na linha 515 do ficheiro RedBlackTree.hpp.

```
00516 {
00517     return std::make_unique<RedBlackTree<Key, Value>>(*this);
00518 }
```

5.15.3.9 contains()

```
template<typename Key, typename Value>
bool RedBlackTree< Key, Value >::contains (
            const Key & key)  [virtual]
```

Verifica se o conjunto contém uma determinada chave.

Parâmetros

key	A chave a ser procurada.

Retorna

true Se a chave estiver presente no conjunto.

false Caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 1020 do ficheiro RedBlackTree.hpp.

```
01021 {
01022     return contains(root, key);
01023 }
```

5.15.3.10 empty()

```
template<typename Key, typename Value>
bool RedBlackTree< Key, Value >::empty () const  [virtual], [noexcept]
```

Verifica se o conjunto está vazio.

Retorna

true Se o conjunto não contiver elementos.

false Caso contrário.

Implementa Dictionary< Key, Value >.
Definido na linha 548 do ficheiro RedBlackTree.hpp.

```
00549 {
00550     return root == nil;
00551 }
```

5.15.3.11 end() [1/2]

```
template<typename Key, typename Value>
iterator RedBlackTree< Key, Value >::end () const  [inline], [noexcept]
```

Retorna um iterador constante para o final do conjunto.
O iterador aponta para a posição após o último elemento (o nó sentinela `nil`).

Retorna

iterator Um iterador constante para o final do conjunto.

Definido na linha 307 do ficheiro RedBlackTree.hpp.

```
00307 { return iterator(nil, nil); }
```

5.15.3.12 end() [2/2]

```
template<typename Key, typename Value>
iterator RedBlackTree< Key, Value >::end ()  [inline], [noexcept]
```

Retorna um iterador para o final do conjunto.
O iterador aponta para a posição após o último elemento (o nó sentinela `nil`).

Retorna

iterator Um iterador para o final do conjunto.

Definido na linha 289 do ficheiro RedBlackTree.hpp.

```
00289 { return iterator(nil, nil); }
```

5.15.3.13 forEach()

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::forEach (
            const std::function< void(const std::pair< Key, Value > &)> & func) const  [virtual]
```

Aplica uma função a cada par chave-valor no conjunto.
Permite iterar sobre todos os elementos do conjunto e aplicar uma função personalizada a cada um deles.

Parâmetros

func	A função a ser aplicada a cada par chave-valor.

Implementa Dictionary< Key, Value >.
Definido na linha 1046 do ficheiro RedBlackTree.hpp.

```
01047 {
01048     for (const auto &pair : *this)
01049         func(pair);
01050 }
```

5.15.3.14 getComparisons()

```
template<typename Key, typename Value>
long long RedBlackTree< Key, Value >::getComparisons () const  [inline], [noexcept]
```

Retorna o número de comparações realizadas durante as operações.
Útil para análise de desempenho da árvore.

Retorna

long long O número total de comparações.

Definido na linha 359 do ficheiro RedBlackTree.hpp.

```
00359 { return comparisons; }
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.15.3.15 getRotations()

```
template<typename Key, typename Value>
long long RedBlackTree< Key, Value >::getRotations () const  [inline], [noexcept]
```

Retorna o número de rotações realizadas durante as operações.
Útil para medir a atividade de rebalanceamento da árvore.

Retorna

long long O número total de rotações.

Definido na linha 368 do ficheiro RedBlackTree.hpp.

```
00368 { return rotations; }
```

5.15.3.16 insert()

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::insert (
            const std::pair< Key, Value > & key)  [virtual]
```

Insere um par chave-valor no conjunto.
Se a chave já existir, a operação é ignorada. Caso contrário, um novo nó é inserido e a árvore é rebalanceada para manter as propriedades Rubro-Negras.

Parâmetros

key	O par chave-valor a ser inserido.

Implementa Dictionary< Key, Value >.
Definido na linha 694 do ficheiro RedBlackTree.hpp.

```
00695 {
00696     root = insert(root, new NodeRB<Key, Value>(key, NodeRB<Key, Value>::RED, nil, nil, nil));
00697 }
```

5.15.3.17 operator=() [1/2]

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::operator= (
            const RedBlackTree< Key, Value > & other)
```

Operador de atribuição por cópia.
Substitui o conteúdo do conjunto atual pelo conteúdo de `other`. Garante a autotribuição segura e libera a memória antiga antes de copiar.

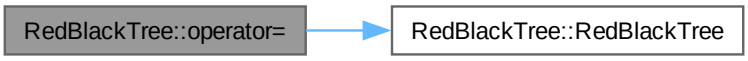
Parâmetros

other	O conjunto a ser copiado.

Definido na linha 529 do ficheiro RedBlackTree.hpp.

```
00530 {
00531     if (this != &other)
00532     {
00533         clear();
00534         clone_recursive(nil, other.root, other.nil);
00535         size_m = other.size_m;
00536         comparisons = other.comparisons;
00537         rotations = other.rotations;
00538     }
00539 }
```

Grafo de chamadas desta função:



5.15.3.18 operator=() [2/2]

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::operator= (
            std::pair< Key, Value > & key)  [inline]
```

Operador de atribuição para atualizar ou inserir um par chave-valor.
Permite usar a sintaxe `tree = {key, value}` para atualizar ou inserir.

Parâmetros

key	O par chave-valor a ser atualizado ou inserido.

Definido na linha 436 do ficheiro RedBlackTree.hpp.

```
00436 { root = update(root, key); };
```

5.15.3.19 operator[]()

```
template<typename Key, typename Value>
Value & RedBlackTree< Key, Value >::operator[] (
            const Key & key)  [virtual]
```

Sobrecarga do operador de indexação para acessar ou inserir um elemento.
Se `key` existir no mapa, retorna uma referência ao seu valor. Se `key` não existir, insere um novo elemento com essa chave (usando o construtor padrão de `Value`) e retorna uma referência ao novo valor.

Parâmetros

key	A chave do elemento a ser acessado ou inserido.

Retorna

Value& Uma referência ao valor associado à chave.

Implementa Dictionary< Key, Value >.
Definido na linha 884 do ficheiro RedBlackTree.hpp.

```
00885 {
00886     NodePtr aux = root;
```

```
00887
00888     while (aux != nil)
00889     {
00890         comparisons++;
00891         if (key == aux->key.first)
00892             return aux->key.second;
00893
00894         comparisons++;
00895         if (key < aux->key.first)
00896             aux = aux->left;
00897         else
00898             aux = aux->right;
00899     }
00900     // Se a chave não for encontrada, insere um novo nó com valor padrão
00901     root = insert(root, new NodeRB<Key, Value>(std::pair<Key, Value>(key, Value()),
      NodeRB<Key, Value>::RED, nil, nil, nil));
00902
00903     return at(root, key); // Retorna o valor associado à nova chave
00904 }
```

5.15.3.20 print()

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::print () const  [virtual]
```

Imprime os elementos do conjunto em ordem crescente (travessia in-order).
Implementa Dictionary< Key, Value >.
Definido na linha 1026 do ficheiro RedBlackTree.hpp.

```
01027 {
01028     printInOrder(root);
01029 }
```

5.15.3.21 remove()

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::remove (
            const Key & key)  [virtual]
```

Remove um elemento do conjunto pela chave.
Se a chave não for encontrada, o conjunto não é modificado. Após a remoção, a árvore é rebalanceada para manter as propriedades Rubro-Negras.

Parâmetros

key	A chave do elemento a ser removido.

Implementa Dictionary< Key, Value >.
Definido na linha 700 do ficheiro RedBlackTree.hpp.

```
00701 {
00702     NodePtr aux = root;
00703
00704     while (aux != nil and aux->key.first != key)
00705     {
00706         comparisons++;
00707         if (key < aux->key.first)
00708             aux = aux->left;
00709         else
00710         {
00711             aux = aux->right;
00712         }
00713     }
00714
00715     if (aux != nil) // Realiza a remoção se a chave for encontrada
00716         remove(aux);
00717
00718     // Se a chave não for encontrada, não faz nada
00719 }
```

5.15.3.22 size()

```
template<typename Key, typename Value>
size_t RedBlackTree< Key, Value >::size () const  [virtual], [noexcept]
```

Retorna o número de elementos no conjunto.

Retorna

size_t O número de elementos.

Implementa Dictionary< Key, Value >.
Definido na linha 542 do ficheiro RedBlackTree.hpp.

```
00543 {
00544     return size_m;
00545 }
```

5.15.3.23 swap()

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::swap (
            RedBlackTree< Key, Value > & other)  [noexcept]
```

Troca o conteúdo deste conjunto com o de `other`.
Operação eficiente que apenas troca os ponteiros internos e os contadores.

Parâmetros

other	O outro conjunto com o qual trocar o conteúdo.

Definido na linha 576 do ficheiro RedBlackTree.hpp.

```
00577 {
00578     std::swap(root, other.root);
00579     std::swap(size_m, other.size_m);
00580     std::swap(nil, other.nil);
00581     std::swap(comparisons, other.comparisons);
00582     std::swap(rotations, other.rotations);
00583 }
```

Grafo de chamadas desta função:

5.15.3.24 update()

```
template<typename Key, typename Value>
void RedBlackTree< Key, Value >::update (
            const std::pair< Key, Value > & key)  [inline], [virtual]
```

Atualiza o valor de uma chave existente ou insere um novo par chave-valor.
Se a chave já existir, seu valor é atualizado. Caso contrário, um novo par chave-valor é inserido.

Parâmetros

key	O par chave-valor a ser atualizado ou inserido.

Implementa Dictionary< Key, Value >.
Definido na linha 427 do ficheiro RedBlackTree.hpp.

```
00427 { update(root, key); };
```

Grafo de chamadas desta função:

Este é o diagrama das funções que utilizam esta função:

5.15.4 Documentação dos símbolos amigos e relacionados

5.15.4.1 IteratorRB< Key, Value >

```
template<typename Key, typename Value>
friend class IteratorRB< Key, Value >  [friend]
```

Declaração da classe IteratorRB como amiga.
Permite que a classe IteratorRB acesse membros privados e protegidos da classe RedBlackTree, facilitando a implementação de iteração sobre a árvore.
Definido na linha 1059 do ficheiro RedBlackTree.hpp.
A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/rb_tree/RedBlackTree.hpp

5.16 Referência à classe Template Slot< Key, Value >

Representa um único slot em uma tabela hash de endereçamento aberto.
`#include <Slot.hpp>`

Membros públicos

- Slot ()=default

  Construtor padrão.
- Slot (const std::pair< Key, Value > &pair)

  Construtor que inicializa o slot com um par chave-valor.
- Slot (const Key &key, const Value &value)

  Construtor que inicializa o slot com uma chave e um valor.
- bool is_empty () const noexcept

  Verifica se o slot está vazio.
- bool is_active () const noexcept

Verifica se o slot está ativo.

- bool is_deleted () const noexcept

  Verifica se o slot foi marcado como deletado.

Atributos Públicos

- std::pair< Key, Value > data {}

  O par chave-valor armazenado no slot.

- HashTableStatus status {HashTableStatus::EMPTY}

  O estado atual do slot (EMPTY, ACTIVE, ou DELETED).

5.16.1 Descrição detalhada

template<typename Key, typename Value>
class Slot< Key, Value >

Representa um único slot em uma tabela hash de endereçamento aberto.
Cada slot armazena um par chave-valor e um status que indica se o slot está vazio, ativo (contendo dados válidos) ou se foi deletado (marcado como uma lápide para não interromper as sequências de sondagem).

Parâmetros de template

Key	O tipo da chave.
Value	O tipo do valor.

Definido na linha 32 do ficheiro Slot.hpp.

5.16.2 Documentação dos Construtores & Destrutor

5.16.2.1 Slot() [1/3]

```
template<typename Key, typename Value>
Slot< Key, Value >::Slot ()  [default]
```

Construtor padrão.
Inicializa um slot com o estado `EMPTY`.

5.16.2.2 Slot() [2/3]

```
template<typename Key, typename Value>
Slot< Key, Value >::Slot (
            const std::pair< Key, Value > & pair)  [inline], [explicit]
```

Construtor que inicializa o slot com um par chave-valor.
Cria um slot e o marca como `ACTIVE`.

Parâmetros

pair	O `std::pair<Key, Value>` a ser armazenado.

Definido na linha 59 do ficheiro Slot.hpp.

```
00060         : data(pair), status(HashTableStatus::ACTIVE) {}
```

5.16.2.3 Slot() [3/3]

```
template<typename Key, typename Value>
Slot< Key, Value >::Slot (
            const Key & key,
            const Value & value)  [inline]
```

Construtor que inicializa o slot com uma chave e um valor.
Cria um slot a partir de uma chave e um valor e o marca como `ACTIVE`.

Parâmetros

key	A chave a ser armazenada.
value	O valor a ser armazenado.

Definido na linha 69 do ficheiro Slot.hpp.

```
00070         : data({key, value}), status(HashTableStatus::ACTIVE) {}
```

5.16.3 Documentação das funções

5.16.3.1 is_active()

```
template<typename Key, typename Value>
bool Slot< Key, Value >::is_active () const  [inline], [noexcept]
```

Verifica se o slot está ativo.

Retorna

`true` se o status do slot for `ACTIVE`, `false` caso contrário.

Definido na linha 85 do ficheiro Slot.hpp.

```
00086     {
00087         return status == HashTableStatus::ACTIVE;
00088     }
```

5.16.3.2 is_deleted()

```
template<typename Key, typename Value>
bool Slot< Key, Value >::is_deleted () const  [inline], [noexcept]
```

Verifica se o slot foi marcado como deletado.

Retorna

`true` se o status do slot for `DELETED`, `false` caso contrário.

Definido na linha 94 do ficheiro Slot.hpp.

```
00095     {
00096         return status == HashTableStatus::DELETED;
00097     }
```

5.16.3.3 is_empty()

```
template<typename Key, typename Value>
bool Slot< Key, Value >::is_empty () const  [inline], [noexcept]
```

Verifica se o slot está vazio.

Retorna

`true` se o status do slot for `EMPTY`, `false` caso contrário.

Definido na linha 76 do ficheiro Slot.hpp.

```
00077     {
00078         return status == HashTableStatus::EMPTY;
00079     }
```

5.16.4 Documentação dos dados membro

5.16.4.1 data

```
template<typename Key, typename Value>
std::pair<Key, Value> Slot< Key, Value >::data {}
```

O par chave-valor armazenado no slot.

Definido na linha 38 do ficheiro Slot.hpp.

```
00038 {};
```

5.16.4.2 status

```
template<typename Key, typename Value>
HashTableStatus Slot< Key, Value >::status {HashTableStatus::EMPTY}
```

O estado atual do slot (EMPTY, ACTIVE, ou DELETED).
Definido na linha 44 do ficheiro Slot.hpp.

```
00044 {HashTableStatus::EMPTY};
```

A documentação para esta classe foi gerada a partir do seguinte ficheiro:

- include/dictionary/hash_table_o/Slot.hpp

5.17 Referência à classe TextProcessor

Responsável por processar ficheiros de texto, extraindo e normalizando palavras.
`#include <TextProcessor.hpp>`

Membros públicos

- TextProcessor (const std::string &input_file)

  Construtor da classe TextProcessor.
- ~TextProcessor ()=default

  Destrutor padrão da classe TextProcessor.
- void processFile (const std::function< void(const std::string &)> &wordHandler)

  Processa o ficheiro de texto palavra por palavra.

Membros públicos estáticos

- static void toLowerCase (std::string &text)

  Converte uma string para minúsculas.

5.17.1 Descrição detalhada

Responsável por processar ficheiros de texto, extraindo e normalizando palavras.
Esta classe abre um ficheiro de texto, lê o seu conteúdo palavra por palavra, normaliza cada palavra (converte para minúsculas e remove caracteres inválidos) e, em seguida, passa cada palavra válida para uma função de callback fornecida pelo utilizador.
O processador de texto utiliza uma expressão regular para identificar palavras válidas. As palavras são consideradas válidas se consistirem apenas de letras (incluindo acentuadas), hífens e apóstrofos, desde que não estejam no início ou no fim da palavra.
Definido na linha 26 do ficheiro TextProcessor.hpp.

5.17.2 Documentação dos Construtores & Destrutor

5.17.2.1 TextProcessor()

```
TextProcessor::TextProcessor (
            const std::string & input_file)  [explicit]
```

Construtor da classe TextProcessor.
Inicializa o processador de texto abrindo o ficheiro de entrada especificado. Se o ficheiro não puder ser aberto, uma mensagem de erro crítico é exibida e o programa é encerrado.

Parâmetros

input_file	O nome do ficheiro de texto a ser processado.

Excepções

std::runtime_error	Se o ficheiro não puder ser aberto, uma exceção é lançada. A mensagem de erro é exibida no fluxo de erro padrão. O programa é encerrado com um código de erro implícito.

Nota

O ficheiro deve existir e ser legível. Caso contrário, o programa exibirá uma mensagem de erro e encerrará a execução.

Definido na linha 3 do ficheiro TextProcessor.cpp.

```
00003                                                       : file_stream(input_file)
00004 {
00005     if (!file_stream or !file_stream.is_open())
00006         throw std::runtime_error("CRITICAL ERROR: Could not open file: " + input_file + ".");
00007 }
```

5.17.2.2 ~TextProcessor()

```
TextProcessor::~TextProcessor ()  [default]
```

Destrutor padrão da classe TextProcessor.

5.17.3 Documentação das funções

5.17.3.1 processFile()

```
void TextProcessor::processFile (
            const std::function< void(const std::string &)> & wordHandler)
```

Processa o ficheiro de texto palavra por palavra.
Lê o ficheiro associado a esta instância, extraindo palavras. Cada palavra é então normalizada através da função `normalize`. Se a palavra normalizada não for vazia, ela é passada para a função de callback `wordHandler`.

Parâmetros

wordHandler	Uma função (ou objeto de função, como um lambda) que aceita uma `const std::string&` e será chamada para cada palavra válida e normalizada encontrada no ficheiro.

Definido na linha 15 do ficheiro TextProcessor.cpp.

```
00016 {
00017     std::string word{};
00018
00019     while (file_stream » word)
00020     {
00021         std::string cleanedWord{normalize(word)};
00022
00023         if (!cleanedWord.empty())
00024             wordHandler(cleanedWord);
00025     }
00026 }
```

Este é o diagrama das funções que utilizam esta função:

5.17.3.2 toLowerCase()

```
void TextProcessor::toLowerCase (
            std::string & text)  [static]
```

Converte uma string para minúsculas.
Esta é uma função de utilidade estática que modifica a string fornecida, convertendo todos os seus caracteres para a forma minúscula.

Parâmetros

text	A string a ser convertida. A conversão é feita no local (in-place).

Definido na linha 9 do ficheiro TextProcessor.cpp.

```
00010 {
00011     std::transform(text.begin(), text.end(), text.begin(), [](unsigned char c)
00012                    { return std::tolower(c); });
00013 }
```

Este é o diagrama das funções que utilizam esta função:

A documentação para esta classe foi gerada a partir dos seguintes ficheiros:

- include/text_processor/TextProcessor.hpp
- src/text_processor/TextProcessor.cpp

Capítulo 6

Documentação do ficheiro

6.1 Referência ao ficheiro build/main/main.d

6.2 main.d

Ir para a documentação deste ficheiro.

```
00001 main.o: src/main/main.cpp
```

6.3 Referência ao ficheiro build/text_processor/TextProcessor.d

6.4 TextProcessor.d

Ir para a documentação deste ficheiro.

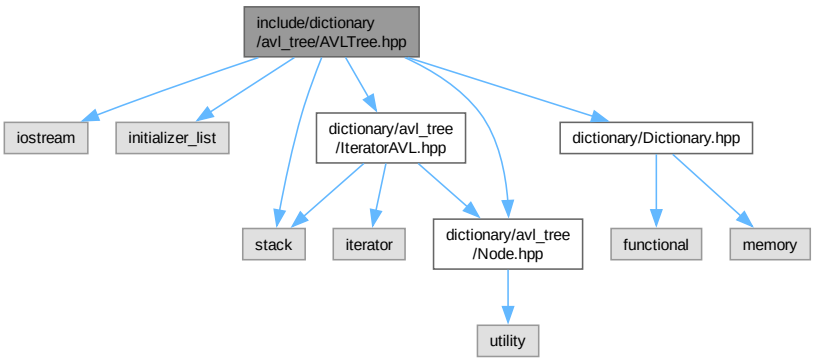
```
00001 TextProcessor.o: src/text_processor/TextProcessor.cpp
```

6.5 Referência ao ficheiro include/dictionary/avl_tree/AVLTree.hpp

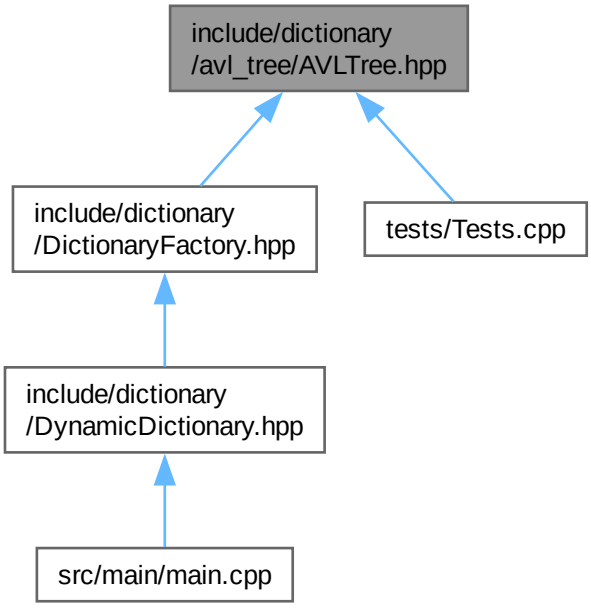
Classe que implementa uma Árvore AVL.

```
#include <iostream>
#include <initializer_list>
#include <stack>
#include "dictionary/avl_tree/Node.hpp"
#include "dictionary/avl_tree/IteratorAVL.hpp"
#include "dictionary/Dictionary.hpp"
```

Diagrama de dependências de inclusão para AVLTree.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class AVLTree< Key, Value >

6.5.1 Descrição detalhada

Classe que implementa uma Árvore AVL.
A Árvore AVL é uma árvore de busca binária auto-balanceável, onde a diferença entre as alturas das subárvores esquerda e direita de qualquer nó é no máximo 1. Isso garante que as operações de busca, inserção e remoção tenham complexidade de tempo O(log n) no pior caso, onde n é o número de elementos no conjunto.

Parâmetros de template

Key,Value	Tipo dos elementos armazenados no conjunto. Deve suportar operadores de comparação (<, ==, >).

Definido no ficheiro AVLTree.hpp.

6.6 AVLTree.hpp

Ir para a documentação deste ficheiro.

```
00001 #pragma once
00002
00003 #include <iostream>
00004 #include <initializer_list>
00005 #include <stack>
00006
00007 #include "dictionary/avl_tree/Node.hpp"
00008 #include "dictionary/avl_tree/IteratorAVL.hpp"
00009 #include "dictionary/Dictionary.hpp"
00010
```

```
00024 template <typename Key, typename Value>
00025 class AVLTree : public Dictionary<Key, Value>
00026 {
00030     using NodePtr = Node<Key, Value> *;
00031
00038     friend class IteratorAVL<Key, Value>;
00039
00046     using iterator = IteratorAVL<Key, Value>;
00047
00048 private:
00052     Node<Key, Value> *root{nullptr};
00053
00057     size_t size_m{0};
00058
00064     long long comparisons{0};
00065
00072     long long rotations{0};
00073
00084     Node<Key, Value> *fixup_node(NodePtr p);
00085
00096     Node<Key, Value> *insert(NodePtr p, const std::pair<Key, Value> &key);
00097
00108     Node<Key, Value> *update(NodePtr p, const std::pair<Key, Value> &key);
00109
00120     Value &at(NodePtr p, const Key &key);
00121
00132     Node<Key, Value> *fixup_deletion(NodePtr p);
00133
00145     Node<Key, Value> *m_remove(NodePtr p, const Key &key);
00146
00158     Node<Key, Value> *remove_successor(NodePtr root, NodePtr node);
00159
00168     Node<Key, Value> *clear(NodePtr root);
00169
00179     int updateHeight(NodePtr node);
00180
00187     int height(NodePtr node);
00188
00198     int balance(NodePtr node);
00199
00208     Node<Key, Value> *rightRotation(NodePtr p);
00209
00218     Node<Key, Value> *leftRotation(NodePtr p);
00219
00228     bool contains(NodePtr root, const Key &key);
00229
00241     NodePtr clone_recursive(const NodePtr &node_other) const;
00242
00248     void printInOrder(NodePtr node) const;
00249
00256     void bshow(NodePtr node, std::string heranca);
00257
00258 public:
00262     AVLTree() = default;
00263
00271     AVLTree(const AVLTree &other);
00272
00280     AVLTree(std::initializer_list<std::pair<Key, Value» list);
00281
00290     std::unique_ptr<Dictionary<Key, Value» clone() const;
00291
00295     ~AVLTree();
00296
00304     iterator begin() noexcept { return iterator(root); }
00305
00313     iterator end() noexcept { return iterator(); }
00314
00322     iterator begin() const noexcept { return iterator(root); }
00323
00331     iterator end() const noexcept { return iterator(); }
00332
00340     iterator cbegin() const noexcept { return iterator(root); }
00341
00349     iterator cend() const noexcept { return iterator(); }
00350
00359     void operator=(const AVLTree &other);
00360
00366     size_t size() const noexcept;
00367
00374     bool empty() const noexcept;
00375
00383     long long getComparisons() const noexcept { return comparisons; }
00384
00392     long long getRotations() const noexcept { return rotations; }
00393
00399     void clear();
00400
```

```
00408     void swap(AVLTree<Key, Value> &other) noexcept;
00409
00418     void insert(const std::pair<Key, Value> &key);
00419
00428     Value &at(const Key &key) { return at(root, key); };
00429
00439     Value &operator[](const Key &key);
00440
00451     void update(const std::pair<Key, Value> &key) { root = update(root, key); };
00452
00460     void operator=(std::pair<Key, Value> &key) { root = update(root, key); };
00461
00470     void remove(const Key &key);
00471
00479     bool contains(const Key &key);
00480
00481     // Funções de impressão
00482
00486     void print() const;
00487
00496     void forEach(const std::function<void(const std::pair<Key, Value> &)> &func) const;
00497
00503     void bshow();
00504 };
00505
00506 // ----------------------------------------Implementação da classe
      AVLTree.-------------------------------------------------------------------
00507
00508 template <typename Key, typename Value>
00509 AVLTree<Key, Value>::AVLTree(std::initializer_list<std::pair<Key, Value» list) : AVLTree()
00510 {
00511     for (const auto &key : list)
00512         insert(key);
00513 }
00514
00515 template <typename Key, typename Value>
00516 AVLTree<Key, Value>::AVLTree(const AVLTree &other) : AVLTree()
00517 {
00518     if (other.root != nullptr)
00519     {
00520         root = clone_recursive(other.root);
00521         size_m = other.size_m;
00522         comparisons = other.comparisons;
00523         rotations = other.rotations;
00524     }
00525 }
00526
00527 template <typename Key, typename Value>
00528 std::unique_ptr<Dictionary<Key, Value» AVLTree<Key, Value>::clone() const
00529 {
00530     return std::make_unique<AVLTree<Key, Value»(*this);
00531 }
00532
00533 template <typename Key, typename Value>
00534 AVLTree<Key, Value>::~AVLTree()
00535 {
00536     clear();
00537 }
00538
00539 template <typename Key, typename Value>
00540 void AVLTree<Key, Value>::operator=(const AVLTree &other)
00541 {
00542     if (this != &other)
00543     {
00544         clear(); // Limpa a árvore atual
00545         if (other.root != nullptr)
00546         {
00547             root = clone_recursive(other.root);
00548             size_m = other.size_m;
00549             comparisons = other.comparisons;
00550             rotations = other.rotations;
00551         }
00552     }
00553 }
00554
00555 template <typename Key, typename Value>
00556 size_t AVLTree<Key, Value>::size() const noexcept
00557 {
00558     return size_m;
00559 }
00560
00561 template <typename Key, typename Value>
00562 bool AVLTree<Key, Value>::empty() const noexcept
00563 {
00564     return root == nullptr;
00565 }
00566
```

```
00567 template <typename Key, typename Value>
00568 Node<Key, Value> *AVLTree<Key, Value>::clear(NodePtr root)
00569 {
00570     if (root != nullptr)
00571     {
00572         root->left = clear(root->left);
00573         root->right = clear(root->right);
00574 
00575         delete root;
00576         return nullptr;
00577     }
00578 
00579     return root;
00580 }
00581 
00582 template <typename Key, typename Value>
00583 void AVLTree<Key, Value>::clear()
00584 {
00585     root = clear(root);
00586     size_m = 0;
00587 }
00588 
00589 template <typename Key, typename Value>
00590 void AVLTree<Key, Value>::swap(AVLTree<Key, Value> &other) noexcept
00591 {
00592     std::swap(root, other.root);
00593     std::swap(size_m, other.size_m);
00594     std::swap(comparisons, other.comparisons);
00595     std::swap(rotations, other.rotations);
00596 }
00597 
00598 template <typename Key, typename Value>
00599 Node<Key, Value> *AVLTree<Key, Value>::fixup_node(NodePtr p)
00600 {
00601     p->height = updateHeight(p);
00602 
00603     int bal = balance(p);
00604 
00605     if (bal == -2 and height(p->left->left) > height(p->left->right))
00606     {
00607         return rightRotation(p);
00608     }
00609     else if (bal == -2 and height(p->left->left) < height(p->left->right))
00610     {
00611         p->left = leftRotation(p->left);
00612         return rightRotation(p);
00613     }
00614     else if (bal == 2 and height(p->right->right) > height(p->right->left))
00615     {
00616         return leftRotation(p);
00617     }
00618     else if (bal == 2 and height(p->right->right) < height(p->right->left))
00619     {
00620         p->right = rightRotation(p->right);
00621         return leftRotation(p);
00622     }
00623 
00624     return p;
00625 }
00626 
00627 template <typename Key, typename Value>
00628 Node<Key, Value> *AVLTree<Key, Value>::insert(NodePtr p, const std::pair<Key, Value> &key)
00629 {
00630     if (p == nullptr)
00631     {
00632         size_m++;
00633         return new Node<Key, Value>(key);
00634     }
00635 
00636     comparisons++;
00637     if (key.first == p->key.first)
00638         return p;
00639 
00640     comparisons++;
00641     if (key < p->key)
00642         p->left = insert(p->left, key);
00643     else
00644         p->right = insert(p->right, key);
00645 
00646     p = fixup_node(p);
00647 
00648     return p;
00649 }
00650 
00651 template <typename Key, typename Value>
00652 void AVLTree<Key, Value>::insert(const std::pair<Key, Value> &key)
00653 {
```

```
00654     root = insert(root, key);
00655 }
00656
00657 template <typename Key, typename Value>
00658 void AVLTree<Key, Value>::remove(const Key &key)
00659 {
00660     root = m_remove(root, key);
00661 }
00662
00663 template <typename Key, typename Value>
00664 Node<Key, Value> *AVLTree<Key, Value>::fixup_deletion(NodePtr p)
00665 {
00666     int bal = balance(p);
00667
00668     if (bal == 2 and balance(p->right) >= 0)
00669     {
00670         return leftRotation(p);
00671     }
00672     if (bal == 2 and balance(p->right) < 0)
00673     {
00674         p->right = rightRotation(p->right);
00675         return leftRotation(p);
00676     }
00677     if (bal == -2 and balance(p->left) <= 0)
00678     {
00679         return rightRotation(p);
00680     }
00681     if (bal == -2 and balance(p->left) > 0)
00682     {
00683         p->left = leftRotation(p->left);
00684         return rightRotation(p);
00685     }
00686
00687     p->height = updateHeight(p);
00688
00689     return p;
00690 }
00691
00692 template <typename Key, typename Value>
00693 Node<Key, Value> *AVLTree<Key, Value>::m_remove(NodePtr p, const Key &key)
00694 {
00695     if (p == nullptr)
00696         return p;
00697
00698     comparisons++;
00699     if (key < p->key.first)
00700         p->left = m_remove(p->left, key);
00701     else if (key > p->key.first)
00702     {
00703         comparisons++;
00704         p->right = m_remove(p->right, key);
00705     }
00706     else if (p->right == nullptr)
00707     {
00708         comparisons += 2;
00709
00710         NodePtr child = p->left;
00711         delete p;
00712         size_m--;
00713         return child;
00714     }
00715     else
00716         p->right = remove_successor(p, p->right);
00717
00718     p = fixup_deletion(p);
00719
00720     return p;
00721 }
00722
00723 template <typename Key, typename Value>
00724 Node<Key, Value> *AVLTree<Key, Value>::remove_successor(NodePtr root, NodePtr node)
00725 {
00726     if (node->left != nullptr)
00727         node->left = remove_successor(root, node->left);
00728     else
00729     {
00730         root->key = node->key;
00731         NodePtr aux = node->right;
00732         delete node;
00733         size_m--;
00734         return aux;
00735     }
00736
00737     node = fixup_deletion(node);
00738
00739     return node;
00740 }
```

```
00741
00742 template <typename Key, typename Value>
00743 Value &AVLTree<Key, Value>::at(NodePtr p, const Key &key)
00744 {
00745     if (p == nullptr)
00746         throw std::out_of_range("Key not found in AVL Tree");
00747
00748     comparisons++;
00749     if (key == p->key.first)
00750         return p->key.second;
00751
00752     comparisons++;
00753     if (key < p->key.first)
00754     {
00755         return at(p->left, key);
00756     }
00757     else
00758         return at(p->right, key);
00759 }
00760
00761 template <typename Key, typename Value>
00762 Value &AVLTree<Key, Value>::operator[](const Key &key)
00763 {
00764     NodePtr aux = root;
00765
00766     while (aux != nullptr)
00767     {
00768         comparisons++;
00769         if (key == aux->key.first)
00770             return aux->key.second;
00771
00772         comparisons++;
00773         if (key < aux->key.first)
00774             aux = aux->left;
00775         else
00776             aux = aux->right;
00777     }
00778     // Se a chave não for encontrada, insere um novo nó com valor padrão
00779     root = insert(root, {key, Value()});
00780
00781     return at(root, key); // Retorna o valor associado à nova chave
00782 }
00783
00784 template <typename Key, typename Value>
00785 Node<Key, Value> *AVLTree<Key, Value>::update(NodePtr p, const std::pair<Key, Value> &key)
00786 {
00787     if (p == nullptr)
00788     {
00789         throw std::out_of_range("Key not found in AVL Tree");
00790     }
00791
00792     comparisons++;
00793     if (key.first == p->key.first)
00794     {
00795         p->key.second = key.second; // Atualiza o valor
00796         return p;
00797     }
00798     else if (key.first < p->key.first)
00799     {
00800         comparisons++;
00801         p->left = update(p->left, key);
00802     }
00803     else
00804     {
00805         comparisons += 2;
00806         p->right = update(p->right, key);
00807     }
00808     p = fixup_node(p);
00809
00810     return p;
00811 }
00812
00813 template <typename Key, typename Value>
00814 int AVLTree<Key, Value>::updateHeight(NodePtr node)
00815 {
00816     return 1 + std::max(height(node->left), height(node->right));
00817 }
00818
00819 template <typename Key, typename Value>
00820 int AVLTree<Key, Value>::height(NodePtr node)
00821 {
00822     return (!node) ? 0 : node->height;
00823 }
00824
00825 template <typename Key, typename Value>
00826 int AVLTree<Key, Value>::balance(NodePtr node)
00827 {
```

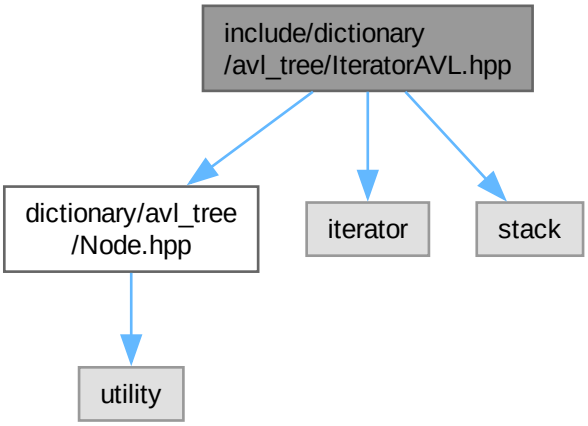
```
00828     return height(node->right) - height(node->left);
00829 }
00830
00831 template <typename Key, typename Value>
00832 Node<Key, Value> *AVLTree<Key, Value>::rightRotation(NodePtr p)
00833 {
00834     rotations++;
00835
00836     NodePtr aux = p->left;
00837     p->left = aux->right;
00838     aux->right = p;
00839
00840     p->height = updateHeight(p);
00841     aux->height = updateHeight(aux);
00842
00843     return aux;
00844 }
00845
00846 template <typename Key, typename Value>
00847 Node<Key, Value> *AVLTree<Key, Value>::leftRotation(NodePtr p)
00848 {
00849     rotations++;
00850
00851     NodePtr aux = p->right;
00852     p->right = aux->left;
00853     aux->left = p;
00854
00855     p->height = updateHeight(p);
00856     aux->height = updateHeight(aux);
00857
00858     return aux;
00859 }
00860
00861 template <typename Key, typename Value>
00862 Node<Key, Value> *AVLTree<Key, Value>::clone_recursive(const NodePtr &node_other) const
00863 {
00864     if (node_other == nullptr)
00865         return nullptr;
00866
00867     // Cria o novo nó com os mesmos dados e altura
00868     NodePtr new_node = new Node<Key, Value>(node_other->key);
00869     new_node->height = node_other->height;
00870
00871     // Define recursivamente os filhos esquerdo e direito
00872     new_node->left = clone_recursive(node_other->left);
00873     new_node->right = clone_recursive(node_other->right);
00874
00875     return new_node;
00876 }
00877
00878 template <typename Key, typename Value>
00879 bool AVLTree<Key, Value>::contains(NodePtr node, const Key &key)
00880 {
00881     if (node == nullptr)
00882         return false;
00883
00884     comparisons++;
00885     if (key == node->key.first)
00886         return true;
00887
00888     comparisons++;
00889     if (key < node->key.first)
00890     {
00891         return contains(node->left, key);
00892     }
00893     else
00894         return contains(node->right, key);
00895 }
00896
00897 template <typename Key, typename Value>
00898 bool AVLTree<Key, Value>::contains(const Key &key)
00899 {
00900     return contains(root, key);
00901 }
00902
00903 template <typename Key, typename Value>
00904 void AVLTree<Key, Value>::print() const
00905 {
00906     printInOrder(root);
00907 }
00908
00909 template <typename Key, typename Value>
00910 void AVLTree<Key, Value>::printInOrder(NodePtr node) const
00911 {
00912     if (node == nullptr)
00913         return;
00914
```

```
00915     else
00916     {
00917         printInOrder(node->left);
00918         std::cout « "[" « node->key.first « ", " « node->key.second « "]" « std::endl;
00919         printInOrder(node->right);
00920     }
00921 }
00922
00923 template <typename Key, typename Value>
00924 void AVLTree<Key, Value>::forEach(const std::function<void(const std::pair<Key, Value> &)> &func)
      const
00925 {
00926     for (const auto &key : *this)
00927         func(key);
00928 }
00929
00930 template <typename Key, typename Value>
00931 void AVLTree<Key, Value>::bshow()
00932 {
00933     bshow(root, "");
00934 }
00935
00936 template <typename Key, typename Value>
00937 void AVLTree<Key, Value>::bshow(NodePtr node, std::string heranca)
00938 {
00939     if (node != nullptr and (node->left != nullptr or node->right != nullptr))
00940         bshow(node->right, heranca + "r");
00941
00942     for (int i = 0; i < (int)heranca.size() - 1; i++)
00943         std::cout « (heranca[i] != heranca[i + 1] ? "    " : "    ");
00944
00945     if (heranca != "")
00946         std::cout « (heranca.back() == 'r' ? "" : "");
00947
00948     if (node == nullptr)
00949     {
00950         std::cout « "#" « std::endl;
00951         return;
00952     }
00953
00954     std::cout « "[" « node->key.first « ", " « node->key.second « "]" « std::endl;
00955
00956     if (node != nullptr and (node->left != nullptr or node->right != nullptr))
00957         bshow(node->left, heranca + "l");
00958 }
```

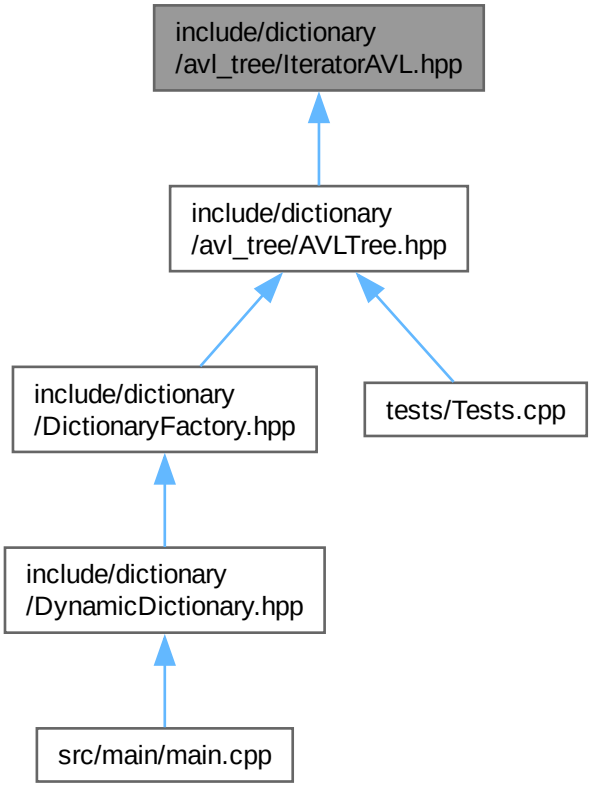
6.7 Referência ao ficheiro include/dictionary/avl_tree/IteratorAVL.hpp

```
#include "dictionary/avl_tree/Node.hpp"
#include <iterator>
#include <stack>
```

Diagrama de dependências de inclusão para IteratorAVL.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class IteratorAVL< Key, Value >

  Um iterador para a árvore AVL.

6.8 IteratorAVL.hpp

Ir para a documentação deste ficheiro.

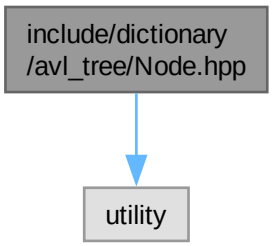
```
00001 #pragma once
00002
00003 #include "dictionary/avl_tree/Node.hpp"
00004 #include <iterator>
00005 #include <stack>
00006
00007 template <typename Key, typename Value>
00008 class AVLTree;
00009
00019 template <typename Key, typename Value>
00020 class IteratorAVL
00021 {
00022 private:
00024     using NodePtr = Node<Key, Value> *;
00026     std::stack<NodePtr> path;
00027
00028 public:
00030     using iterator_category = std::input_iterator_tag;
00032     using value_type = std::pair<Key, Value>;
00034     using difference_type = std::ptrdiff_t;
00036     using pointer = value_type *;
00038     using reference = value_type &;
00040     using const_pointer = const value_type *;
00042     using const_reference = const value_type &;
00044     using NodeType = Node<Key, Value>;
00046     using NodePtrType = NodePtr;
00047
00053     IteratorAVL() {}
00054
00063     IteratorAVL(NodePtr root)
00064     {
00065         NodePtr current = root;
00066         while (current != nullptr)
00067         {
00068             path.push(current);
00069             current = current->left;
00070         }
00071     }
00072
00080     reference operator*() const
00081     {
00082         return path.top()->key;
00083     }
00084
00092     pointer operator->() const
00093     {
00094         return &(path.top()->key);
00095     }
00096
00104     IteratorAVL &operator++()
00105     {
00106         if (path.empty())
00107             return *this;
00108
00109         NodePtr node = path.top();
00110         path.pop();
00111
00112         if (node->right != nullptr)
00113         {
00114             NodePtr current = node->right;
00115
00116             while (current != nullptr)
00117             {
00118                 path.push(current);
00119                 current = current->left;
00120             }
00121         }
00122
00123         return *(this);
00124     }
00125
00134     IteratorAVL operator++(int)
00135     {
00136         IteratorAVL temp = *this;
```

```
00137         ++(*this);
00138         return temp;
00139     }
00140
00150     bool operator==(const IteratorAVL &other) const
00151     {
00152         if (path.empty() && other.path.empty())
00153             return true;
00154
00155         if (path.empty() || other.path.empty())
00156             return false;
00157
00158         return path.top() == other.path.top();
00159     }
00160
00170     bool operator!=(const IteratorAVL &other) const
00171     {
00172         return !(*this == other);
00173     }
00174 };
```

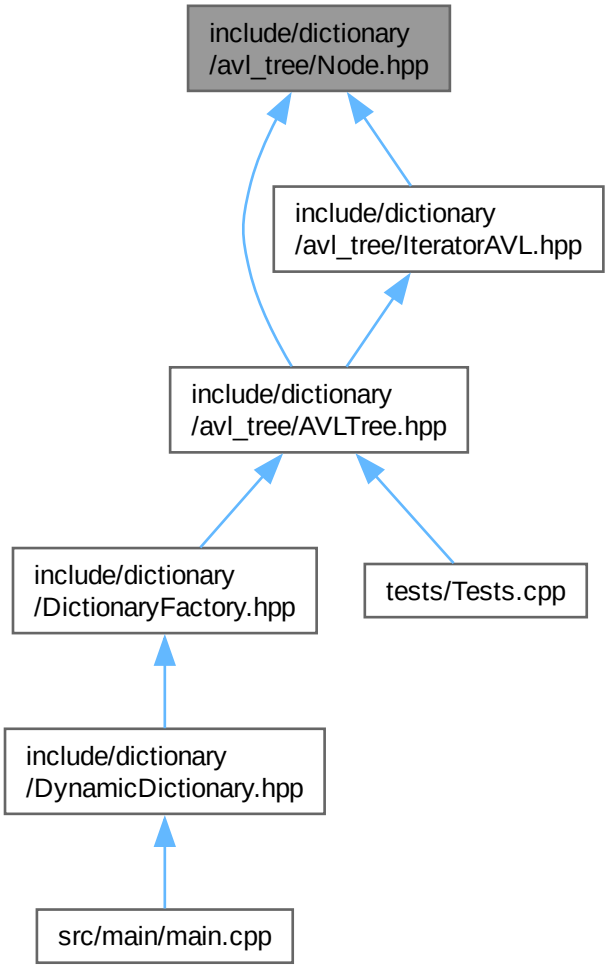
6.9 Referência ao ficheiro include/dictionary/avl_tree/Node.hpp

`#include <utility>`

Diagrama de dependências de inclusão para Node.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- struct Node< Key, Value >

  Estrutura que representa um nó em uma árvore binária, comumente utilizada em árvores AVL.

6.10 Node.hpp

Ir para a documentação deste ficheiro.

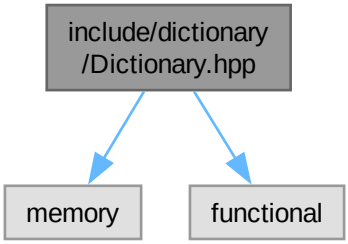
```
00001 #pragma once
00002
00003 #include <utility>
00004
00060 template <typename Key, typename Value>
00061 struct Node
00062 {
00063     std::pair<Key, Value> key;
00064     int height;
00065     Node<Key, Value> *left;
00066     Node<Key, Value> *right;
00067
```

```
00068     Node(const std::pair<Key, Value> &key, const int &height = 1, Node<Key, Value> *left = nullptr,
      Node<Key, Value> *right = nullptr)
00069         : key(key), height(height), left(left), right(right) {}
00070 };
```

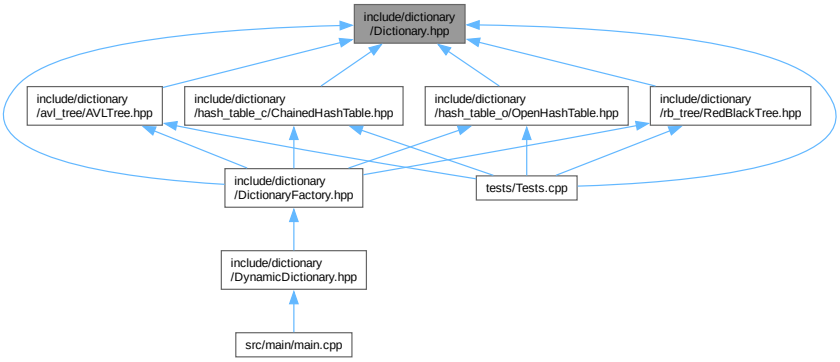
6.11 Referência ao ficheiro include/dictionary/Dictionary.hpp

```
#include <memory>
#include <functional>
```

Diagrama de dependências de inclusão para Dictionary.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class Dictionary< Key, Value >

  Define a interface para uma estrutura de dados de dicionário (ou mapa).

6.12 Dictionary.hpp

Ir para a documentação deste ficheiro.

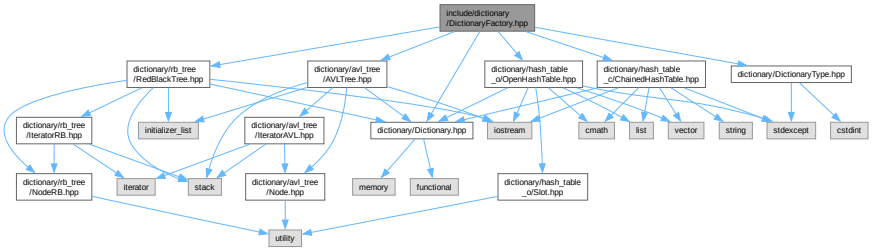
```
00001 #pragma once
00002
00003 #include <memory>
00004 #include <functional>
00005
00016 template <typename Key, typename Value>
00017 class Dictionary
```

```
00018 {
00019 public:
00023     virtual ~Dictionary() = default;
00024
00030     virtual void insert(const std::pair<Key, Value> &key_value) = 0;
00031
00037     virtual void remove(const Key &key) = 0;
00038
00044     virtual void update(const std::pair<Key, Value> &key_value) = 0;
00045
00053     virtual bool contains(const Key &key) = 0;
00054
00061     virtual Value &at(const Key &key) = 0;
00062
00069     virtual Value &operator[](const Key &key) = 0;
00070
00074     virtual void clear() = 0;
00075
00081     virtual size_t size() const noexcept = 0;
00082
00089     virtual bool empty() const noexcept = 0;
00090
00097     virtual void print() const = 0;
00098
00104     virtual void forEach(const std::function<void(const std::pair<Key, Value> &)> &func) const = 0;
00105
00111     virtual std::unique_ptr<Dictionary<Key, Value»
00112     clone() const = 0;
00113 };
```

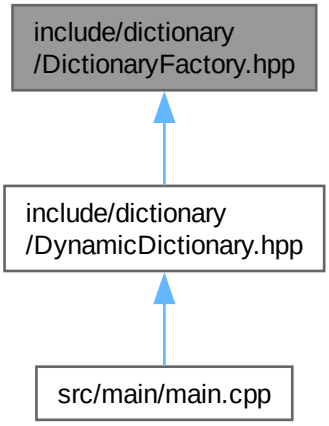
6.13 Referência ao ficheiro include/dictionary/DictionaryFactory.hpp

```
#include "dictionary/avl_tree/AVLTree.hpp"
#include "dictionary/rb_tree/RedBlackTree.hpp"
#include "dictionary/hash_table_c/ChainedHashTable.hpp"
#include "dictionary/hash_table_o/OpenHashTable.hpp"
#include "dictionary/Dictionary.hpp"
#include "dictionary/DictionaryType.hpp"
```

Diagrama de dependências de inclusão para DictionaryFactory.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Funções

- template<typename Key, typename Value>
  std::unique_ptr< Dictionary< Key, Value > > create_dictionary (const DictionaryType &type=DictionaryType::RBTREE)

  Cria uma instância de um dicionário do tipo especificado.
- template<typename Key, typename Value>
  std::unique_ptr< Dictionary< Key, Value > > create_dictionary (const DictionaryType &type, const std↩
  ::initializer_list< std::pair< Key, Value > > &list)

  Cria uma instância de um dicionário do tipo especificado, inicializando-o com uma lista de pares chave-valor.

6.13.1 Documentação das funções

6.13.1.1 create_dictionary() [1/2]

```
template<typename Key, typename Value>
std::unique_ptr< Dictionary< Key, Value > > create_dictionary (
            const DictionaryType & type,
            const std::initializer_list< std::pair< Key, Value > > & list)
```

Cria uma instância de um dicionário do tipo especificado, inicializando-o com uma lista de pares chave-valor.
Esta função atua como uma fábrica (factory) para criar diferentes implementações de dicionários que herdam da classe base `Dictionary`, inicializando-os com os valores fornecidos. A implementação específica é escolhida com base no valor do enumerador `DictionaryType` fornecido.

Parâmetros de template

Key	O tipo das chaves no dicionário.
Value	O tipo dos valores associados às chaves.

Parâmetros

type	O tipo de dicionário a ser criado, conforme o enumerador `DictionaryType`.
list	Uma lista de pares chave-valor para inicializar o dicionário.

Retorna

Um `std::unique_ptr` para a nova instância do dicionário.

Excepções

std::invalid_argument	Lançada se o `type` fornecido for desconhecido ou ainda não implementado.

Definido na linha 56 do ficheiro DictionaryFactory.hpp.

```
00058 {
00059     switch (type)
00060     {
00061     case DictionaryType::AVL:
00062         return std::make_unique<AVLTree<Key, Value»(list);
00063     case DictionaryType::RBTREE:
00064         return std::make_unique<RedBlackTree<Key, Value»(list);
00065     case DictionaryType::CHAINING_HASH:
00066         return std::make_unique<ChainedHashTable<Key, Value»(list);
00067     case DictionaryType::OPEN_ADDRESSING_HASH:
00068         return std::make_unique<OpenHashTable<Key, Value»(list);
00069     default:
00070         throw std::invalid_argument("Tipo de dicionário desconhecido");
00071     }
00072 }
```

6.13.1.2 create_dictionary() [2/2]

```
template<typename Key, typename Value>
std::unique_ptr< Dictionary< Key, Value > > create_dictionary (
            const DictionaryType & type = DictionaryType::RBTREE)
```

Cria uma instância de um dicionário do tipo especificado.
Esta função atua como uma fábrica (factory) para criar diferentes implementações de dicionários que herdam da classe base `Dictionary`. A implementação específica é escolhida com base no valor do enumerador `DictionaryType` fornecido.

Parâmetros de template

Key	O tipo das chaves no dicionário.
Value	O tipo dos valores associados às chaves.

Parâmetros

type	O tipo de dicionário a ser criado, conforme o enumerador `DictionaryType`.

Retorna

Um `std::unique_ptr` para a nova instância do dicionário.

Excepções

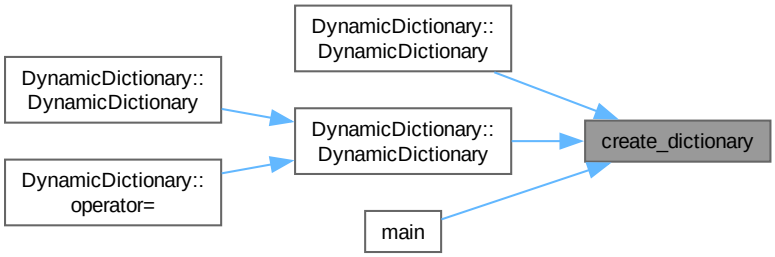
std::invalid_argument	Lançada se o `type` fornecido for desconhecido ou ainda não implementado.

Definido na linha 24 do ficheiro DictionaryFactory.hpp.

```
00025 {
00026     switch (type)
00027     {
00028     case DictionaryType::AVL:
00029         return std::make_unique<AVLTree<Key, Value»();
00030     case DictionaryType::RBTREE:
00031         return std::make_unique<RedBlackTree<Key, Value»();
00032     case DictionaryType::CHAINING_HASH:
00033         return std::make_unique<ChainedHashTable<Key, Value»();
00034     case DictionaryType::OPEN_ADDRESSING_HASH:
```

```
00035         return std::make_unique<OpenHashTable<Key, Value»();
00036     default:
00037         throw std::invalid_argument("Tipo de dicionário desconhecido");
00038     }
00039 }
```

Este é o diagrama das funções que utilizam esta função:



6.14 DictionaryFactory.hpp

Ir para a documentação deste ficheiro.

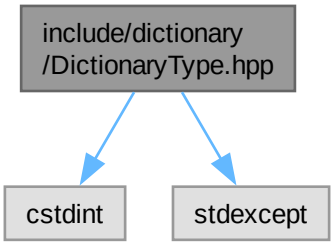
```
00001 #pragma once
00002
00003 #include "dictionary/avl_tree/AVLTree.hpp"
00004 #include "dictionary/rb_tree/RedBlackTree.hpp"
00005 #include "dictionary/hash_table_c/ChainedHashTable.hpp"
00006 #include "dictionary/hash_table_o/OpenHashTable.hpp"
00007 #include "dictionary/Dictionary.hpp"
00008 #include "dictionary/DictionaryType.hpp"
00009
00023 template <typename Key, typename Value>
00024 std::unique_ptr<Dictionary<Key, Value» create_dictionary(const DictionaryType &type =
      DictionaryType::RBTREE)
00025 {
00026     switch (type)
00027     {
00028     case DictionaryType::AVL:
00029         return std::make_unique<AVLTree<Key, Value»();
00030     case DictionaryType::RBTREE:
00031         return std::make_unique<RedBlackTree<Key, Value»();
00032     case DictionaryType::CHAINING_HASH:
00033         return std::make_unique<ChainedHashTable<Key, Value»();
00034     case DictionaryType::OPEN_ADDRESSING_HASH:
00035         return std::make_unique<OpenHashTable<Key, Value»();
00036     default:
00037         throw std::invalid_argument("Tipo de dicionário desconhecido");
00038     }
00039 }
00040
00055 template <typename Key, typename Value>
00056 std::unique_ptr<Dictionary<Key, Value» create_dictionary(const DictionaryType &type,
00057                                                          const std::initializer_list<std::pair<Key,
      Value» &list)
00058 {
00059     switch (type)
00060     {
00061     case DictionaryType::AVL:
00062         return std::make_unique<AVLTree<Key, Value»(list);
00063     case DictionaryType::RBTREE:
00064         return std::make_unique<RedBlackTree<Key, Value»(list);
00065     case DictionaryType::CHAINING_HASH:
00066         return std::make_unique<ChainedHashTable<Key, Value»(list);
00067     case DictionaryType::OPEN_ADDRESSING_HASH:
00068         return std::make_unique<OpenHashTable<Key, Value»(list);
00069     default:
00070         throw std::invalid_argument("Tipo de dicionário desconhecido");
00071     }
00072 }
```

6.15 Referência ao ficheiro include/dictionary/DictionaryType.hpp

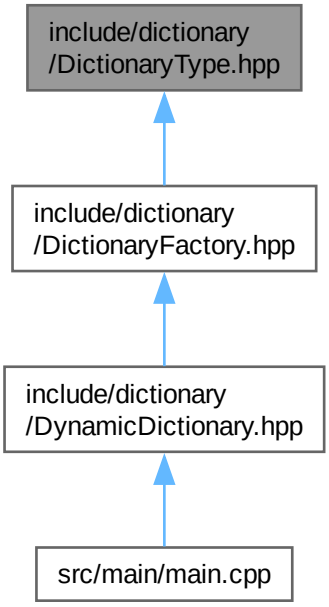
Enumera as estruturas de dados subjacentes disponíveis para uma implementação de dicionário.

```
#include <cstdint>
#include <stdexcept>
```

Diagrama de dependências de inclusão para DictionaryType.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Enumerações

- enum class DictionaryType : uint8_t { AVL , RBTREE , CHAINING_HASH , OPEN_ADDRESSING_HASH }

Funções

- std::string get_structure_name (DictionaryType type)

Obtém o nome da estrutura de dados a partir do tipo de dicionário.

- DictionaryType get_structure_type (const std::string &structure_type)

  Obtém o tipo de estrutura de dados a partir de uma string.

6.15.1 Descrição detalhada

Enumera as estruturas de dados subjacentes disponíveis para uma implementação de dicionário.
Isso permite a seleção de um tipo de dicionário específico em tempo de execução ou de compilação, cada um com diferentes características de desempenho em relação às operações de inserção, exclusão e busca.

- `AVL`: Uma árvore de busca binária auto-balanceada (árvore Adelson-Velsky e Landis).
- `RBTREE`: Uma árvore de busca binária auto-balanceada (Árvore Rubro-Negra).
- `CHAINING_HASH`: Uma tabela hash que resolve colisões por encadeamento.
- `OPEN_ADDRESSING_HASH`: Uma tabela hash que resolve colisões usando endereçamento aberto.

Definido no ficheiro DictionaryType.hpp.

6.15.2 Documentação dos valores da enumeração

6.15.2.1 DictionaryType

```
enum class DictionaryType :  uint8_t  [strong]
```

Valores de enumerações

AVL	Árvore AVL (Adelson-Velsky e Landis)
RBTREE	Árvore Rubro-Negra.
CHAINING_HASH	Tabela hash com encadeamento (Chained Hash Table)
OPEN_ADDRESSING_HASH	Tabela hash com endereçamento aberto (Open Addressing Hash Table)

Definido na linha 19 do ficheiro DictionaryType.hpp.

```
00020 {
00021     AVL,
00022     RBTREE,
00023     CHAINING_HASH,
00024     OPEN_ADDRESSING_HASH,
00025 };
```

6.15.3 Documentação das funções

6.15.3.1 get_structure_name()

```
std::string get_structure_name (
            DictionaryType type)
```

Obtém o nome da estrutura de dados a partir do tipo de dicionário.
Esta função converte um valor do enumerador `DictionaryType` em uma string representando o nome da estrutura de dados correspondente.

Parâmetros

type	O tipo de dicionário, conforme o enumerador `DictionaryType`.

Retorna

std::string O nome da estrutura de dados correspondente.

Definido na linha 36 do ficheiro DictionaryType.hpp.

```
00037 {
00038     switch (type)
00039     {
00040     case DictionaryType::AVL:
00041         return "AVL TREE";
00042     case DictionaryType::RBTREE:
00043         return "RB TREE";
00044     case DictionaryType::CHAINING_HASH:
00045         return "CHAINING_HASH";
00046     case DictionaryType::OPEN_ADDRESSING_HASH:
00047         return "OPEN_ADDRESSING_HASH";
00048     default:
00049         throw std::invalid_argument("Tipo de estrutura desconhecido");
00050     }
00051 }
```

Este é o diagrama das funções que utilizam esta função:

6.15.3.2 get_structure_type()

```
DictionaryType get_structure_type (
            const std::string & structure_type)
```

Obtém o tipo de estrutura de dados a partir de uma string.
Esta função converte uma string representando o tipo de estrutura de dados em um valor do enum DictionaryType correspondente.

Parâmetros

structure_type	A string representando o tipo da estrutura de dados.

Retorna

DictionaryType O tipo da estrutura de dados correspondente.

Definido na linha 62 do ficheiro DictionaryType.hpp.

```
00063 {
00064     if (structure_type == "avl" or structure_type == "avltree")
00065         return DictionaryType::AVL;
00066     else if (structure_type == "rbt" or structure_type == "rbtree")
00067         return DictionaryType::RBTREE;
00068     else if (structure_type == "chash" or structure_type == "hashtable")
00069         return DictionaryType::CHAINING_HASH;
00070     else if (structure_type == "ohash" or structure_type == "openhashtable")
00071         return DictionaryType::OPEN_ADDRESSING_HASH;
00072     else
00073         throw std::invalid_argument("Unknown structure_type type: " + structure_type);
00074 }
```

Este é o diagrama das funções que utilizam esta função:

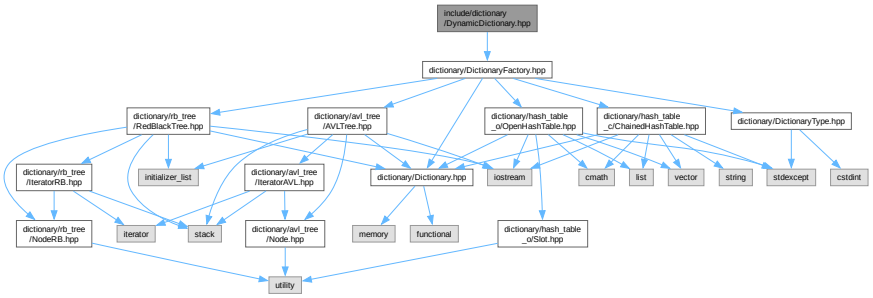
6.16 DictionaryType.hpp

Ir para a documentação deste ficheiro.

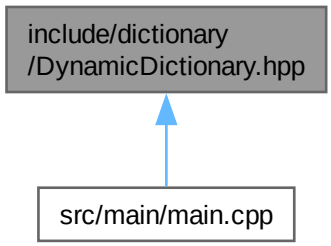
```
00001 #pragma once
00002
00003 #include <cstdint>
00004 #include <stdexcept>
00005
00019 enum class DictionaryType : uint8_t
00020 {
00021     AVL,
00022     RBTREE,
00023     CHAINING_HASH,
00024     OPEN_ADDRESSING_HASH,
00025 };
00026
00036 std::string get_structure_name(DictionaryType type)
00037 {
00038     switch (type)
00039     {
00040     case DictionaryType::AVL:
00041         return "AVL TREE";
00042     case DictionaryType::RBTREE:
00043         return "RB TREE";
00044     case DictionaryType::CHAINING_HASH:
00045         return "CHAINING_HASH";
00046     case DictionaryType::OPEN_ADDRESSING_HASH:
00047         return "OPEN_ADDRESSING_HASH";
00048     default:
00049         throw std::invalid_argument("Tipo de estrutura desconhecido");
00050     }
00051 }
00052
00062 DictionaryType get_structure_type(const std::string &structure_type)
00063 {
00064     if (structure_type == "avl" or structure_type == "avltree")
00065         return DictionaryType::AVL;
00066     else if (structure_type == "rbt" or structure_type == "rbtree")
00067         return DictionaryType::RBTREE;
00068     else if (structure_type == "chash" or structure_type == "hashtable")
00069         return DictionaryType::CHAINING_HASH;
00070     else if (structure_type == "ohash" or structure_type == "openhashtable")
00071         return DictionaryType::OPEN_ADDRESSING_HASH;
00072     else
00073         throw std::invalid_argument("Unknown structure_type type: " + structure_type);
00074 }
```

6.17 Referência ao ficheiro include/dictionary/DynamicDictionary.hpp

`#include "dictionary/DictionaryFactory.hpp"`
Diagrama de dependências de inclusão para DynamicDictionary.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class DynamicDictionary< Key, Value >

  Uma classe de dicionário dinâmico que atua como um wrapper (invólucro).

6.18 DynamicDictionary.hpp

Ir para a documentação deste ficheiro.

```
00001 #pragma once
00002
00003 #include "dictionary/DictionaryFactory.hpp"
00004
00016 template <typename Key, typename Value>
00017 class DynamicDictionary : public Dictionary<Key, Value>
00018 {
00019 private:
00020     std::unique_ptr<Dictionary<Key, Value» dictionary{nullptr}; // Ponteiro inteligente para a
      implementação do dicionário subjacente
00021     DictionaryType type;                                        // O tipo de dicionário a ser
      utilizado
00022
00027     void check_dictionary() const
00028     {
00029         if (!dictionary)
00030             throw std::runtime_error("Falha ao criar o dicionário. Verifique o tipo especificado.");
00031     }
00032
00033 public:
```

```
00039     DynamicDictionary(DictionaryType type = DictionaryType::RBTREE) : type(type),
      dictionary(create_dictionary<Key, Value>(type))
00040     {
00041         check_dictionary();
00042     }
00043
00052     DynamicDictionary(const DynamicDictionary &other) : type(other.type),
      dictionary(other.dictionary->clone())
00053     {
00054         check_dictionary();
00055     }
00056
00062     DynamicDictionary(const std::initializer_list<std::pair<Key, Value» list, DictionaryType type =
      DictionaryType::RBTREE)
00063         : type(type), dictionary(create_dictionary<Key, Value>(type, list))
00064     {
00065         check_dictionary();
00066     }
00067
00072     std::unique_ptr<Dictionary<Key, Value» clone() const
00073     {
00074         return std::make_unique<DynamicDictionary<Key, Value»(*this);
00075     }
00076
00084     void insert(const std::pair<Key, Value> &key_value)
00085     {
00086         dictionary->insert(key_value);
00087     }
00088
00096     void remove(const Key &key)
00097     {
00098         dictionary->remove(key);
00099     }
00100
00108     void update(const std::pair<Key, Value> &key_value)
00109     {
00110         dictionary->update(key_value);
00111     }
00112
00118     bool contains(const Key &key)
00119     {
00120         return dictionary->contains(key);
00121     }
00122
00132     Value &at(const Key &key)
00133     {
00134         return dictionary->at(key);
00135     }
00136
00145     DynamicDictionary &operator=(const DynamicDictionary &other)
00146     {
00147         if (this != &other)
00148         {
00149             dictionary = other.dictionary->clone;
00150         }
00151
00152         return *this;
00153     }
00154
00163     Value &operator[](const Key &key)
00164     {
00165         return dictionary->operator[](key);
00166     }
00167
00171     void clear()
00172     {
00173         dictionary->clear();
00174     }
00175
00180     size_t size() const noexcept
00181     {
00182         return dictionary->size();
00183     }
00184
00189     bool empty() const noexcept
00190     {
00191         return dictionary->empty();
00192     }
00193
00199     void print() const
00200     {
00201         dictionary->print();
00202     }
00203
00208     void forEach(const std::function<void(const std::pair<Key, Value> &)> &func) const
00209     {
00210         dictionary->forEach(func);
```

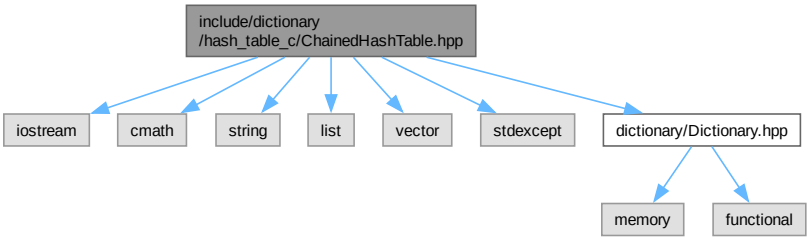
```
00211     }
00212
00217     Dictionary<Key, Value> &get_dictionary() const
00218     {
00219         return *dictionary;
00220     }
00221 };
```

6.19 Referência ao ficheiro include/dictionary/hash_table_c/ChainedHashTable.hpp

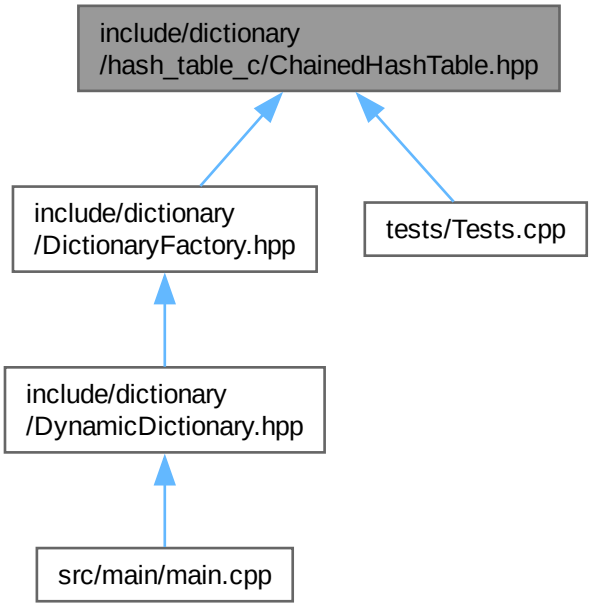
Implementação de um dicionário utilizando uma tabela hash com encadeamento separado.

```
#include <iostream>
#include <cmath>
#include <string>
#include <list>
#include <vector>
#include <stdexcept>
#include "dictionary/Dictionary.hpp"
```

Diagrama de dependências de inclusão para ChainedHashTable.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class ChainedHashTable< Key, Value, Hash >

6.19.1 Descrição detalhada

Implementação de um dicionário utilizando uma tabela hash com encadeamento separado.
`ChainedHashTable` gerencia uma coleção de pares chave-valor, oferecendo acesso rápido aos elementos. A resolução de colisões é feita através de encadeamento, onde cada "bucket" (ou slot) da tabela pode conter uma lista de elementos que mapeiam para o mesmo índice. A tabela é redimensionada automaticamente (rehashing) quando o fator de carga (número de elementos / tamanho da tabela) excede um valor máximo definido.

Parâmetros de template

Key	O tipo dos objetos que funcionam como chaves.
Value	O tipo dos objetos que funcionam como valores.
Hash	Um objeto de função que calcula o código hash para uma chave. Por padrão, utiliza `std::hash<Key>`.

Definido no ficheiro ChainedHashTable.hpp.

6.20 ChainedHashTable.hpp

Ir para a documentação deste ficheiro.

```
00001 #pragma once
00002
00003 #include <iostream>
00004 #include <cmath>
00005 #include <string>
```

```
00006 #include <list>
00007 #include <vector>
00008 #include <stdexcept>
00009
00010 #include "dictionary/Dictionary.hpp"
00011
00028 template <typename Key, typename Value, typename Hash = std::hash<Key»
00029 class ChainedHashTable : public Dictionary<Key, Value>
00030 {
00031 private:
00032     // Quantidade de pares (chave, valor)
00033     size_t m_number_of_elements;
00034
00035     // Tamanho atual da tabela
00036     size_t m_table_size;
00037
00038     // O maior valor que o fator de carga pode ter.
00039     // Seja load_factor = m_number_of_elements/m_table_size.
00040     // Temos que load_factor <= m_max_load_factor.
00041     // Quando load_factor ultrapassa o valor de m_max_load_factor,
00042     // eh preciso executar a operacao de rehashing.
00043     float m_max_load_factor;
00044
00045     // tabela
00046     std::vector<std::list<std::pair<Key, Value»> m_table;
00047
00048     // referencia para a funcao de codificacao
00049     Hash m_hashing;
00050
00051     mutable long long comparisons{0}; // contador de comparações para análise de desempenho
00052
00053     mutable long long collisions{0}; // contador de colisões para análise de desempenho
00054
00065     size_t get_next_prime(size_t x);
00066
00078     size_t hash_code(const Key &k) const;
00079
00080 public:
00088     ChainedHashTable(const size_t &tableSize = 19, const float &load_factor = 1.0f);
00089
00097     ChainedHashTable(const std::initializer_list<std::pair<Key, Value» &list, const size_t &tableSize
      = 19, const float &load_factor = 1.0f);
00098
00104     std::unique_ptr<Dictionary<Key, Value» clone() const;
00105
00114     long long getComparisons() const noexcept { return comparisons; }
00115
00124     long long getCollisions() const noexcept { return collisions; }
00125
00130     size_t size() const noexcept;
00131
00136     bool empty() const noexcept;
00137
00142     size_t bucket_count() const noexcept;
00143
00151     size_t bucket_size(size_t n) const;
00152
00159     size_t bucket(const Key &k) const;
00160
00164     void clear();
00165
00171     float load_factor() const noexcept;
00172
00178     float max_load_factor() const noexcept;
00179
00183     ~ChainedHashTable() = default;
00184
00194     void insert(const std::pair<Key, Value> &key_value);
00195
00206     void update(const std::pair<Key, Value> &key_value);
00207
00214     bool contains(const Key &k);
00215
00225     Value &at(const Key &k);
00226
00236     const Value &at(const Key &k) const;
00237
00247     void rehash(size_t m);
00248
00258     void remove(const Key &k);
00259
00269     void reserve(size_t n) noexcept;
00270
00280     void set_max_load_factor(float lf);
00281
00292     Value &operator[](const Key &k);
00293
```

```
00303     const Value &operator[](const Key &k) const;
00304
00310     void print() const;
00311
00320     void forEach(const std::function<void(const std::pair<Key, Value> &)> &func) const;
00321 };
00322
00323
      //-----------------------------------------------------------IMPLEMENTAÇÕES-----------------------------------------
00324
00325 template <typename Key, typename Value, typename Hash>
00326 size_t ChainedHashTable<Key, Value, Hash>::get_next_prime(size_t x)
00327 {
00328     if (x <= 2)
00329         return 3;
00330
00331     x = (x % 2 == 0) ? x + 1 : x;
00332     bool not_prime = true;
00333
00334     while (not_prime)
00335     {
00336         not_prime = false;
00337         for (int i = 3; i <= sqrt(x); i += 2)
00338         {
00339             if (x % i == 0)
00340             {
00341                 not_prime = true;
00342                 break;
00343             }
00344         }
00345         x += 2;
00346     }
00347
00348     return x - 2;
00349 }
00350
00351 template <typename Key, typename Value, typename Hash>
00352 size_t ChainedHashTable<Key, Value, Hash>::hash_code(const Key &k) const
00353 {
00354     return m_hashing(k) % m_table_size;
00355 }
00356
00357 template <typename Key, typename Value, typename Hash>
00358 ChainedHashTable<Key, Value, Hash>::ChainedHashTable(const size_t &tableSize, const float
      &load_factor) : m_number_of_elements(0), m_table_size(tableSize)
00359 {
00360     m_table.resize(m_table_size);
00361
00362     if (load_factor <= 0)
00363         m_max_load_factor = 1.0f;
00364     else
00365         m_max_load_factor = load_factor;
00366 }
00367
00368 template <typename Key, typename Value, typename Hash>
00369 ChainedHashTable<Key, Value, Hash>::ChainedHashTable(const std::initializer_list<std::pair<Key, Value»
      &list, const size_t &tableSize, const float &load_factor) : ChainedHashTable(tableSize, load_factor)
00370 {
00371     for (const auto &pair : list)
00372         insert(pair);
00373 }
00374
00375 template <typename Key, typename Value, typename Hash>
00376 std::unique_ptr<Dictionary<Key, Value» ChainedHashTable<Key, Value, Hash>::clone() const
00377 {
00378     return std::make_unique<ChainedHashTable<Key, Value, Hash»(*this);
00379 }
00380
00381 template <typename Key, typename Value, typename Hash>
00382 size_t ChainedHashTable<Key, Value, Hash>::size() const noexcept
00383 {
00384     return m_number_of_elements;
00385 }
00386
00387 template <typename Key, typename Value, typename Hash>
00388 bool ChainedHashTable<Key, Value, Hash>::empty() const noexcept
00389 {
00390     return m_number_of_elements == 0;
00391 }
00392
00393 template <typename Key, typename Value, typename Hash>
00394 size_t ChainedHashTable<Key, Value, Hash>::bucket_count() const noexcept
00395 {
00396     return m_table_size;
00397 }
00398
00399 template <typename Key, typename Value, typename Hash>
```

```
00400 size_t ChainedHashTable<Key, Value, Hash>::bucket_size(size_t n) const
00401 {
00402     if (n >= m_table_size)
00403         throw std::out_of_range("invalid index");
00404 
00405     return m_table[n].size();
00406 }
00407 
00408 template <typename Key, typename Value, typename Hash>
00409 size_t ChainedHashTable<Key, Value, Hash>::bucket(const Key &k) const
00410 {
00411     return hash_code(k);
00412 }
00413 
00414 template <typename Key, typename Value, typename Hash>
00415 float ChainedHashTable<Key, Value, Hash>::load_factor() const noexcept
00416 {
00417     return static_cast<float>(m_number_of_elements) / m_table_size;
00418 }
00419 
00420 template <typename Key, typename Value, typename Hash>
00421 float ChainedHashTable<Key, Value, Hash>::max_load_factor() const noexcept
00422 {
00423     return m_max_load_factor;
00424 }
00425 
00426 template <typename Key, typename Value, typename Hash>
00427 void ChainedHashTable<Key, Value, Hash>::clear()
00428 {
00429     for (size_t i = 0; i < m_table_size; ++i)
00430         m_table[i].clear();
00431 
00432     m_number_of_elements = 0;
00433 }
00434 
00435 template <typename Key, typename Value, typename Hash>
00436 void ChainedHashTable<Key, Value, Hash>::insert(const std::pair<Key, Value> &key_value)
00437 {
00438     if (load_factor() >= m_max_load_factor)
00439         rehash(m_table_size * 2);
00440 
00441     size_t hash_index = hash_code(key_value.first);
00442 
00443     for (const auto &pair : m_table[hash_index])
00444     {
00445         comparisons++;
00446         if (pair.first == key_value.first)
00447             return; // chave ja existe, nao adiciona
00448     }
00449 
00450     if (!m_table[hash_index].empty())
00451         collisions++;
00452 
00453     m_table[hash_index].push_back(key_value);
00454     m_number_of_elements++;
00455 }
00456 
00457 template <typename Key, typename Value, typename Hash>
00458 void ChainedHashTable<Key, Value, Hash>::update(const std::pair<Key, Value> &key_value)
00459 {
00460     size_t hash_index = hash_code(key_value.first);
00461 
00462     for (auto &pair : m_table[hash_index])
00463     {
00464         comparisons++;
00465         if (pair.first == key_value.first)
00466         {
00467             pair.second = key_value.second; // atualiza o valor associado a chave
00468             return;
00469         }
00470     }
00471 
00472     throw std::out_of_range("Key not found in the hash table");
00473 }
00474 
00475 template <typename Key, typename Value, typename Hash>
00476 bool ChainedHashTable<Key, Value, Hash>::contains(const Key &k)
00477 {
00478     for (auto &i : m_table[hash_code(k)])
00479     {
00480         comparisons++;
00481         if (i.first == k)
00482             return true;
00483     }
00484     return false;
00485 }
00486 
```

```
00487 template <typename Key, typename Value, typename Hash>
00488 Value &ChainedHashTable<Key, Value, Hash>::at(const Key &k)
00489 {
00490     size_t hash_index = hash_code(k);
00491 
00492     for (auto &pair : m_table[hash_index])
00493     {
00494         comparisons++;
00495         if (pair.first == k)
00496             return pair.second; // retorna o valor associado a chave
00497     }
00498 
00499     throw std::out_of_range("Key not found in the hash table");
00500 }
00501 
00502 template <typename Key, typename Value, typename Hash>
00503 const Value &ChainedHashTable<Key, Value, Hash>::at(const Key &k) const
00504 {
00505     size_t hash_index = hash_code(k);
00506 
00507     for (const auto &pair : m_table[hash_index])
00508     {
00509         comparisons++;
00510         if (pair.first == k)
00511             return pair.second; // retorna o valor associado a chave
00512     }
00513 
00514     throw std::out_of_range("Key not found in the hash table");
00515 }
00516 
00517 template <typename Key, typename Value, typename Hash>
00518 void ChainedHashTable<Key, Value, Hash>::rehash(size_t m)
00519 {
00520     size_t new_table_size = get_next_prime(m);
00521 
00522     if (new_table_size > m_table_size)
00523     {
00524         std::vector<std::list<std::pair<Key, Value»> aux;
00525         m_table.swap(aux);
00526         m_table.resize(new_table_size);
00527 
00528         m_table_size = new_table_size;
00529         m_number_of_elements = 0;
00530 
00531         for (auto &vec : aux)
00532             for (auto &listas : vec)
00533                 insert({listas.first, listas.second});
00534     }
00535 }
00536 
00537 template <typename Key, typename Value, typename Hash>
00538 void ChainedHashTable<Key, Value, Hash>::remove(const Key &k)
00539 {
00540     size_t slot = hash_code(k); // calcula o slot em que estaria a chave
00541     for (auto it = m_table[slot].begin(); it != m_table[slot].end(); ++it)
00542     {
00543         comparisons++;
00544         if (it->first == k)
00545         {
00546             m_table[slot].erase(it); // se encontrar, deleta
00547             m_number_of_elements--;
00548             return; // sai da funcao apos remover
00549         }
00550     }
00551 }
00552 
00553 template <typename Key, typename Value, typename Hash>
00554 void ChainedHashTable<Key, Value, Hash>::reserve(size_t n) noexcept
00555 {
00556     if (n > m_table_size * m_max_load_factor)
00557         rehash(n / m_max_load_factor);
00558 }
00559 
00560 template <typename Key, typename Value, typename Hash>
00561 void ChainedHashTable<Key, Value, Hash>::set_max_load_factor(float lf)
00562 {
00563     if (lf <= 0)
00564         throw std::out_of_range("max load factor must be greater than 0");
00565 
00566     m_max_load_factor = lf;
00567 
00568     // Se o novo fator de carga for menor que o atual,
00569     // podemos precisar redimensionar a tabela.
00570     if (load_factor() > m_max_load_factor)
00571         reserve(m_number_of_elements);
00572 }
00573 
```

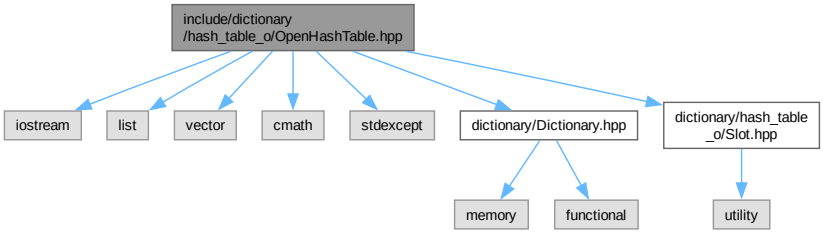
```
00574 template <typename Key, typename Value, typename Hash>
00575 Value &ChainedHashTable<Key, Value, Hash>::operator[](const Key &k)
00576 {
00577     if (load_factor() >= m_max_load_factor)
00578         rehash(2 * m_table_size);
00579
00580     size_t slot = hash_code(k);
00581
00582     for (auto &pair : m_table[slot])
00583     {
00584         comparisons++;
00585         if (pair.first == k)
00586             return pair.second; // retorna o valor associado a chave
00587     }
00588
00589     if (!m_table[slot].empty())
00590         collisions++;
00591
00592     m_table[slot].push_back({k, Value()}); // insere um novo elemento com valor padrão
00593     m_number_of_elements++;
00594     return m_table[slot].back().second; // retorna o valor associado a chave
00595 }
00596
00597 template <typename Key, typename Value, typename Hash>
00598 const Value &ChainedHashTable<Key, Value, Hash>::operator[](const Key &k) const
00599 {
00600     return at(k); // chama a funcao at para obter o valor associado a chave
00601 }
00602
00603 template <typename Key, typename Value, typename Hash>
00604 void ChainedHashTable<Key, Value, Hash>::print() const
00605 {
00606     forEach([](const std::pair<Key, Value> &par)
00607             { std::cout « "[" « par.first « ", " « par.second « "]" « std::endl; });
00608 }
00609
00610 template <typename Key, typename Value, typename Hash>
00611 void ChainedHashTable<Key, Value, Hash>::forEach(const std::function<void(const std::pair<Key, Value>
      &)> &func) const
00612 {
00613     for (const auto &bucket : m_table)
00614         for (const auto &pair : bucket)
00615             func(pair); // aplica a funcao a cada par chave-valor
00616 }
```

6.21 Referência ao ficheiro include/dictionary/hash_table_o/OpenHashTable.hpp

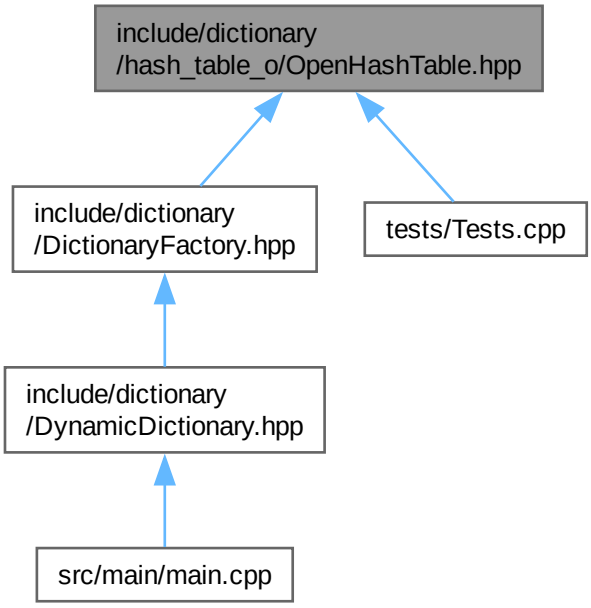
Implementação de uma tabela hash aberta (Open Hash Table).

```
#include <iostream>
#include <list>
#include <vector>
#include <cmath>
#include <stdexcept>
#include "dictionary/Dictionary.hpp"
#include "dictionary/hash_table_o/Slot.hpp"
```

Diagrama de dependências de inclusão para OpenHashTable.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class OpenHashTable< Key, Value, Hash >

6.21.1 Descrição detalhada

Implementação de uma tabela hash aberta (Open Hash Table).
Esta classe implementa uma tabela hash aberta, onde cada slot pode conter um par chave-valor. A tabela utiliza endereçamento aberto para resolver colisões, e permite inserção, remoção, atualização e busca de elementos. A tabela é redimensionada automaticamente quando o fator de carga ultrapassa um limite máximo definido pelo usuário.

Parâmetros de template

Key	Tipo da chave usada para indexação.
Value	Tipo do valor associado a cada chave.
Hash	Tipo da função de hash usada para calcular os índices.

Nota

A tabela é projetada para ser eficiente em termos de espaço e tempo, minimizando colisões e mantendo um bom desempenho em operações de inserção, busca e remoção.

Definido no ficheiro OpenHashTable.hpp.

6.22 OpenHashTable.hpp

Ir para a documentação deste ficheiro.

```
00001 #pragma once
00002
00003 #include <iostream>
00004 #include <list>
00005 #include <vector>
00006 #include <cmath>
00007 #include <stdexcept>
00008
00009 #include "dictionary/Dictionary.hpp"
00010 #include "dictionary/hash_table_o/Slot.hpp"
00011
00031 template <typename Key, typename Value, typename Hash = std::hash<Key»
00032 class OpenHashTable : public Dictionary<Key, Value>
00033 {
00034 private:
00035     // Quantidade de pares (chave, valor)
00036     size_t m_number_of_elements;
00037
00038     // Tamanho atual da tabela
00039     size_t m_table_size;
00040
00041     // O maior valor que o fator de carga pode ter.
00042     // Seja load_factor = m_number_of_elements/m_table_size.
00043     // Temos que load_factor <= m_max_load_factor.
00044     // Quando load_factor ultrapassa o valor de m_max_load_factor,
00045     // eh preciso executar a operacao de rehashing.
00046     float m_max_load_factor;
00047
00048     // tabela
00049     std::vector<Slot<Key, Value» m_table;
00050
00051     // referencia para a funcao de codificacao
00052     Hash m_hashing;
00053
00054     long long comparisons{0}; // contador de comparações para análise de desempenho
00055
00056     long long collisions{0}; // contador de colisões para análise de desempenho
00057
00068     size_t get_next_prime(size_t x);
00069
00083     size_t hash_code(const Key &k, const size_t &try_count = 0) const;
00084
00097     size_t findIndex(const Key &key);
00098
00099 public:
00107     OpenHashTable(const size_t &tableSize = 19, const float &load_factor = 0.5f);
00108
00116     OpenHashTable(const std::initializer_list<std::pair<Key, Value» &list, const size_t &tableSize =
      19, const float &load_factor = 0.5f);
00117
00123     std::unique_ptr<Dictionary<Key, Value» clone() const;
00124
00133     long long getComparisons() const noexcept { return comparisons; }
00134
00143     long long getCollisions() const noexcept { return collisions; }
00144
00149     size_t size() const noexcept;
00150
00155     bool empty() const noexcept;
00156
00161     size_t bucket_count() const noexcept;
00162
00169     size_t bucket(const Key &k) const;
00170
00174     void clear();
00175
00181     float load_factor() const noexcept;
00182
00188     float max_load_factor() const noexcept;
00189
00193     ~OpenHashTable() = default;
00194
00204     void insert(const std::pair<Key, Value> &key_value);
00205
00216     void update(const std::pair<Key, Value> &key_value);
00217
00224     bool contains(const Key &k);
00225
00235     Value &at(const Key &k);
00236
00246     const Value &at(const Key &k) const;
00247
00261     void rehash(size_t m);
00262
00272     void remove(const Key &k);
00273
00283     void reserve(size_t n) noexcept;
```

```
00284
00294     void set_max_load_factor(float lf);
00295
00306     Value &operator[](const Key &k);
00307
00317     const Value &operator[](const Key &k) const;
00318
00324     void print() const;
00325
00334     void forEach(const std::function<void(const std::pair<Key, Value> &)> &func) const;
00335 };
00336
00337
      //-----------------------------------------------------------IMPLEMENTAÇÕES-----------------------------------------------
00338
00339 template <typename Key, typename Value, typename Hash>
00340 size_t OpenHashTable<Key, Value, Hash>::get_next_prime(size_t x)
00341 {
00342     if (x <= 2)
00343         return 3;
00344
00345     x = (x % 2 == 0) ? x + 1 : x;
00346     bool not_prime = true;
00347
00348     while (not_prime)
00349     {
00350         not_prime = false;
00351         for (int i = 3; i <= sqrt(x); i += 2)
00352         {
00353             if (x % i == 0)
00354             {
00355                 not_prime = true;
00356                 break;
00357             }
00358         }
00359         x += 2;
00360     }
00361
00362     return x - 2;
00363 }
00364
00365 template <typename Key, typename Value, typename Hash>
00366 size_t OpenHashTable<Key, Value, Hash>::hash_code(const Key &k, const size_t &try_count) const
00367 {
00368     return (m_hashing(k) + (try_count * try_count)) % m_table_size;
00369 }
00370
00371 template <typename Key, typename Value, typename Hash>
00372 OpenHashTable<Key, Value, Hash>::OpenHashTable(const size_t &tableSize, const float &load_factor) :
      m_number_of_elements(0), m_table_size(tableSize)
00373 {
00374     m_table.resize(m_table_size);
00375
00376     if (load_factor <= 0)
00377         m_max_load_factor = 0.5f;
00378     else
00379         m_max_load_factor = load_factor;
00380 }
00381
00382 template <typename Key, typename Value, typename Hash>
00383 OpenHashTable<Key, Value, Hash>::OpenHashTable(const std::initializer_list<std::pair<Key, Value»
      &list, const size_t &tableSize, const float &load_factor) : OpenHashTable(tableSize, load_factor)
00384 {
00385     for (const auto &pair : list)
00386         insert(pair);
00387 }
00388
00389 template <typename Key, typename Value, typename Hash>
00390 std::unique_ptr<Dictionary<Key, Value» OpenHashTable<Key, Value, Hash>::clone() const
00391 {
00392     return std::make_unique<OpenHashTable<Key, Value, Hash»(*this);
00393 }
00394
00395 template <typename Key, typename Value, typename Hash>
00396 size_t OpenHashTable<Key, Value, Hash>::size() const noexcept
00397 {
00398     return m_number_of_elements;
00399 }
00400
00401 template <typename Key, typename Value, typename Hash>
00402 bool OpenHashTable<Key, Value, Hash>::empty() const noexcept
00403 {
00404     return m_number_of_elements == 0;
00405 }
00406
00407 template <typename Key, typename Value, typename Hash>
00408 size_t OpenHashTable<Key, Value, Hash>::bucket_count() const noexcept
```

```
00409 {
00410     return m_table_size;
00411 }
00412
00413 template <typename Key, typename Value, typename Hash>
00414 size_t OpenHashTable<Key, Value, Hash>::bucket(const Key &k) const
00415 {
00416     return hash_code(k);
00417 }
00418
00419 template <typename Key, typename Value, typename Hash>
00420 float OpenHashTable<Key, Value, Hash>::load_factor() const noexcept
00421 {
00422     return static_cast<float>(m_number_of_elements) / m_table_size;
00423 }
00424
00425 template <typename Key, typename Value, typename Hash>
00426 float OpenHashTable<Key, Value, Hash>::max_load_factor() const noexcept
00427 {
00428     return m_max_load_factor;
00429 }
00430
00431 template <typename Key, typename Value, typename Hash>
00432 void OpenHashTable<Key, Value, Hash>::clear()
00433 {
00434     m_table.clear();
00435     m_table.resize(m_table_size);
00436
00437     m_number_of_elements = 0;
00438 }
00439
00440 template <typename Key, typename Value, typename Hash>
00441 void OpenHashTable<Key, Value, Hash>::insert(const std::pair<Key, Value> &key_value)
00442 {
00443     if (load_factor() >= m_max_load_factor)
00444         rehash(m_table_size * 2);
00445
00446     size_t hash_index{(size_t)-1};
00447     size_t first_deleted_index{(size_t)-1};
00448
00449     for (size_t i = 0; i < m_table_size; i++)
00450     {
00451         size_t current_index = hash_code(key_value.first, i);
00452
00453         if (m_table[current_index].is_empty())
00454         {
00455             hash_index = current_index;
00456             break;
00457         }
00458         else if (m_table[current_index].is_active())
00459         {
00460             comparisons++;
00461             if (m_table[current_index].data.first == key_value.first)
00462                 return;
00463             collisions++;
00464         }
00465         else
00466         {
00467             if (first_deleted_index == (size_t)-1)
00468                 first_deleted_index = current_index;
00469         }
00470     }
00471
00472     if (first_deleted_index != (size_t)-1)
00473         hash_index = first_deleted_index;
00474
00475     if (hash_index == (size_t)-1)
00476         throw std::out_of_range("Hash table is full, cannot insert new element");
00477
00478     m_table[hash_index] = Slot<Key, Value>(key_value);
00479     m_number_of_elements++;
00480 }
00481
00482 template <typename Key, typename Value, typename Hash>
00483 void OpenHashTable<Key, Value, Hash>::update(const std::pair<Key, Value> &key_value)
00484 {
00485     size_t hash_index = findIndex(key_value.first);
00486
00487     if (hash_index != (size_t)-1)
00488         m_table[hash_index].data.second = key_value.second;
00489     else
00490         throw std::out_of_range("Key not found in the hash table");
00491 }
00492
00493 template <typename Key, typename Value, typename Hash>
00494 size_t OpenHashTable<Key, Value, Hash>::findIndex(const Key &key)
00495 {
```

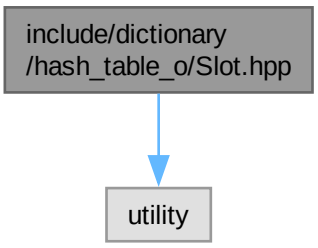
```
00496     size_t hash_index{(size_t)-1};
00497
00498     for (size_t i = 0; i < m_table_size; i++)
00499     {
00500         size_t current_index = hash_code(key, i);
00501
00502         if (m_table[current_index].is_empty())
00503             break;
00504
00505         comparisons++;
00506         if (m_table[current_index].is_active() and m_table[current_index].data.first == key)
00507         {
00508             hash_index = current_index;
00509             break;
00510         }
00511     }
00512
00513     return hash_index;
00514 }
00515
00516 template <typename Key, typename Value, typename Hash>
00517 bool OpenHashTable<Key, Value, Hash>::contains(const Key &k)
00518 {
00519     return findIndex(k) != (size_t)-1;
00520 }
00521
00522 template <typename Key, typename Value, typename Hash>
00523 Value &OpenHashTable<Key, Value, Hash>::at(const Key &k)
00524 {
00525     size_t hash_index = findIndex(k);
00526
00527     if (hash_index != (size_t)-1)
00528         return m_table[hash_index].data.second;
00529     else
00530         throw std::out_of_range("Key not found in the hash table");
00531 }
00532
00533 template <typename Key, typename Value, typename Hash>
00534 const Value &OpenHashTable<Key, Value, Hash>::at(const Key &k) const
00535 {
00536     size_t hash_index = findIndex(k);
00537
00538     if (hash_index != (size_t)-1)
00539         return m_table[hash_index].data.second;
00540     else
00541         throw std::out_of_range("Key not found in the hash table");
00542 }
00543
00544 template <typename Key, typename Value, typename Hash>
00545 void OpenHashTable<Key, Value, Hash>::rehash(size_t m)
00546 {
00547     size_t new_table_size = get_next_prime(m);
00548
00549     if (new_table_size > m_table_size)
00550     {
00551         std::vector<Slot<Key, Value» aux;
00552         m_table.swap(aux);
00553         m_table.resize(new_table_size);
00554
00555         m_table_size = new_table_size;
00556         m_number_of_elements = 0;
00557
00558         for (auto &slot : aux)
00559             if (slot.is_active())
00560                 insert({slot.data.first, slot.data.second});
00561     }
00562 }
00563
00564 template <typename Key, typename Value, typename Hash>
00565 void OpenHashTable<Key, Value, Hash>::remove(const Key &k)
00566 {
00567     size_t slot = findIndex(k); // calcula o slot em que estaria a chave
00568
00569     if (slot != (size_t)-1)
00570     {
00571         m_number_of_elements--;
00572         m_table[slot].status = HashTableStatus::DELETED;
00573     }
00574 }
00575
00576 template <typename Key, typename Value, typename Hash>
00577 void OpenHashTable<Key, Value, Hash>::reserve(size_t n) noexcept
00578 {
00579     if (n > m_table_size * m_max_load_factor)
00580         rehash(n / m_max_load_factor);
00581 }
00582
```

```
00583 template <typename Key, typename Value, typename Hash>
00584 void OpenHashTable<Key, Value, Hash>::set_max_load_factor(float lf)
00585 {
00586     if (lf <= 0)
00587         throw std::out_of_range("max load factor must be greater than 0");
00588
00589     m_max_load_factor = lf;
00590
00591     // Se o novo fator de carga for menor que o atual,
00592     // podemos precisar redimensionar a tabela.
00593     if (load_factor() > m_max_load_factor)
00594         reserve(m_number_of_elements);
00595 }
00596
00597 template <typename Key, typename Value, typename Hash>
00598 Value &OpenHashTable<Key, Value, Hash>::operator[](const Key &k)
00599 {
00600     size_t hash_index = findIndex(k);
00601
00602     if (hash_index != (size_t)-1)
00603         return m_table[hash_index].data.second;
00604     else
00605         insert({k, Value{}}); // insere um novo elemento com valor padrão
00606
00607     return m_table[findIndex(k)].data.second; // retorna o valor associado a chave
00608 }
00609
00610 template <typename Key, typename Value, typename Hash>
00611 const Value &OpenHashTable<Key, Value, Hash>::operator[](const Key &k) const
00612 {
00613     return at(k); // chama a funcao at para obter o valor associado a chave
00614 }
00615
00616 template <typename Key, typename Value, typename Hash>
00617 void OpenHashTable<Key, Value, Hash>::print() const
00618 {
00619     forEach([](const std::pair<Key, Value> &par)
00620             { std::cout « "[" « par.first « ", " « par.second « "]" « std::endl; });
00621 }
00622
00623 template <typename Key, typename Value, typename Hash>
00624 void OpenHashTable<Key, Value, Hash>::forEach(const std::function<void(const std::pair<Key, Value> &)>
      &func) const
00625 {
00626     for (const auto &slot : m_table)
00627         if (slot.is_active()) // verifica se o slot esta ativo
00628             func(slot.data);  // aplica a funcao a cada par chave-valor
00629 }
```

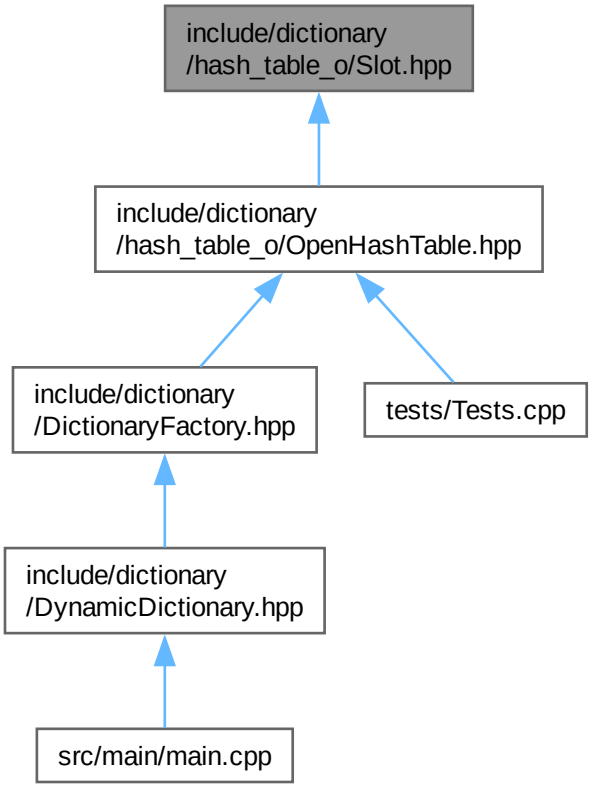
6.23 Referência ao ficheiro include/dictionary/hash_table_o/Slot.hpp

`#include <utility>`

Diagrama de dependências de inclusão para Slot.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class Slot< Key, Value >

  Representa um único slot em uma tabela hash de endereçamento aberto.

Enumerações

- enum class HashTableStatus : char { EMPTY , ACTIVE , DELETED }

  Enum que representa o estado de um slot na tabela hash.

6.23.1 Documentação dos valores da enumeração

6.23.1.1 HashTableStatus

```
enum class HashTableStatus : char [strong]
```

Enum que representa o estado de um slot na tabela hash.
Utilizado para controlar o estado de cada posição na tabela em esquemas de endereçamento aberto, permitindo diferenciar entre slots vazios, ocupados e removidos (lápides).

Valores de enumerações

EMPTY	O slot está vazio e nunca foi usado.
ACTIVE	O slot contém um par chave-valor ativo.

DELETED	O slot continha um par chave-valor que foi removido (lápide).

Definido na linha 13 do ficheiro Slot.hpp.

```
00014 {
00015     EMPTY,
00016     ACTIVE,
00017     DELETED
00018 };
```

6.24 Slot.hpp

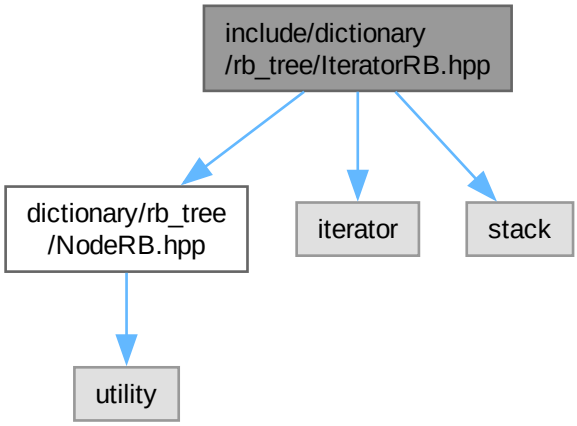
Ir para a documentação deste ficheiro.

```
00001 #pragma once
00002
00003 #include <utility>
00004
00013 enum class HashTableStatus : char
00014 {
00015     EMPTY,
00016     ACTIVE,
00017     DELETED
00018 };
00019
00031 template <typename Key, typename Value>
00032 struct Slot
00033 {
00038     std::pair<Key, Value> data{};
00039
00044     HashTableStatus status{HashTableStatus::EMPTY};
00045
00051     Slot() = default;
00052
00059     explicit Slot(const std::pair<Key, Value> &pair)
00060         : data(pair), status(HashTableStatus::ACTIVE) {}
00061
00069     Slot(const Key &key, const Value &value)
00070         : data({key, value}), status(HashTableStatus::ACTIVE) {}
00071
00076     bool is_empty() const noexcept
00077     {
00078         return status == HashTableStatus::EMPTY;
00079     }
00080
00085     bool is_active() const noexcept
00086     {
00087         return status == HashTableStatus::ACTIVE;
00088     }
00089
00094     bool is_deleted() const noexcept
00095     {
00096         return status == HashTableStatus::DELETED;
00097     }
00098 };
```

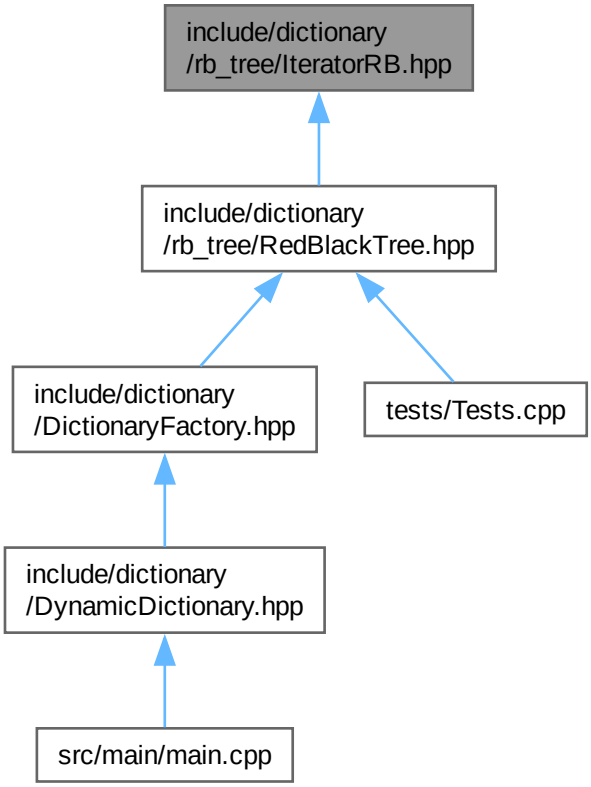
6.25 Referência ao ficheiro include/dictionary/rb_tree/IteratorRB.hpp

```
#include "dictionary/rb_tree/NodeRB.hpp"
#include <iterator>
#include <stack>
```

Diagrama de dependências de inclusão para IteratorRB.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class IteratorRB< Key, Value >

  Classe de iterador para a Red-Black Tree.

6.26 IteratorRB.hpp

Ir para a documentação deste ficheiro.

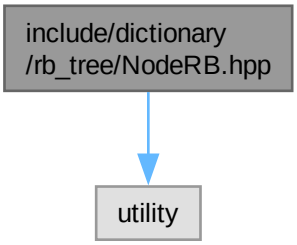
```
00001 #pragma once
00002
00003 #include "dictionary/rb_tree/NodeRB.hpp"
00004 #include <iterator>
00005 #include <stack>
00006
00007 template <typename Key, typename Value>
00008 class RBTree;
00009
00019 template <typename Key, typename Value>
00020 class IteratorRB
00021 {
00022 private:
00024     using NodePtr = NodeRB<Key, Value> *;
00025
00027     std::stack<NodePtr> path;
00028
00030     NodePtr nil{nullptr};
00031
00032 public:
00034     using iterator_category = std::input_iterator_tag;
00036     using value_type = std::pair<Key, Value>;
00038     using difference_type = std::ptrdiff_t;
00040     using pointer = value_type *;
00042     using reference = value_type &;
00044     using const_pointer = const value_type *;
00046     using const_reference = const value_type &;
00048     using NodeType = NodeRB<Key, Value>;
00050     using NodePtrType = NodePtr;
00051
00057     IteratorRB() = default;
00058
00067     IteratorRB(NodePtr root, NodePtr nil) : nil(nil)
00068     {
00069         NodePtr current = root;
00070         while (current != nil)
00071         {
00072             path.push(current);
00073             current = current->left;
00074         }
00075     }
00076
00084     reference operator*() const
00085     {
00086         return path.top()->key;
00087     }
00088
00096     pointer operator->() const
00097     {
00098         return &(path.top()->key);
00099     }
00100
00108     IteratorRB &operator++()
00109     {
00110         if (path.empty())
00111             return *this;
00112
00113         NodePtr node = path.top();
00114         path.pop();
00115
00116         if (node->right != nil)
00117         {
00118             NodePtr current = node->right;
00119
00120             while (current != nil)
00121             {
00122                 path.push(current);
00123                 current = current->left;
00124             }
00125         }
00126
00127         return *(this);
00128     }
00129
```

```
00138     IteratorRB operator++(int)
00139     {
00140         IteratorRB temp = *this;
00141         ++(*this);
00142         return temp;
00143     }
00144
00154     bool operator==(const IteratorRB &other) const
00155     {
00156         if (path.empty() && other.path.empty())
00157             return true;
00158
00159         if (path.empty() || other.path.empty())
00160             return false;
00161
00162         return path.top() == other.path.top();
00163     }
00164
00174     bool operator!=(const IteratorRB &other) const
00175     {
00176         return !(*this == other);
00177     }
00178 };
```

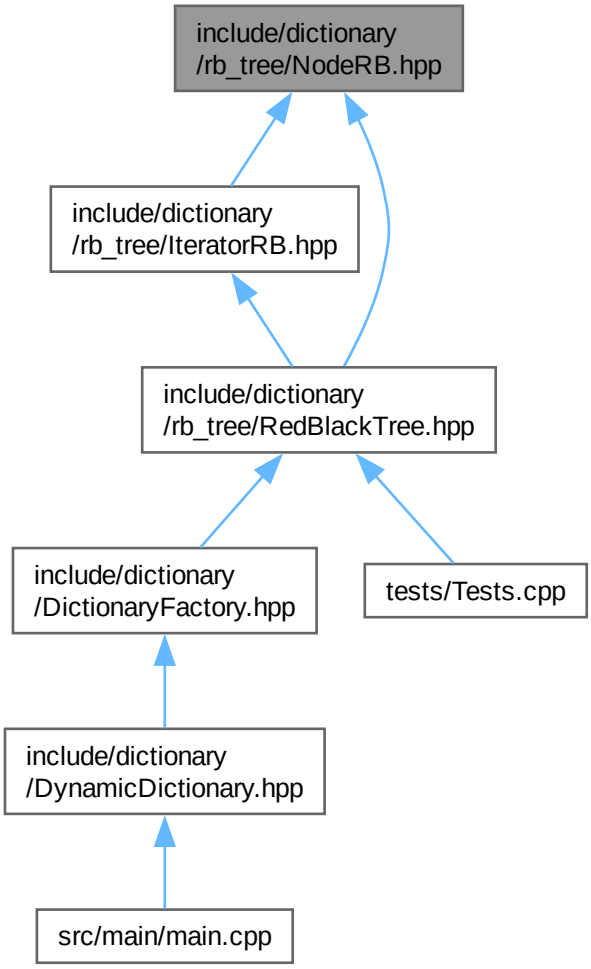
6.27 Referência ao ficheiro include/dictionary/rb_tree/NodeRB.hpp

`#include <utility>`

Diagrama de dependências de inclusão para NodeRB.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- struct NodeRB< Key, Value >

  Estrutura que representa um nó em uma Árvore Rubro-Negra (Red-Black Tree).

6.28 NodeRB.hpp

Ir para a documentação deste ficheiro.

```
00001 #pragma once
00002
00003 #include <utility>
00004
00024 template <typename Key, typename Value>
00025 struct NodeRB
00026 {
00031     constexpr static bool BLACK = false;
00032
00037     constexpr static bool RED = true;
00038
00042     std::pair<Key, Value> key;
00043
```

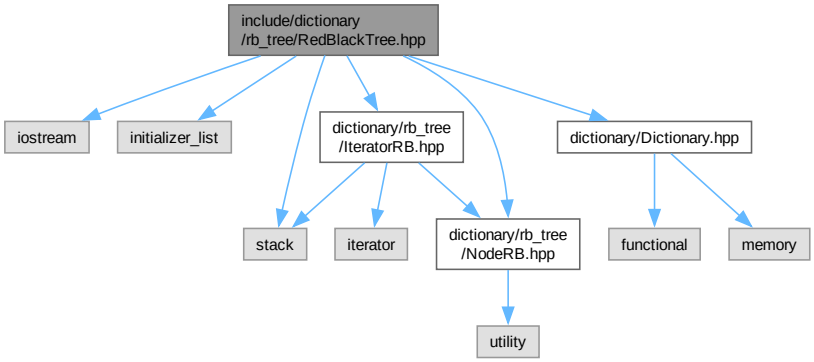
```
00052     bool color{RED};
00053
00058     NodeRB<Key, Value> *parent{nullptr};
00059
00064     NodeRB<Key, Value> *left{nullptr};
00065
00070     NodeRB<Key, Value> *right{nullptr};
00071
00081     NodeRB(const std::pair<Key, Value> &key, const bool &color, NodeRB<Key, Value> *parent,
      NodeRB<Key, Value> *left, NodeRB<Key, Value> *right)
00082         : key(key), color(color), parent(parent), left(left), right(right) {}
00083 };
```

6.29 Referência ao ficheiro include/dictionary/rb_tree/RedBlackTree.hpp

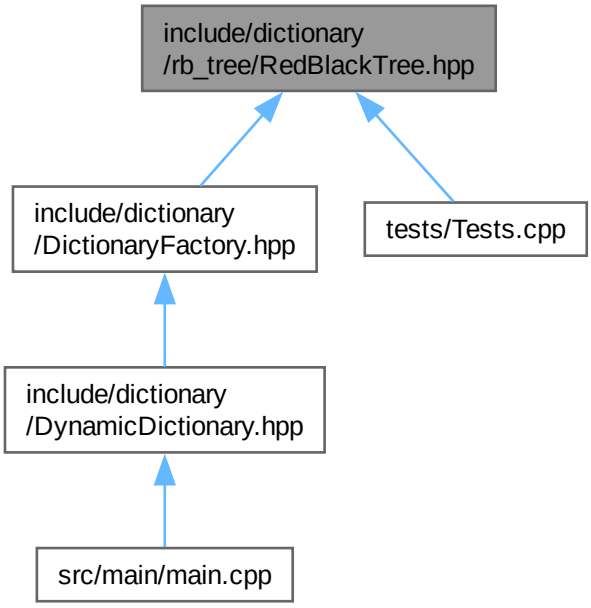
Implementação de uma Árvore Rubro-Negra (Red-Black Tree).

```
#include <iostream>
#include <initializer_list>
#include <stack>
#include "dictionary/rb_tree/NodeRB.hpp"
#include "dictionary/rb_tree/IteratorRB.hpp"
#include "dictionary/Dictionary.hpp"
```

Diagrama de dependências de inclusão para RedBlackTree.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class RedBlackTree< Key, Value >

6.29.1 Descrição detalhada

Implementação de uma Árvore Rubro-Negra (Red-Black Tree).
A Árvore Rubro-Negra é uma árvore de busca binária auto-balanceada que garante que as operações de inserção, remoção e busca tenham complexidade de tempo no pior caso de O(log n), onde n é o número de nós na árvore. Isso é alcançado através da manutenção de um conjunto de propriedades (propriedades Rubro-Negras) que mantêm a árvore aproximadamente balanceada.

Parâmetros de template

Key	O tipo da chave dos elementos.
Value	O tipo do valor associado a cada chave.

Definido no ficheiro RedBlackTree.hpp.

6.30 RedBlackTree.hpp

Ir para a documentação deste ficheiro.

```
00001 #pragma once
00002
00003 #include <iostream>
00004 #include <initializer_list>
00005 #include <stack>
00006
00007 #include "dictionary/rb_tree/NodeRB.hpp"
00008 #include "dictionary/rb_tree/IteratorRB.hpp"
```

```
00009 #include "dictionary/Dictionary.hpp"
00010
00025 template <typename Key, typename Value>
00026 class RedBlackTree : public Dictionary<Key, Value>
00027 {
00031     using NodePtr = NodeRB<Key, Value> *;
00032
00039     friend class IteratorRB<Key, Value>;
00040
00047     using iterator = IteratorRB<Key, Value>;
00048
00049 private:
00053     NodeRB<Key, Value> *root{nullptr};
00054
00062     NodeRB<Key, Value> *nil{nullptr};
00063
00067     size_t size_m{0};
00068
00074     long long comparisons{0};
00075
00082     long long rotations{0};
00083
00093     void fixup_node(NodePtr p);
00094
00106     NodeRB<Key, Value> *insert(NodePtr p, const NodePtr key);
00107
00117     void update(NodePtr p, const std::pair<Key, Value> &key);
00118
00129     Value &at(NodePtr p, const Key &key);
00130
00141     void fixup_deletion(NodePtr p);
00142
00152     void remove(NodePtr key);
00153
00163     NodeRB<Key, Value> *minimun(NodePtr node);
00164
00173     NodeRB<Key, Value> *clear(NodePtr root);
00174
00183     void rightRotation(NodePtr p);
00184
00193     void leftRotation(NodePtr p);
00194
00203     bool contains(NodePtr root, const Key &key);
00204
00214     NodeRB<Key, Value> *clone_recursive(const NodePtr parent_new, const NodePtr node_other, const
      NodePtr nil_other) const;
00215
00221     void printInOrder(NodePtr node) const;
00222
00229     void bshow(NodePtr node, std::string heranca);
00230
00231 public:
00238     RedBlackTree();
00239
00247     RedBlackTree(const RedBlackTree &other);
00248
00256     RedBlackTree(std::initializer_list<std::pair<Key, Value» list);
00257
00266     std::unique_ptr<Dictionary<Key, Value» clone() const;
00267
00271     ~RedBlackTree();
00272
00280     iterator begin() noexcept { return iterator(root, nil); }
00281
00289     iterator end() noexcept { return iterator(nil, nil); }
00290
00298     iterator begin() const noexcept { return iterator(root, nil); }
00299
00307     iterator end() const noexcept { return iterator(nil, nil); }
00308
00316     iterator cbegin() const noexcept { return iterator(root, nil); }
00317
00325     iterator cend() const noexcept { return iterator(nil, nil); }
00326
00335     void operator=(const RedBlackTree &other);
00336
00342     size_t size() const noexcept;
00343
00350     bool empty() const noexcept;
00351
00359     long long getComparisons() const noexcept { return comparisons; }
00360
00368     long long getRotations() const noexcept { return rotations; }
00369
00376     void clear();
00377
00385     void swap(RedBlackTree<Key, Value> &other) noexcept;
```

```
00386
00395     void insert(const std::pair<Key, Value> &key);
00396
00405     Value &at(const Key &key) { return at(root, key); };
00406
00417     Value &operator[](const Key &key);
00418
00427     void update(const std::pair<Key, Value> &key) { update(root, key); };
00428
00436     void operator=(std::pair<Key, Value> &key) { root = update(root, key); };
00437
00446     void remove(const Key &key);
00447
00455     bool contains(const Key &key);
00456
00457     // Funções de impressão
00458
00462     void print() const;
00463
00472     void forEach(const std::function<void(const std::pair<Key, Value> &)> &func) const;
00473
00479     void bshow();
00480 };
00481
00482 // ----------------------------------------Implementação da classe
      RedBlackTree.-----------------------------------------------------------------
00483
00484 template <typename Key, typename Value>
00485 RedBlackTree<Key, Value>::RedBlackTree()
00486 {
00487     nil = new NodeRB<Key, Value>({Key(), Value()}, NodeRB<Key, Value>::BLACK, nullptr, nullptr,
      nullptr);
00488     nil->left = nil->right = nil->parent = nil;
00489     root = nil;
00490     root->parent = nil;
00491 }
00492
00493 template <typename Key, typename Value>
00494 RedBlackTree<Key, Value>::RedBlackTree(std::initializer_list<std::pair<Key, Value» list) :
      RedBlackTree()
00495 {
00496     for (const auto &key : list)
00497         insert(key);
00498 }
00499
00500 template <typename Key, typename Value>
00501 RedBlackTree<Key, Value>::RedBlackTree(const RedBlackTree &other) : RedBlackTree()
00502 {
00503     if (other.root != other.nil)
00504     {
00505         root = clone_recursive(nil, other.root, other.nil);
00506         size_m = other.size_m;
00507         comparisons = other.comparisons;
00508         rotations = other.rotations;
00509     }
00510 }
00511
00512 template <typename Key, typename Value>
00513 std::unique_ptr<Dictionary<Key,
00514                            Value»
00515 RedBlackTree<Key, Value>::clone() const
00516 {
00517     return std::make_unique<RedBlackTree<Key, Value»(*this);
00518 }
00519
00520 template <typename Key, typename Value>
00521 RedBlackTree<Key, Value>::~RedBlackTree()
00522 {
00523     clear();
00524     delete nil; // Libera o nó nil (sentinela) após limpar a árvore
00525     nil = nullptr;
00526 }
00527
00528 template <typename Key, typename Value>
00529 void RedBlackTree<Key, Value>::operator=(const RedBlackTree &other)
00530 {
00531     if (this != &other)
00532     {
00533         clear();
00534         clone_recursive(nil, other.root, other.nil);
00535         size_m = other.size_m;
00536         comparisons = other.comparisons;
00537         rotations = other.rotations;
00538     }
00539 }
00540
00541 template <typename Key, typename Value>
```

```
00542 size_t RedBlackTree<Key, Value>::size() const noexcept
00543 {
00544     return size_m;
00545 }
00546
00547 template <typename Key, typename Value>
00548 bool RedBlackTree<Key, Value>::empty() const noexcept
00549 {
00550     return root == nil;
00551 }
00552
00553 template <typename Key, typename Value>
00554 NodeRB<Key, Value> *RedBlackTree<Key, Value>::clear(NodePtr root)
00555 {
00556     if (root != nil)
00557     {
00558         root->left = clear(root->left);
00559         root->right = clear(root->right);
00560
00561         delete root;
00562         return nil; // Retorna o nó nil após limpar a subárvore
00563     }
00564
00565     return root;
00566 }
00567
00568 template <typename Key, typename Value>
00569 void RedBlackTree<Key, Value>::clear()
00570 {
00571     root = clear(root);
00572     size_m = 0;
00573 }
00574
00575 template <typename Key, typename Value>
00576 void RedBlackTree<Key, Value>::swap(RedBlackTree<Key, Value> &other) noexcept
00577 {
00578     std::swap(root, other.root);
00579     std::swap(size_m, other.size_m);
00580     std::swap(nil, other.nil);
00581     std::swap(comparisons, other.comparisons);
00582     std::swap(rotations, other.rotations);
00583 }
00584
00585 template <typename Key, typename Value>
00586 void RedBlackTree<Key, Value>::fixup_node(NodePtr p)
00587 {
00588     while (p != root && p->parent->color == NodeRB<Key, Value>::RED)
00589     {
00590         if (p->parent == p->parent->parent->left)
00591         {
00592             NodePtr uncle = p->parent->parent->right;
00593
00594             if (uncle->color == NodeRB<Key, Value>::RED)
00595             {
00596                 p->parent->color = NodeRB<Key, Value>::BLACK;
00597                 uncle->color = NodeRB<Key, Value>::BLACK;
00598                 p->parent->parent->color = NodeRB<Key, Value>::RED;
00599                 p = p->parent->parent;
00600             }
00601             else
00602             {
00603                 if (p == p->parent->right)
00604                 {
00605                     p = p->parent;
00606                     leftRotation(p);
00607                 }
00608
00609                 p->parent->color = NodeRB<Key, Value>::BLACK;
00610                 p->parent->parent->color = NodeRB<Key, Value>::RED;
00611                 rightRotation(p->parent->parent);
00612             }
00613         }
00614         else
00615         {
00616             NodePtr uncle = p->parent->parent->left;
00617
00618             if (uncle->color == NodeRB<Key, Value>::RED)
00619             {
00620                 p->parent->color = NodeRB<Key, Value>::BLACK;
00621                 uncle->color = NodeRB<Key, Value>::BLACK;
00622                 p->parent->parent->color = NodeRB<Key, Value>::RED;
00623                 p = p->parent->parent;
00624             }
00625             else
00626             {
00627                 if (p == p->parent->left)
00628                 {
```

```
00629                     p = p->parent;
00630                     rightRotation(p);
00631                 }
00632 
00633                 p->parent->color = NodeRB<Key, Value>::BLACK;
00634                 p->parent->parent->color = NodeRB<Key, Value>::RED;
00635                 leftRotation(p->parent->parent);
00636             }
00637         }
00638     }
00639 
00640     root->color = NodeRB<Key, Value>::BLACK; // A raiz sempre deve ser preta
00641 }
00642 
00643 template <typename Key, typename Value>
00644 NodeRB<Key, Value> *RedBlackTree<Key, Value>::insert(NodePtr p, const NodePtr key)
00645 {
00646     NodePtr aux = p;
00647     NodePtr parent = nil;
00648 
00649     while (aux != nil)
00650     {
00651         comparisons++;
00652         parent = aux;
00653         if (key->key.first < aux->key.first)
00654             aux = aux->left;
00655         else if (key->key.first > aux->key.first)
00656         {
00657             comparisons++;
00658             aux = aux->right;
00659         }
00660         else
00661         {
00662             comparisons += 2;
00663             return p; // A chave já existe, não insere novamente
00664         }
00665     }
00666     size_m++;
00667 
00668     key->parent = parent;
00669 
00670     if (parent == nil)
00671     {
00672         root = key; // Se a árvore estava vazia, o novo nó se torna a raiz
00673     }
00674     else if (key->key.first < parent->key.first)
00675     {
00676         comparisons++;
00677         parent->left = key;
00678     }
00679     else
00680     {
00681         comparisons += 2;
00682         parent->right = key;
00683     }
00684 
00685     key->left = nil;
00686     key->right = nil;
00687 
00688     fixup_node(key);
00689 
00690     return root; // Retorna a nova raiz da árvore
00691 }
00692 
00693 template <typename Key, typename Value>
00694 void RedBlackTree<Key, Value>::insert(const std::pair<Key, Value> &key)
00695 {
00696     root = insert(root, new NodeRB<Key, Value>(key, NodeRB<Key, Value>::RED, nil, nil, nil));
00697 }
00698 
00699 template <typename Key, typename Value>
00700 void RedBlackTree<Key, Value>::remove(const Key &key)
00701 {
00702     NodePtr aux = root;
00703 
00704     while (aux != nil and aux->key.first != key)
00705     {
00706         comparisons++;
00707         if (key < aux->key.first)
00708             aux = aux->left;
00709         else
00710         {
00711             aux = aux->right;
00712         }
00713     }
00714 
00715     if (aux != nil) // Realiza a remoção se a chave for encontrada
```

```
00716         remove(aux);
00717
00718     // Se a chave não for encontrada, não faz nada
00719 }
00720
00721 template <typename Key, typename Value>
00722 void RedBlackTree<Key, Value>::fixup_deletion(NodePtr x)
00723 {
00724     while (x != root and x->color == NodeRB<Key, Value>::BLACK)
00725     {
00726         if (x == x->parent->left) // Se x for filho esquerdo
00727         {
00728             NodePtr w = x->parent->right;            // Irmão de x
00729             if (w->color == NodeRB<Key, Value>::RED) // Caso 1
00730             {
00731                 w->color = NodeRB<Key, Value>::BLACK;       // Muda a cor do irmão para preto
00732                 x->parent->color = NodeRB<Key, Value>::RED; // Muda a cor do pai para vermelho
00733                 leftRotation(x->parent);                    // Rotação à esquerda
00734                 w = x->parent->right;                       // Atualiza w
00735             }
00736
00737             if (w->left->color == NodeRB<Key, Value>::BLACK and
00738                 w->right->color == NodeRB<Key, Value>::BLACK) // Caso 2
00739             {
00740                 w->color = NodeRB<Key, Value>::RED; // Muda a cor do irmão para vermelho
00741                 x = x->parent;                      // Move x para o pai
00742             }
00743             else
00744             {
00745                 if (w->right->color == NodeRB<Key, Value>::BLACK) // Caso 3
00746                 {
00747                     w->left->color = NodeRB<Key, Value>::BLACK; // Muda a cor do filho esquerdo do
    irmão para preto
00748                     w->color = NodeRB<Key, Value>::RED;         // Muda a cor do irmão para vermelho
00749                     rightRotation(w);                           // Rotação à direita
00750                     w = x->parent->right;                       // Atualiza w
00751                 }
00752
00753                 w->color = x->parent->color; // Caso 4
00754                 x->parent->color = NodeRB<Key, Value>::BLACK;
00755                 w->right->color = NodeRB<Key, Value>::BLACK;
00756                 leftRotation(x->parent);
00757                 x = root; // Termina o loop
00758             }
00759         }
00760         else
00761         {
00762             NodePtr w = x->parent->left;             // Irmão de x
00763             if (w->color == NodeRB<Key, Value>::RED) // Caso 1
00764             {
00765                 w->color = NodeRB<Key, Value>::BLACK;       // Muda a cor do irmão para preto
00766                 x->parent->color = NodeRB<Key, Value>::RED; // Muda a cor do pai para vermelho
00767                 rightRotation(x->parent);                   // Rotação à direita
00768                 w = x->parent->left;                        // Atualiza w
00769             }
00770
00771             if (w->right->color == NodeRB<Key, Value>::BLACK and
00772                 w->left->color == NodeRB<Key, Value>::BLACK) // Caso 2
00773             {
00774                 w->color = NodeRB<Key, Value>::RED; // Muda a cor do irmão para vermelho
00775                 x = x->parent;                      // Move x para o pai
00776             }
00777             else
00778             {
00779                 if (w->left->color == NodeRB<Key, Value>::BLACK) // Caso 3
00780                 {
00781                     w->right->color = NodeRB<Key, Value>::BLACK; // Muda a cor do filho direito do
    irmão para preto
00782                     w->color = NodeRB<Key, Value>::RED;          // Muda a cor do irmão para vermelho
00783                     leftRotation(w);                             // Rotação à esquerda
00784                     w = x->parent->left;                         // Atualiza w
00785                 }
00786
00787                 w->color = x->parent->color; // Caso 4
00788                 x->parent->color = NodeRB<Key, Value>::BLACK;
00789                 w->left->color = NodeRB<Key, Value>::BLACK;
00790                 rightRotation(x->parent);
00791                 x = root; // Termina o loop
00792             }
00793         }
00794     }
00795     x->color = NodeRB<Key, Value>::BLACK; // Garante que a raiz seja preta
00796 }
00797
00798 template <typename Key, typename Value>
00799 void RedBlackTree<Key, Value>::remove(NodePtr key)
00800 {
```

```
00801     NodePtr aux{nullptr};
00802     NodePtr aux2{nullptr};
00803 
00804     if (key->left == nil or key->right == nil)
00805     {
00806         aux = key;
00807     }
00808     else
00809     {
00810         aux = minimun(key->right); // Encontra o sucessor (menor na subárvore direita)
00811     }
00812     if (aux->left != nil)
00813     {
00814         aux2 = aux->left;
00815     }
00816     else
00817     {
00818         aux2 = aux->right;
00819     }
00820     aux2->parent = aux->parent;
00821 
00822     if (aux->parent == nil)
00823     {
00824         root = aux2; // Se o nó removido era a raiz, atualiza a raiz
00825     }
00826     else if (aux == aux->parent->left)
00827     {
00828         aux->parent->left = aux2;
00829     }
00830     else
00831     {
00832         aux->parent->right = aux2;
00833     }
00834 
00835     if (aux != key)
00836     {
00837         key->key = aux->key; // Copia a chave do nó removido para o nó substituto
00838     }
00839 
00840     if (aux->color == NodeRB<Key, Value>::BLACK)
00841     {
00842         // Se o nó removido era preto, precisamos corrigir o balanceamento
00843         fixup_deletion(aux2);
00844     }
00845 
00846     delete aux; // Libera o nó removido
00847     size_m--;
00848 }
00849 
00850 template <typename Key, typename Value>
00851 NodeRB<Key, Value> *RedBlackTree<Key, Value>::minimun(NodePtr node)
00852 {
00853     NodePtr aux = node;
00854 
00855     while (aux->left != nil)
00856         aux = aux->left;
00857 
00858     if (aux == nil)
00859         throw std::out_of_range("No minimum node found in the subtree");
00860 
00861     return aux; // Retorna o nó com a menor chave na subárvore
00862 }
00863 
00864 template <typename Key, typename Value>
00865 Value &RedBlackTree<Key, Value>::at(NodePtr p, const Key &key)
00866 {
00867     if (p == nil)
00868         throw std::out_of_range("Key not found in the Red-Black Tree");
00869 
00870     comparisons++;
00871     if (key == p->key.first)
00872         return p->key.second;
00873 
00874     comparisons++;
00875     if (key < p->key.first)
00876     {
00877         return at(p->left, key);
00878     }
00879     else
00880         return at(p->right, key);
00881 }
00882 
00883 template <typename Key, typename Value>
00884 Value &RedBlackTree<Key, Value>::operator[](const Key &key)
00885 {
00886     NodePtr aux = root;
00887 
```

```
00888     while (aux != nil)
00889     {
00890         comparisons++;
00891         if (key == aux->key.first)
00892             return aux->key.second;
00893 
00894         comparisons++;
00895         if (key < aux->key.first)
00896             aux = aux->left;
00897         else
00898             aux = aux->right;
00899     }
00900     // Se a chave não for encontrada, insere um novo nó com valor padrão
00901     root = insert(root, new NodeRB<Key, Value>(std::pair<Key, Value>(key, Value()),
      NodeRB<Key, Value>::RED, nil, nil, nil));
00902 
00903     return at(root, key); // Retorna o valor associado à nova chave
00904 }
00905 
00906 template <typename Key, typename Value>
00907 void RedBlackTree<Key, Value>::update(NodePtr p, const std::pair<Key, Value> &key)
00908 {
00909     if (p == nil)
00910         throw std::out_of_range("Key not found in the Red-Black Tree for update");
00911 
00912     comparisons++;
00913     if (key.first == p->key.first)
00914     {
00915         p->key.second = key.second; // Atualiza o valor
00916     }
00917     else if (key.first < p->key.first)
00918     {
00919         comparisons++;
00920         update(p->left, key);
00921     }
00922     else
00923         update(p->right, key);
00924 }
00925 
00926 template <typename Key, typename Value>
00927 void RedBlackTree<Key, Value>::rightRotation(NodePtr p)
00928 {
00929     rotations++; // Incrementa o contador de rotações
00930 
00931     NodePtr aux = p->left;
00932     p->left = aux->right;
00933 
00934     if (aux->right != nil)
00935         aux->right->parent = p;
00936     aux->parent = p->parent;
00937 
00938     if (p->parent == nil)
00939     {
00940         root = aux; // Atualiza a raiz se necessário
00941     }
00942     else if (p == p->parent->right)
00943     {
00944         p->parent->right = aux;
00945     }
00946     else
00947     {
00948         p->parent->left = aux;
00949     }
00950     aux->right = p;
00951     p->parent = aux;
00952 }
00953 
00954 template <typename Key, typename Value>
00955 void RedBlackTree<Key, Value>::leftRotation(NodePtr p)
00956 {
00957     rotations++; // Incrementa o contador de rotações
00958 
00959     NodePtr aux = p->right;
00960     p->right = aux->left;
00961 
00962     if (aux->left != nil)
00963         aux->left->parent = p;
00964     aux->parent = p->parent;
00965 
00966     if (p->parent == nil)
00967     {
00968         root = aux; // Atualiza a raiz se necessário
00969     }
00970     else if (p == p->parent->left)
00971     {
00972         p->parent->left = aux;
00973     }
```

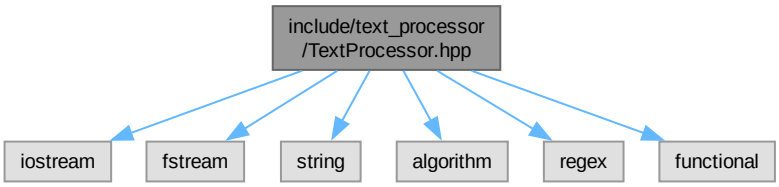
```
00974     else
00975     {
00976         p->parent->right = aux;
00977     }
00978     aux->left = p;
00979     p->parent = aux;
00980 }
00981
00982 template <typename Key, typename Value>
00983 NodeRB<Key, Value> *RedBlackTree<Key, Value>::clone_recursive(const NodePtr parent_new, const NodePtr
      node_other, const NodePtr nil_other) const
00984 {
00985     if (node_other == nil_other)
00986     {
00987         return nil; // Retorna o sentinela da NOVA árvore
00988     }
00989
00990     // Cria o novo nó com os mesmos dados e cor
00991     NodePtr new_node = new NodeRB<Key, Value>(node_other->key, node_other->color, parent_new, nil,
      nil);
00992
00993     // Define recursivamente os filhos esquerdo e direito
00994     new_node->left = clone_recursive(new_node, node_other->left, nil_other);
00995     new_node->right = clone_recursive(new_node, node_other->right, nil_other);
00996
00997     return new_node;
00998 }
00999
01000 template <typename Key, typename Value>
01001 bool RedBlackTree<Key, Value>::contains(NodePtr node, const Key &key)
01002 {
01003     if (node == nil)
01004         return false;
01005
01006     comparisons++;
01007     if (key == node->key.first)
01008         return true;
01009
01010     comparisons++;
01011     if (key < node->key.first)
01012     {
01013         return contains(node->left, key);
01014     }
01015     else
01016         return contains(node->right, key);
01017 }
01018
01019 template <typename Key, typename Value>
01020 bool RedBlackTree<Key, Value>::contains(const Key &key)
01021 {
01022     return contains(root, key);
01023 }
01024
01025 template <typename Key, typename Value>
01026 void RedBlackTree<Key, Value>::print() const
01027 {
01028     printInOrder(root);
01029 }
01030
01031 template <typename Key, typename Value>
01032 void RedBlackTree<Key, Value>::printInOrder(NodePtr node) const
01033 {
01034     if (node == nil)
01035         return;
01036
01037     else
01038     {
01039         printInOrder(node->left);
01040         std::cout « "[" « node->key.first « ", " « node->key.second « "]" « std::endl;
01041         printInOrder(node->right);
01042     }
01043 }
01044
01045 template <typename Key, typename Value>
01046 void RedBlackTree<Key, Value>::forEach(const std::function<void(const std::pair<Key, Value> &)> &func)
      const
01047 {
01048     for (const auto &pair : *this)
01049         func(pair);
01050 }
01051
01052 template <typename Key, typename Value>
01053 void RedBlackTree<Key, Value>::bshow()
01054 {
01055     bshow(root, "");
01056 }
01057
```

```
01058 template <typename Key, typename Value>
01059 void RedBlackTree<Key, Value>::bshow(NodePtr node, std::string heranca)
01060 {
01061     if (node != nil and (node->left != nil or node->right != nil))
01062         bshow(node->right, heranca + "r");
01063
01064     for (int i = 0; i < (int)heranca.size() - 1; i++)
01065         std::cout « (heranca[i] != heranca[i + 1] ? "    " : "    ");
01066
01067     if (heranca != "")
01068         std::cout « (heranca.back() == 'r' ? "" : "");
01069
01070     if (node == nil)
01071     {
01072         std::cout « "#" « std::endl;
01073         return;
01074     }
01075
01076     std::cout « (node->color == NodeRB<Key, Value>::RED ? "\x1b[31m" : "\x1b[30m") « "[" «
      node->key.first « ", " « node->key.second « "]"
01077               « "\x1b[0m" « std::endl;
01078
01079     if (node != nil and (node->left != nil or node->right != nil))
01080         bshow(node->left, heranca + "l");
01081 }
```

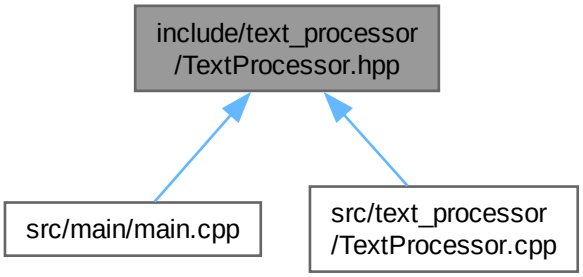
6.31 Referência ao ficheiro include/text_processor/TextProcessor.hpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <algorithm>
#include <regex>
#include <functional>
```

Diagrama de dependências de inclusão para TextProcessor.hpp:



Este grafo mostra quais são os ficheiros que incluem directamente ou indirectamente este ficheiro:



Componentes

- class TextProcessor

  Responsável por processar ficheiros de texto, extraindo e normalizando palavras.

6.32 TextProcessor.hpp

Ir para a documentação deste ficheiro.

```
00001 #pragma once
00002
00003 #include <iostream>
00004 #include <fstream>
00005 #include <string>
00006 #include <algorithm>
00007 #include <regex>
00008 #include <functional>
00009
00026 class TextProcessor
00027 {
00028 private:
00032     std::ifstream file_stream;
00033
00044     std::string normalize(const std::string &word) const;
00045
00046 public:
00062     explicit TextProcessor(const std::string &input_file);
00063
00067     ~TextProcessor() = default;
00068
00077     static void toLowerCase(std::string &text);
00078
00090     void processFile(const std::function<void(const std::string &)> &wordHandler);
00091 };
```

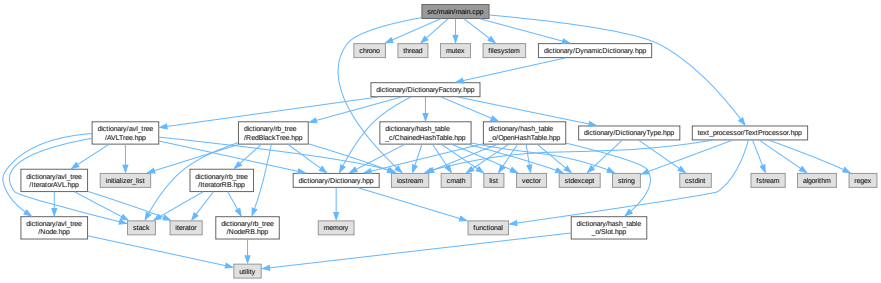
6.33 Referência ao ficheiro readme.md

6.34 Referência ao ficheiro src/main/main.cpp

```
#include <iostream>
#include <chrono>
#include <thread>
#include <mutex>
#include <filesystem>
#include "dictionary/DynamicDictionary.hpp"
```

```
#include "text_processor/TextProcessor.hpp"
```

Diagrama de dependências de inclusão para main.cpp:



Funções

- void create_directory (const string &DIR)

  Cria um diretório se ele não existir.
- void logException (const std::exception &e)

  Registra uma exceção em um arquivo de log.
- void print_usage ()

  Imprime a mensagem de uso do programa.
- string metrics (const Dictionary< std::string, unsigned int > &word_count)

  Imprime as métricas de desempenho de uma estrutura de dicionário.
- void write_output (const std::string &filename, const Dictionary< std::string, unsigned int > &word_count, const std::chrono::duration< double, std::milli > &buildTime, const string &structure_type)

  Escreve a contagem de palavras e métricas em um arquivo de saída.
- void counter_words (const std::string &filename, Dictionary< std::string, unsigned int > &word_count, const string &structure_type)

  Conta as palavras em um arquivo de texto e armazena a contagem em um dicionário.
- int main (int argc, char *argv[])

  Função principal do programa.

Variáveis

- const string INPUT_DIR = "files/"
- const string OUTPUT_DIR = "out/"
- const string LOG_DIR = "log/"
- mutex mtx

6.34.1 Documentação das funções

6.34.1.1 counter_words()

```
void counter_words (
            const std::string & filename,
            Dictionary< std::string, unsigned int > & word_count,
            const string & structure_type)
```

Conta as palavras em um arquivo de texto e armazena a contagem em um dicionário.
Esta função processa um arquivo de texto, conta a ocorrência de cada palavra e armazena a contagem em um dicionário. A função é executada em uma thread separada para permitir a contagem concorrente de palavras em diferentes estruturas de dados.
A função utiliza a classe `TextProcessor` para ler o arquivo e processar cada palavra. Cada palavra é normalizada e passada para um lambda que incrementa a contagem no dicionário. A contagem de palavras é feita de forma concorrente, utilizando um mutex para proteger o acesso ao dicionário.

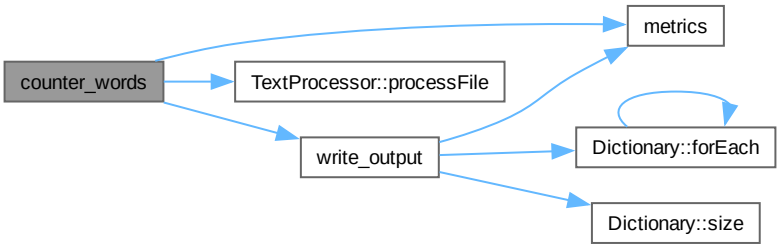
Parâmetros

filename	O nome do arquivo de texto a ser processado.
word_count	O dicionário onde as palavras e suas contagens serão armazenadas.
structure_type	O tipo da estrutura de dados utilizada para armazenar as contagens de palavras.

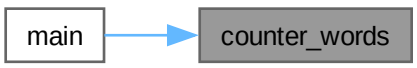
Definido na linha 200 do ficheiro main.cpp.

```
00201 {
00202     TextProcessor processor(INPUT_DIR + filename);
00203
00204     auto startTime = chrono::high_resolution_clock::now();
00205
00206     processor.processFile([&word_count](const string &word)
00207                           { word_count[word]++; });
00208
00209     auto endTime = chrono::high_resolution_clock::now();
00210     chrono::duration<double, milli> buildTime = endTime - startTime;
00211
00212     {
00213         lock_guard<mutex> lock(mtx); // Protege o acesso ao dicionário
00214
00215         cout « "================================================================================" « endl;
00216         cout « "structure: " « structure_type « endl;
00217         cout « "build time: " « buildTime.count() « " ms" « endl;
00218         cout « metrics(word_count);
00219         cout « "================================================================================" « endl;
00220         cout « endl;
00221     }
00222     {
00223         lock_guard<mutex> lock(mtx); // Protege o acesso ao arquivo de saída
00224
00225         write_output(filename, word_count, buildTime, structure_type);
00226     }
00227 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



6.34.1.2 create_directory()

```
void create_directory (
```

```
            const string & DIR)
```

Cria um diretório se ele não existir.
Esta função verifica se o diretório especificado existe e, se não existir, tenta criá-lo.

Parâmetros

DIR	O caminho do diretório a ser criado.

Excepções

std::runtime_error	Se ocorrer um erro ao criar o diretório.

Definido na linha 28 do ficheiro main.cpp.

```
00029 {
00030     try
00031     {
00032         if (!filesystem::exists(DIR))
00033             filesystem::create_directory(DIR); // Cria o diretório se não existir
00034     }
00035     catch (const std::exception &e)
00036     {
00037         cerr « "Error creating directory '" « DIR « "': " « e.what() « endl;
00038         throw std::runtime_error("Failed to create directory: " + DIR);
00039     }
00040 }
```

Este é o diagrama das funções que utilizam esta função:

6.34.1.3 logException()

```
void logException (
            const std::exception & e)
```

Registra uma exceção em um arquivo de log.
Esta função captura uma exceção e registra sua mensagem em um arquivo de log, juntamente com a data e hora atuais. O log é armazenado no diretório LOG_DIR e o arquivo é nomeado "log.txt".
A função utiliza a biblioteca `<chrono>` para obter o tempo atual e a biblioteca `<iomanip>` para formatar a data e hora. A mensagem da exceção é obtida através do método `what()` da classe `std::exception`. O log é escrito em um arquivo no formato "YYYY-MM-DD HH:MM:SS - mensagem_da_exceção".

Parâmetros

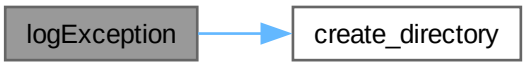
e	A exceção a ser registrada.

Definido na linha 57 do ficheiro main.cpp.

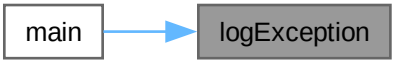
```
00058 {
00059     create_directory(LOG_DIR); // Cria o diretório de log se não existir
00060
00061     time_t logTime = chrono::system_clock::to_time_t(chrono::system_clock::now());
00062     ofstream logFile(LOG_DIR + "log.txt", ios::app);
00063     logFile « put_time(localtime(&logTime), "%Y-%m-%d %H:%M:%S") « " - " « e.what() « '\n';
```

```
00064     logFile.close();
00065 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



6.34.1.4 main()

```
int main (
              int argc,
              char * argv[])
```

Função principal do programa.
Esta função serve como ponto de entrada para a aplicação de contagem de palavras. Ela processa os argumentos da linha de comando para determinar qual(is) estrutura(s) de dados utilizar e qual arquivo de texto processar.
A função pode operar em dois modos:

1. **Modo específico**: Se um tipo de estrutura de dados (ex: "avl", "rbt") é fornecido, a contagem de palavras é realizada usando apenas essa estrutura.
2. **Modo "all"**: Se o argumento for "all", a função cria quatro threads, cada uma executando a contagem de palavras em paralelo com uma estrutura de dados diferente (AVL, Red-Black Tree, Chaining Hash, Open Addressing Hash) para fins de comparação de desempenho.

A função também lida com a validação do número de argumentos e captura exceções que possam ocorrer durante a execução, registrando-as em um log.

Parâmetros

argc	O número de argumentos fornecidos na linha de comando. O programa espera pelo menos 3 (nome do programa, tipo da estrutura, arquivo de entrada).
argv	Um vetor de strings contendo os argumentos da linha de comando. - `argv[0]`: O nome do programa (geralmente "Dictionary"). - `argv[1]`: O tipo da estrutura de dados a ser usada ("avl", "rbt", "chash", "ohash") ou "all" - `argv[2]`: O caminho para o arquivo de texto de entrada.

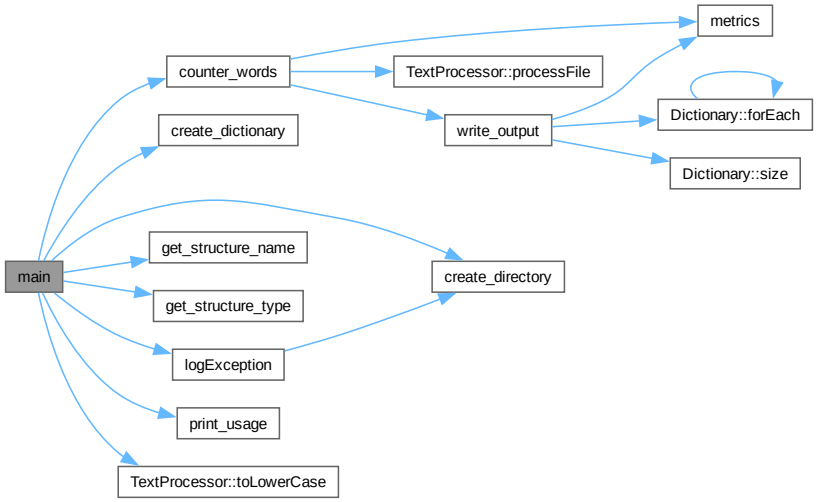
Retorna

int Retorna 0 em caso de execução bem-sucedida. Retorna um código de erro implícito em caso de falha (geralmente gerenciado pelo sistema operacional).

Definido na linha 259 do ficheiro main.cpp.

```
00260 {
00261     if (argc == 2 and string(argv[1]) == "help")
00262     {
00263         print_usage();
00264         return 0; // Se o usuário solicitar ajuda, exibe a mensagem de uso e encerra o programa
00265     }
00266
00267     if (argc < 3)
00268     {
00269         print_usage();
00270         logException(std::invalid_argument("Invalid number of arguments"));
00271         throw std::invalid_argument("Invalid number of arguments. Expected at least 3 arguments.");
00272     }
00273
00274     // Cria os diretórios de entrada e de saída se não existirem
00275     try
00276     {
00277         if (!filesystem::exists(INPUT_DIR))
00278         {
00279             create_directory(INPUT_DIR);
00280             cout « "Input directory created: " « INPUT_DIR « endl;
00281             cout « "Please place your input files in this directory." « endl;
00282             return 0; // Se o diretório de entrada não existir, cria e encerra o programa
00283         }
00284         create_directory(OUTPUT_DIR);
00285     }
00286     catch (const std::exception &e)
00287     {
00288         logException(e);
00289         cerr « "Failed to create necessary directories: " « e.what() « endl;
00290         exit(EXIT_FAILURE); // Retorna um código de erro se não for possível criar os diretórios
00291     }
00292
00293     setlocale(LC_ALL, "Pt_BR.UTF-8");
00294
00295     string structure_type = argv[1];
00296     string input_file = argv[2];
00297
00298     TextProcessor::toLowerCase(structure_type);
00299
00300     try
00301     {
00302         if (structure_type == "all")
00303         {
00304             thread threads[4];
00305             unique_ptr<Dictionary<string, unsigned int» counters[4];
00306
00307             for (size_t i = 0; i < 4; i++)
00308             {
00309                 counters[i] = create_dictionary<string, unsigned int>(DictionaryType(i));
00310                 threads[i] = thread(counter_words, input_file, ref(*counters[i]),
    get_structure_name(DictionaryType(i)));
00311             }
00312
00313             for (size_t i = 0; i < 4; ++i)
00314                 threads[i].join();
00315         }
00316         else
00317         {
00318             DictionaryType type = get_structure_type(structure_type);
00319             unique_ptr<Dictionary<string, unsigned int»
    counter(create_dictionary<string, unsigned int>(type));
00320             counter_words(input_file, *counter, get_structure_name(type));
00321         }
00322     }
00323     catch (const std::exception &e)
00324     {
00325         cerr « "An error occurred: " « e.what() « endl;
00326         logException(e);
00327     }
00328
00329     cout « "Processing completed." « endl;
00330     cout « "Results are saved in the '" « OUTPUT_DIR « "' directory." « endl;
00331
00332     return 0;
00333 }
```

Grafo de chamadas desta função:



6.34.1.5 metrics()

```
string metrics (
            const Dictionary< std::string, unsigned int > & word_count)
```

Imprime as métricas de desempenho de uma estrutura de dicionário.
Esta função recebe um dicionário contendo a contagem de palavras e imprime as métricas de desempenho, como colisões, comparações, rotações, etc., dependendo do tipo da estrutura de dados utilizada.

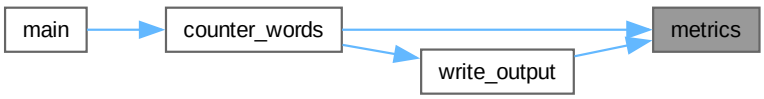
Parâmetros

word_count	O dicionário contendo a contagem de palavras.
output_file	O arquivo onde as métricas serão escritas.

Definido na linha 95 do ficheiro main.cpp.

```
00096 {
00097     stringstream ss;
00098     if (auto *chained_hash = dynamic_cast<const ChainedHashTable<std::string, unsigned int>
      *>(&word_count))
00099     {
00100         ss « "Collisions: " « chained_hash->getCollisions() « endl;
00101         ss « "Comparisons: " « chained_hash->getComparisons() « endl;
00102     }
00103     else if (auto *open_hash = dynamic_cast<const OpenHashTable<std::string, unsigned int>
      *>(&word_count))
00104     {
00105         ss « "Collisions: " « open_hash->getCollisions() « endl;
00106         ss « "Comparisons: " « open_hash->getComparisons() « endl;
00107     }
00108     else if (auto *avl_tree = dynamic_cast<const AVLTree<std::string, unsigned int> *>(&word_count))
00109     {
00110         ss « "Rotations: " « avl_tree->getRotations() « endl;
00111         ss « "Comparisons: " « avl_tree->getComparisons() « endl;
00112     }
00113     else if (auto *rbtree = dynamic_cast<const RedBlackTree<std::string, unsigned int>
      *>(&word_count))
00114     {
00115         ss « "Rotations: " « rbtree->getRotations() « endl;
00116         ss « "Comparisons: " « rbtree->getComparisons() « endl;
00117     }
00118
00119     return ss.str();
00120 }
```

Este é o diagrama das funções que utilizam esta função:



6.34.1.6 print_usage()

```
void print_usage ()
```

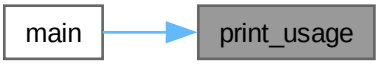
Imprime a mensagem de uso do programa.

Esta função exibe uma mensagem de uso do programa, informando ao usuário como executar o programa corretamente, quais estruturas de dados estão disponíveis e como especificar o arquivo de entrada. É útil para usuários que não estão familiarizados com a linha de comando ou que precisam de ajuda para entender como usar o programa.

Definido na linha 75 do ficheiro main.cpp.

```
00076 {
00077     cout « "Helper: Dictionary Word Counter" « endl;
00078     cout « "Use: ./Dictionary <structure> <input_file>" « endl;
00079     cout « "Available structures: avl, rbt, chash, ohash, all" « endl;
00080     cout « "Example: ./Dictionary avl input.txt" « endl;
00081     cout « "Note: The input file should be placed in the 'files/' directory." « endl;
00082     cout « "If you want see again this message, run: ./Dictionary help" « endl;
00083 }
```

Este é o diagrama das funções que utilizam esta função:



6.34.1.7 write_output()

```
void write_output (
            const std::string & filename,
            const Dictionary< std::string, unsigned int > & word_count,
            const std::chrono::duration< double, std::milli > & buildTime,
            const string & structure_type)
```

Escreve a contagem de palavras e métricas em um arquivo de saída.

Esta função recebe um dicionário contendo a contagem de palavras, o tempo de construção e o tipo de estrutura utilizada, e escreve essas informações em um arquivo especificado.

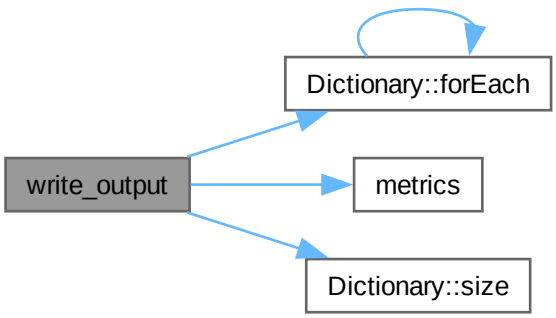
Parâmetros

filename	O nome do arquivo onde as informações serão escritas.
word_count	O dicionário contendo as palavras e suas respectivas contagens.
buildTime	O tempo gasto para construir a estrutura de dados.
structure_type	O tipo da estrutura de dados utilizada (ex: "AVL", "RBTREE", etc.).

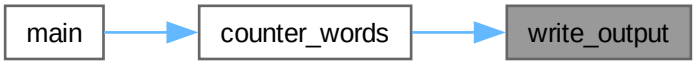
Definido na linha 133 do ficheiro main.cpp.

```
00134 {
00135     ofstream output_file(OUTPUT_DIR + filename, ios::app);
00136 
00137     if (!output_file or !output_file.is_open())
00138         throw std::runtime_error("Failed to open output file " + filename);
00139 
00140     if (output_file.tellp() == 0) // Verifica se o arquivo está vazio
00141     {
00142         output_file « "================================================================================" «
      endl;
00143         output_file « "Word Count for file: " « filename « endl;
00144         output_file « "================================================================================" «
      endl;
00145         output_file « endl;
00146 
00147         if (structure_type == "CHAINING_HASH" or structure_type == "OPEN_ADDRESSING_HASH")
00148         {
00149             vector<pair<string, unsigned int» word_vector;
00150 
00151             word_count.forEach([&word_vector](const std::pair<std::string, unsigned int> &pair)
00152                                { word_vector.emplace_back(pair); });
00153 
00154             sort(word_vector.begin(), word_vector.end(), [](const auto &a, const auto &b)
00155                  { return a.first < b.first; });
00156 
00157             for (const auto &[word, count] : word_vector)
00158                 output_file « "[" « word « ", " « count « "]" « endl;
00159         }
00160         else
00161         {
00162             word_count.forEach([&output_file](const std::pair<std::string, unsigned int> &pair)
00163                                { output_file « "[" « pair.first « ", " « pair.second « "]" « endl; });
00164         }
00165     }
00166 
00167     output_file « endl;
00168     output_file « "================================================================================" « endl;
00169 
00170     output_file « "Metrics:" « endl;
00171     output_file « "Structure: " « structure_type « endl;
00172     output_file « "Build time: " « buildTime.count() « " ms" « endl;
00173     output_file « "Size: " « word_count.size() « endl;
00174 
00175     output_file « metrics(word_count);
00176 
00177     output_file « "================================================================================" « endl
00178                 « endl;
00179 
00180     output_file.close();
00181 }
```

Grafo de chamadas desta função:



Este é o diagrama das funções que utilizam esta função:



6.34.2 Documentação das variáveis

6.34.2.1 INPUT_DIR

```
const string INPUT_DIR = "files/"
```

Definido na linha 12 do ficheiro main.cpp.

6.34.2.2 LOG_DIR

```
const string LOG_DIR = "log/"
```

Definido na linha 14 do ficheiro main.cpp.

6.34.2.3 mtx

```
mutex mtx
```

Definido na linha 16 do ficheiro main.cpp.

6.34.2.4 OUTPUT_DIR

```
const string OUTPUT_DIR = "out/"
```

Definido na linha 13 do ficheiro main.cpp.

6.35 main.cpp

Ir para a documentação deste ficheiro.

```
00001 #include <iostream>
00002 #include <chrono>
00003 #include <thread>
00004 #include <mutex>
00005 #include <filesystem>
00006
00007 #include "dictionary/DynamicDictionary.hpp"
00008 #include "text_processor/TextProcessor.hpp"
00009
00010 using namespace std;
00011
00012 const string INPUT_DIR = "files/"; // Diretório de entrada para os arquivos de texto
00013 const string OUTPUT_DIR = "out/";  // Diretório de saída para os arquivos de contagem
00014 const string LOG_DIR = "log/";     // Diretório de log para erros e exceções
00015
00016 mutex mtx; // Mutex para proteger o acesso ao dicionário
00017
00028 void create_directory(const string &DIR)
00029 {
00030     try
00031     {
00032         if (!filesystem::exists(DIR))
00033             filesystem::create_directory(DIR); // Cria o diretório se não existir
00034     }
00035     catch (const std::exception &e)
00036     {
00037         cerr « "Error creating directory '" « DIR « "': " « e.what() « endl;
00038         throw std::runtime_error("Failed to create directory: " + DIR);
00039     }
00040 }
00041
```

```
00057 void logException(const std::exception &e)
00058 {
00059     create_directory(LOG_DIR); // Cria o diretório de log se não existir
00060
00061     time_t logTime = chrono::system_clock::to_time_t(chrono::system_clock::now());
00062     ofstream logFile(LOG_DIR + "log.txt", ios::app);
00063     logFile « put_time(localtime(&logTime), "%Y-%m-%d %H:%M:%S") « " - " « e.what() « '\n';
00064     logFile.close();
00065 }
00066
00075 void print_usage()
00076 {
00077     cout « "Helper: Dictionary Word Counter" « endl;
00078     cout « "Use: ./Dictionary <structure> <input_file>" « endl;
00079     cout « "Available structures: avl, rbt, chash, ohash, all" « endl;
00080     cout « "Example: ./Dictionary avl input.txt" « endl;
00081     cout « "Note: The input file should be placed in the 'files/' directory." « endl;
00082     cout « "If you want see again this message, run: ./Dictionary help" « endl;
00083 }
00084
00095 string metrics(const Dictionary<std::string, unsigned int> &word_count)
00096 {
00097     stringstream ss;
00098     if (auto *chained_hash = dynamic_cast<const ChainedHashTable<std::string, unsigned int>
      *>(&word_count))
00099     {
00100         ss « "Collisions: " « chained_hash->getCollisions() « endl;
00101         ss « "Comparisons: " « chained_hash->getComparisons() « endl;
00102     }
00103     else if (auto *open_hash = dynamic_cast<const OpenHashTable<std::string, unsigned int>
      *>(&word_count))
00104     {
00105         ss « "Collisions: " « open_hash->getCollisions() « endl;
00106         ss « "Comparisons: " « open_hash->getComparisons() « endl;
00107     }
00108     else if (auto *avl_tree = dynamic_cast<const AVLTree<std::string, unsigned int> *>(&word_count))
00109     {
00110         ss « "Rotations: " « avl_tree->getRotations() « endl;
00111         ss « "Comparisons: " « avl_tree->getComparisons() « endl;
00112     }
00113     else if (auto *rbtree = dynamic_cast<const RedBlackTree<std::string, unsigned int>
      *>(&word_count))
00114     {
00115         ss « "Rotations: " « rbtree->getRotations() « endl;
00116         ss « "Comparisons: " « rbtree->getComparisons() « endl;
00117     }
00118
00119     return ss.str();
00120 }
00121
00133 void write_output(const std::string &filename, const Dictionary<std::string, unsigned int>
      &word_count, const std::chrono::duration<double, std::milli> &buildTime, const string &structure_type)
00134 {
00135     ofstream output_file(OUTPUT_DIR + filename, ios::app);
00136
00137     if (!output_file or !output_file.is_open())
00138         throw std::runtime_error("Failed to open output file " + filename);
00139
00140     if (output_file.tellp() == 0) // Verifica se o arquivo está vazio
00141     {
00142         output_file « "================================================================================" «
      endl;
00143         output_file « "Word Count for file: " « filename « endl;
00144         output_file « "================================================================================" «
      endl;
00145         output_file « endl;
00146
00147         if (structure_type == "CHAINING_HASH" or structure_type == "OPEN_ADDRESSING_HASH")
00148         {
00149             vector<pair<string, unsigned int» word_vector;
00150
00151             word_count.forEach([&word_vector](const std::pair<std::string, unsigned int> &pair)
00152                                { word_vector.emplace_back(pair); });
00153
00154             sort(word_vector.begin(), word_vector.end(), [](const auto &a, const auto &b)
00155                  { return a.first < b.first; });
00156
00157             for (const auto &[word, count] : word_vector)
00158                 output_file « "[" « word « ", " « count « "]" « endl;
00159         }
00160         else
00161         {
00162             word_count.forEach([&output_file](const std::pair<std::string, unsigned int> &pair)
00163                                { output_file « "[" « pair.first « ", " « pair.second « "]" « endl; });
00164         }
00165     }
00166
```

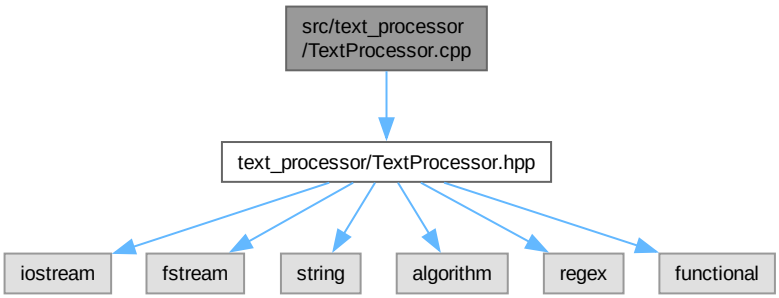
```
00167     output_file « endl;
00168     output_file « "================================================================================" « endl;
00169
00170     output_file « "Metrics:" « endl;
00171     output_file « "Structure: " « structure_type « endl;
00172     output_file « "Build time: " « buildTime.count() « " ms" « endl;
00173     output_file « "Size: " « word_count.size() « endl;
00174
00175     output_file « metrics(word_count);
00176
00177     output_file « "================================================================================" « endl
00178                 « endl;
00179
00180     output_file.close();
00181 }
00182
00200 void counter_words(const std::string &filename, Dictionary<std::string, unsigned int> &word_count,
      const string &structure_type)
00201 {
00202     TextProcessor processor(INPUT_DIR + filename);
00203
00204     auto startTime = chrono::high_resolution_clock::now();
00205
00206     processor.processFile([&word_count](const string &word)
00207                           { word_count[word]++; });
00208
00209     auto endTime = chrono::high_resolution_clock::now();
00210     chrono::duration<double, milli> buildTime = endTime - startTime;
00211
00212     {
00213         lock_guard<mutex> lock(mtx); // Protege o acesso ao dicionário
00214
00215         cout « "================================================================================" « endl;
00216         cout « "structure: " « structure_type « endl;
00217         cout « "build time: " « buildTime.count() « " ms" « endl;
00218         cout « metrics(word_count);
00219         cout « "================================================================================" « endl;
00220         cout « endl;
00221     }
00222     {
00223         lock_guard<mutex> lock(mtx); // Protege o acesso ao arquivo de saída
00224
00225         write_output(filename, word_count, buildTime, structure_type);
00226     }
00227 }
00228
00259 int main(int argc, char *argv[])
00260 {
00261     if (argc == 2 and string(argv[1]) == "help")
00262     {
00263         print_usage();
00264         return 0; // Se o usuário solicitar ajuda, exibe a mensagem de uso e encerra o programa
00265     }
00266
00267     if (argc < 3)
00268     {
00269         print_usage();
00270         logException(std::invalid_argument("Invalid number of arguments"));
00271         throw std::invalid_argument("Invalid number of arguments. Expected at least 3 arguments.");
00272     }
00273
00274     // Cria os diretórios de entrada e de saída se não existirem
00275     try
00276     {
00277         if (!filesystem::exists(INPUT_DIR))
00278         {
00279             create_directory(INPUT_DIR);
00280             cout « "Input directory created: " « INPUT_DIR « endl;
00281             cout « "Please place your input files in this directory." « endl;
00282             return 0; // Se o diretório de entrada não existir, cria e encerra o programa
00283         }
00284         create_directory(OUTPUT_DIR);
00285     }
00286     catch (const std::exception &e)
00287     {
00288         logException(e);
00289         cerr « "Failed to create necessary directories: " « e.what() « endl;
00290         exit(EXIT_FAILURE); // Retorna um código de erro se não for possível criar os diretórios
00291     }
00292
00293     setlocale(LC_ALL, "Pt_BR.UTF-8");
00294
00295     string structure_type = argv[1];
00296     string input_file = argv[2];
00297
00298     TextProcessor::toLowerCase(structure_type);
00299
```

```
00300     try
00301     {
00302         if (structure_type == "all")
00303         {
00304             thread threads[4];
00305             unique_ptr<Dictionary<string, unsigned int» counters[4];
00306 
00307             for (size_t i = 0; i < 4; i++)
00308             {
00309                 counters[i] = create_dictionary<string, unsigned int>(DictionaryType(i));
00310                 threads[i] = thread(counter_words, input_file, ref(*counters[i]),
      get_structure_name(DictionaryType(i)));
00311             }
00312 
00313             for (size_t i = 0; i < 4; ++i)
00314                 threads[i].join();
00315         }
00316         else
00317         {
00318             DictionaryType type = get_structure_type(structure_type);
00319             unique_ptr<Dictionary<string, unsigned int»
      counter(create_dictionary<string, unsigned int>(type));
00320             counter_words(input_file, *counter, get_structure_name(type));
00321         }
00322     }
00323     catch (const std::exception &e)
00324     {
00325         cerr « "An error occurred: " « e.what() « endl;
00326         logException(e);
00327     }
00328 
00329     cout « "Processing completed." « endl;
00330     cout « "Results are saved in the '" « OUTPUT_DIR « "' directory." « endl;
00331 
00332     return 0;
00333 }
```

6.36 Referência ao ficheiro src/text_processor/TextProcessor.cpp

`#include "text_processor/TextProcessor.hpp"`

Diagrama de dependências de inclusão para TextProcessor.cpp:



6.37 TextProcessor.cpp

Ir para a documentação deste ficheiro.

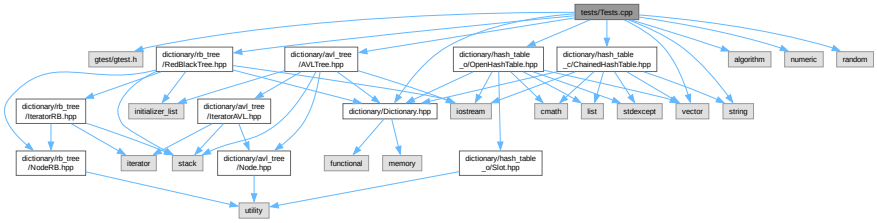
```
00001 #include "text_processor/TextProcessor.hpp"
00002 
00003 TextProcessor::TextProcessor(const std::string &input_file) : file_stream(input_file)
00004 {
00005     if (!file_stream or !file_stream.is_open())
00006         throw std::runtime_error("CRITICAL ERROR: Could not open file: " + input_file + ".");
00007 }
00008 
00009 void TextProcessor::toLowerCase(std::string &text)
00010 {
00011     std::transform(text.begin(), text.end(), text.begin(), [](unsigned char c)
```

```
00012                   { return std::tolower(c); });
00013 }
00014
00015 void TextProcessor::processFile(const std::function<void(const std::string &)> &wordHandler)
00016 {
00017     std::string word{};
00018
00019     while (file_stream » word)
00020     {
00021         std::string cleanedWord{normalize(word)};
00022
00023         if (!cleanedWord.empty())
00024             wordHandler(cleanedWord);
00025     }
00026 }
00027
00028 std::string TextProcessor::normalize(const std::string &word) const
00029 {
00030     std::string lowerWord{word};
00031
00032     toLowerCase(lowerWord);
00033
00034     static const std::regex word_regex("[a-zà-ÿ]+(?:[-’][a-zà-ÿ]+)*"); // Regex para palavras válidas,
    incluindo acentos e hífens.
00035
00036     std::smatch match;
00037
00038     if (std::regex_search(lowerWord, match, word_regex))
00039         return match.str();
00040
00041     return "";
00042 }
```

6.38 Referência ao ficheiro tests/Tests.cpp

```
#include "gtest/gtest.h"
#include "dictionary/Dictionary.hpp"
#include "dictionary/avl_tree/AVLTree.hpp"
#include "dictionary/rb_tree/RedBlackTree.hpp"
#include "dictionary/hash_table_c/ChainedHashTable.hpp"
#include "dictionary/hash_table_o/OpenHashTable.hpp"
#include <string>
#include <vector>
#include <algorithm>
#include <numeric>
#include <random>
```

Diagrama de dependências de inclusão para Tests.cpp:



Componentes

- class DictionaryTest< T >

  Fixture genérico para a interface Dictionary.
- class AVLTreeSpecificTest
- class HashTableStressTest< T >

  Usamos novamente um teste tipado para aplicar os mesmos testes de estresse para ChainedHashTable e OpenHashTable.
- class GeneralStressTest< T >

Definições de tipos

- using HashTableImplementations
- using Implementations

Funções

- TYPED_TEST_SUITE_P (DictionaryTest)

  Declaração da suíte de testes tipados.
- TYPED_TEST_P (DictionaryTest, DefaultConstructor)

  Testa o estado inicial de um dicionário recém-criado.
- TYPED_TEST_P (DictionaryTest, InsertAndSize)

  Testa a inserção de um único elemento e o tamanho.
- TYPED_TEST_P (DictionaryTest, InsertDuplicates)

  Testa se a inserção de chaves duplicadas é ignorada.
- TYPED_TEST_P (DictionaryTest, Contains)

  Testa a funcionalidade de 'contains'.
- TYPED_TEST_P (DictionaryTest, Remove)

  Testa a remoção de elementos.
- TYPED_TEST_P (DictionaryTest, RemoveNodeWithTwoChildren)

  Testa a remoção de um nó com dois filhos (importante para árvores)
- TYPED_TEST_P (DictionaryTest, At)

  Testa a função 'at' para acesso e exceções.
- TYPED_TEST_P (DictionaryTest, Update)

  Testa a função 'update'.
- TYPED_TEST_P (DictionaryTest, BracketOperator)

  Testa o operador de colchetes [].
- TYPED_TEST_P (DictionaryTest, Clear)

  Testa a limpeza do dicionário.
- TYPED_TEST_P (DictionaryTest, Clone)

  Testa a clonagem do dicionário.
- TYPED_TEST_P (DictionaryTest, ForEach)

  Testa a iteração com forEach.
- REGISTER_TYPED_TEST_SUITE_P (DictionaryTest, DefaultConstructor, InsertAndSize, InsertDuplicates, Contains, Remove, RemoveNodeWithTwoChildren, At, Update, BracketOperator, Clear, Clone, ForEach)

  Registra todos os testes definidos acima para a suíte.
- TEST_F (AVLTreeSpecificTest, InsertTriggersSingleRightRotation)

  Força uma rotação simples à direita (Left-Left case)
- TEST_F (AVLTreeSpecificTest, InsertTriggersSingleLeftRotation)

  Força uma rotação simples à esquerda (Right-Right case)
- TEST_F (AVLTreeSpecificTest, InsertTriggersRightLeftRotation)

  Força uma rotação dupla direita-esquerda (Right-Left case)
- TEST_F (AVLTreeSpecificTest, InsertTriggersLeftRightRotation)

  Força uma rotação dupla esquerda-direita (Left-Right case)
- TEST_F (AVLTreeSpecificTest, RemovalTriggersRebalancing)

  Testa a remoção que deve acionar rotações para rebalancear.
- TYPED_TEST_SUITE_P (HashTableStressTest)
- TYPED_TEST_P (HashTableStressTest, HighCollisionRate)

  Testa o comportamento sob alta taxa de colisão Para isso, precisamos de uma função de hash que possamos controlar. Como não podemos, vamos simular inserindo múltiplos de um valor inicial que, dependendo do tamanho da tabela, provavelmente colidirão.
- TYPED_TEST_P (HashTableStressTest, RehashingOnHighLoadFactor)

Testa a funcionalidade de redimensionamento (rehashing)

- REGISTER_TYPED_TEST_SUITE_P (HashTableStressTest, HighCollisionRate, RehashingOnHighLoad↩ Factor)
- INSTANTIATE_TYPED_TEST_SUITE_P (MyHashImplementations, HashTableStressTest, HashTableImplementations)
- TYPED_TEST_SUITE_P (GeneralStressTest)
- TYPED_TEST_P (GeneralStressTest, LargeRandomInsertAndRemove)
- REGISTER_TYPED_TEST_SUITE_P (GeneralStressTest, LargeRandomInsertAndRemove)
- INSTANTIATE_TYPED_TEST_SUITE_P (MyImplementations, DictionaryTest, Implementations)

  Instancia a suíte de testes para cada uma das implementações.
- INSTANTIATE_TYPED_TEST_SUITE_P (MyGeneralStress, GeneralStressTest, Implementations)

  Instancia a suíte de testes de estresse para as implementações de dicionário.

6.38.1 Documentação dos tipos

6.38.1.1 HashTableImplementations

```
using HashTableImplementations
```

Valor inicial:

```
::testing::Types<
    ChainedHashTable<int, std::string>,
    OpenHashTable<int, std::string>>
```

Definido na linha 413 do ficheiro Tests.cpp.

6.38.1.2 Implementations

```
using Implementations
```

Valor inicial:

```
::testing::Types<
    AVLTree<int, std::string>,
    RedBlackTree<int, std::string>,
    ChainedHashTable<int, std::string>,
    OpenHashTable<int, std::string>>
```

Definido na linha 495 do ficheiro Tests.cpp.

6.38.2 Documentação das funções

6.38.2.1 INSTANTIATE_TYPED_TEST_SUITE_P() [1/3]

```
INSTANTIATE_TYPED_TEST_SUITE_P (
            MyGeneralStress ,
            GeneralStressTest ,
            Implementations )
```

Instancia a suíte de testes de estresse para as implementações de dicionário.

6.38.2.2 INSTANTIATE_TYPED_TEST_SUITE_P() [2/3]

```
INSTANTIATE_TYPED_TEST_SUITE_P (
            MyHashImplementations ,
            HashTableStressTest ,
            HashTableImplementations )
```

6.38.2.3 INSTANTIATE_TYPED_TEST_SUITE_P() [3/3]

```
INSTANTIATE_TYPED_TEST_SUITE_P (
            MyImplementations ,
            DictionaryTest ,
            Implementations )
```

Instancia a suíte de testes para cada uma das implementações.

6.38.2.4 REGISTER_TYPED_TEST_SUITE_P() [1/3]

```
REGISTER_TYPED_TEST_SUITE_P (
            DictionaryTest ,
            DefaultConstructor ,
            InsertAndSize ,
            InsertDuplicates ,
            Contains ,
            Remove ,
            RemoveNodeWithTwoChildren ,
            At ,
            Update ,
            BracketOperator ,
            Clear ,
            Clone ,
            ForEach )
```

Registra todos os testes definidos acima para a suíte.

6.38.2.5 REGISTER_TYPED_TEST_SUITE_P() [2/3]

```
REGISTER_TYPED_TEST_SUITE_P (
            GeneralStressTest ,
            LargeRandomInsertAndRemove )
```

6.38.2.6 REGISTER_TYPED_TEST_SUITE_P() [3/3]

```
REGISTER_TYPED_TEST_SUITE_P (
            HashTableStressTest ,
            HighCollisionRate ,
            RehashingOnHighLoadFactor )
```

6.38.2.7 TEST_F() [1/5]

```
TEST_F (
            AVLTreeSpecificTest ,
            InsertTriggersLeftRightRotation )
```

Força uma rotação dupla esquerda-direita (Left-Right case)
Definido na linha 273 do ficheiro Tests.cpp.

```
00274 {
00275     avl.insert({30, "thirty"});
00276     avl.insert({10, "ten"});
00277     avl.insert({20, "twenty"}); // Causa desbalanceamento LR no nó 30
00278
00279     EXPECT_EQ(avl.size(), static_cast<size_t>(3));
00280     EXPECT_TRUE(avl.contains(10));
00281     EXPECT_TRUE(avl.contains(20));
00282     EXPECT_TRUE(avl.contains(30));
00283 }
```

6.38.2.8 TEST_F() [2/5]

```
TEST_F (
            AVLTreeSpecificTest ,
            InsertTriggersRightLeftRotation )
```

Força uma rotação dupla direita-esquerda (Right-Left case)
Definido na linha 260 do ficheiro Tests.cpp.

```
00261 {
00262     avl.insert({10, "ten"});
00263     avl.insert({30, "thirty"});
00264     avl.insert({20, "twenty"}); // Causa desbalanceamento RL no nó 10
00265
00266     EXPECT_EQ(avl.size(), static_cast<size_t>(3));
00267     EXPECT_TRUE(avl.contains(10));
00268     EXPECT_TRUE(avl.contains(20));
00269     EXPECT_TRUE(avl.contains(30));
00270 }
```

6.38.2.9 TEST_F() [3/5]

```
TEST_F (
            AVLTreeSpecificTest ,
            InsertTriggersSingleLeftRotation )
```

Força uma rotação simples à esquerda (Right-Right case)
Definido na linha 247 do ficheiro Tests.cpp.

```
00248 {
00249     avl.insert({10, "ten"});
00250     avl.insert({20, "twenty"});
00251     avl.insert({30, "thirty"}); // Causa desbalanceamento RR no nó 10
00252
00253     EXPECT_EQ(avl.size(), static_cast<size_t>(3));
00254     EXPECT_TRUE(avl.contains(10));
00255     EXPECT_TRUE(avl.contains(20));
00256     EXPECT_TRUE(avl.contains(30));
00257 }
```

6.38.2.10 TEST_F() [4/5]

```
TEST_F (
            AVLTreeSpecificTest ,
            InsertTriggersSingleRightRotation )
```

Força uma rotação simples à direita (Left-Left case)
Definido na linha 230 do ficheiro Tests.cpp.

```
00231 {
00232     avl.insert({30, "thirty"});
00233     avl.insert({20, "twenty"});
00234     avl.insert({10, "ten"}); // Causa desbalanceamento LL no nó 30
00235
00236     // Após a rotação, 20 deve ser a nova raiz.
00237     // A verificação é feita checando o tamanho e a presença das chaves,
00238     // mas o principal é garantir que a inserção não quebrou a estrutura.
00239     EXPECT_EQ(avl.size(), static_cast<size_t>(3));
00240     EXPECT_TRUE(avl.contains(10));
00241     EXPECT_TRUE(avl.contains(20));
00242     EXPECT_TRUE(avl.contains(30));
00243     // Um teste mais profundo (caixa-branca) poderia verificar quem é a raiz.
00244 }
```

6.38.2.11 TEST_F() [5/5]

```
TEST_F (
            AVLTreeSpecificTest ,
            RemovalTriggersRebalancing )
```

Testa a remoção que deve acionar rotações para rebalancear.
Definido na linha 286 do ficheiro Tests.cpp.

```
00287 {
00288     // Cria uma árvore maior e mais complexa
00289     int keys[] = {40, 20, 60, 10, 30, 50, 70, 5, 15, 25, 35};
00290     for (int key : keys)
00291     {
00292         avl.insert({key, std::to_string(key)});
00293     }
00294     ASSERT_EQ(avl.size(), static_cast<size_t>(11));
00295
00296     // A remoção de uma chave distante (e.g., 70) pode forçar
00297     // o rebalanceamento em um nó ancestral.
00298     avl.remove(70);
00299     EXPECT_EQ(avl.size(), static_cast<size_t>(10));
00300     EXPECT_FALSE(avl.contains(70));
00301
00302     avl.remove(50);
00303     EXPECT_EQ(avl.size(), static_cast<size_t>(9));
00304     EXPECT_FALSE(avl.contains(50));
00305
00306     // Remover a raiz antiga deve funcionar
00307     avl.remove(40);
00308     EXPECT_EQ(avl.size(), static_cast<size_t>(8));
00309     EXPECT_FALSE(avl.contains(40));
00310 }
```

6.38.2.12 TYPED_TEST_P() [1/15]

```
TYPED_TEST_P (
             DictionaryTest ,
             At )
```

Testa a função 'at' para acesso e exceções.
Definido na linha 109 do ficheiro Tests.cpp.

```
00110 {
00111     this->dict->insert({1, "one"});
00112     EXPECT_EQ(this->dict->at(1), "one");
00113     EXPECT_THROW(this->dict->at(2), std::out_of_range);
00114
00115     // Modificar valor através da referência
00116     this->dict->at(1) = "uno";
00117     EXPECT_EQ(this->dict->at(1), "uno");
00118 }
```

6.38.2.13 TYPED_TEST_P() [2/15]

```
TYPED_TEST_P (
             DictionaryTest ,
             BracketOperator )
```

Testa o operador de colchetes [].
Definido na linha 132 do ficheiro Tests.cpp.

```
00133 {
00134     (*this->dict)[1] = "one"; // Inserir
00135     EXPECT_EQ(this->dict->size(), static_cast<size_t>(1));
00136     EXPECT_EQ((*this->dict)[1], "one");
00137
00138     (*this->dict)[1] = "uno"; // Atualizar
00139     EXPECT_EQ(this->dict->size(), static_cast<size_t>(1));
00140     EXPECT_EQ((*this->dict)[1], "uno");
00141
00142     (*this->dict)[2] = "two"; // Inserir
00143     EXPECT_EQ(this->dict->size(), static_cast<size_t>(2));
00144     EXPECT_EQ((*this->dict)[2], "two");
00145 }
```

6.38.2.14 TYPED_TEST_P() [3/15]

```
TYPED_TEST_P (
             DictionaryTest ,
             Clear )
```

Testa a limpeza do dicionário.
Definido na linha 148 do ficheiro Tests.cpp.

```
00149 {
00150     this->dict->insert({1, "one"});
00151     this->dict->insert({2, "two"});
00152     this->dict->clear();
00153     EXPECT_TRUE(this->dict->empty());
00154     EXPECT_EQ(this->dict->size(), static_cast<size_t>(0));
00155     EXPECT_FALSE(this->dict->contains(1));
00156 }
```

6.38.2.15 TYPED_TEST_P() [4/15]

```
TYPED_TEST_P (
             DictionaryTest ,
             Clone )
```

Testa a clonagem do dicionário.
Definido na linha 159 do ficheiro Tests.cpp.

```
00160 {
00161     this->dict->insert({1, "one"});
00162     this->dict->insert({2, "two"});
00163
00164     auto clone = this->dict->clone();
00165     ASSERT_NE(clone, nullptr);
00166
00167     // Verifica se não são o mesmo objeto
00168     EXPECT_NE(clone.get(), this->dict.get());
00169
```

```
00170     // Verifica se o conteúdo é o mesmo
00171     EXPECT_EQ(clone->size(), this->dict->size());
00172     EXPECT_TRUE(clone->contains(1));
00173     EXPECT_EQ(clone->at(1), "one");
00174     EXPECT_TRUE(clone->contains(2));
00175     EXPECT_EQ(clone->at(2), "two");
00176
00177     // Modifica o original e verifica se o clone não foi afetado (deep copy)
00178     this->dict->remove(1);
00179     EXPECT_FALSE(this->dict->contains(1));
00180     EXPECT_TRUE(clone->contains(1)); // O clone ainda deve ter o elemento
00181     EXPECT_EQ(clone->size(), static_cast<size_t>(2));
00182 }
```

6.38.2.16 TYPED_TEST_P() [5/15]

```
TYPED_TEST_P (
              DictionaryTest ,
              Contains )
```

Testa a funcionalidade de 'contains'.

Definido na linha 61 do ficheiro Tests.cpp.

```
00062 {
00063     this->dict->insert({10, "ten"});
00064     this->dict->insert({20, "twenty"});
00065     EXPECT_TRUE(this->dict->contains(10));
00066     EXPECT_TRUE(this->dict->contains(20));
00067     EXPECT_FALSE(this->dict->contains(30));
00068 }
```

6.38.2.17 TYPED_TEST_P() [6/15]

```
TYPED_TEST_P (
              DictionaryTest ,
              DefaultConstructor )
```

Testa o estado inicial de um dicionário recém-criado.

Definido na linha 37 do ficheiro Tests.cpp.

```
00038 {
00039     EXPECT_TRUE(this->dict->empty());
00040     EXPECT_EQ(this->dict->size(), static_cast<size_t>(0));
00041 }
```

6.38.2.18 TYPED_TEST_P() [7/15]

```
TYPED_TEST_P (
              DictionaryTest ,
              ForEach )
```

Testa a iteração com forEach.

Definido na linha 185 do ficheiro Tests.cpp.

```
00186 {
00187     this->dict->insert({3, "three"});
00188     this->dict->insert({1, "one"});
00189     this->dict->insert({2, "two"});
00190
00191     std::vector<int> keys;
00192     this->dict->forEach([&keys](const std::pair<int, std::string> &pair)
00193                         { keys.push_back(pair.first); });
00194
00195     // Para árvores, a ordem deve ser mantida. Para a tabela hash, não.
00196     // Então só verificamos se os elementos estão presentes.
00197     EXPECT_EQ(keys.size(), static_cast<size_t>(3));
00198     std::sort(keys.begin(), keys.end());
00199     EXPECT_EQ(keys[0], 1);
00200     EXPECT_EQ(keys[1], 2);
00201     EXPECT_EQ(keys[2], 3);
00202 }
```

6.38.2.19 TYPED_TEST_P() [8/15]

```
TYPED_TEST_P (
              DictionaryTest ,
              InsertAndSize )
```

Testa a inserção de um único elemento e o tamanho.
Definido na linha 44 do ficheiro Tests.cpp.

```
00045 {
00046     this->dict->insert({1, "one"});
00047     EXPECT_FALSE(this->dict->empty());
00048     EXPECT_EQ(this->dict->size(), static_cast<size_t>(1));
00049 }
```

6.38.2.20 TYPED_TEST_P() [9/15]

```
TYPED_TEST_P (
             DictionaryTest ,
             InsertDuplicates )
```

Testa se a inserção de chaves duplicadas é ignorada.
Definido na linha 52 do ficheiro Tests.cpp.

```
00053 {
00054     this->dict->insert({1, "one"});
00055     this->dict->insert({1, "uno"}); // Chave duplicada
00056     EXPECT_EQ(this->dict->size(), static_cast<size_t>(1));
00057     EXPECT_EQ(this->dict->at(1), "one"); // O valor original deve ser mantido
00058 }
```

6.38.2.21 TYPED_TEST_P() [10/15]

```
TYPED_TEST_P (
             DictionaryTest ,
             Remove )
```

Testa a remoção de elementos.
Definido na linha 71 do ficheiro Tests.cpp.

```
00072 {
00073     this->dict->insert({1, "one"});
00074     this->dict->insert({2, "two"});
00075     this->dict->insert({3, "three"});
00076
00077     this->dict->remove(2);
00078     EXPECT_EQ(this->dict->size(), static_cast<size_t>(2));
00079     EXPECT_FALSE(this->dict->contains(2));
00080     EXPECT_TRUE(this->dict->contains(1));
00081     EXPECT_TRUE(this->dict->contains(3));
00082
00083     // Tenta remover um elemento não existente
00084     this->dict->remove(42);
00085     EXPECT_EQ(this->dict->size(), static_cast<size_t>(2));
00086 }
```

6.38.2.22 TYPED_TEST_P() [11/15]

```
TYPED_TEST_P (
             DictionaryTest ,
             RemoveNodeWithTwoChildren )
```

Testa a remoção de um nó com dois filhos (importante para árvores)
Definido na linha 89 do ficheiro Tests.cpp.

```
00090 {
00091     // A ordem de inserção pode influenciar a estrutura da árvore
00092     this->dict->insert({20, "twenty"});
00093     this->dict->insert({10, "ten"});
00094     this->dict->insert({30, "thirty"});
00095     this->dict->insert({5, "five"});
00096     this->dict->insert({15, "fifteen"});
00097
00098     // Remover a raiz (20), que tem dois filhos
00099     this->dict->remove(20);
00100     EXPECT_EQ(this->dict->size(), static_cast<size_t>(4));
00101     EXPECT_FALSE(this->dict->contains(20));
00102     EXPECT_TRUE(this->dict->contains(10));
00103     EXPECT_TRUE(this->dict->contains(30));
00104     EXPECT_TRUE(this->dict->contains(5));
00105     EXPECT_TRUE(this->dict->contains(15));
00106 }
```

6.38.2.23 TYPED_TEST_P() [12/15]

```
TYPED_TEST_P (
             DictionaryTest ,
             Update )
```

Testa a função 'update'.

Definido na linha 121 do ficheiro Tests.cpp.

```
00122 {
00123     this->dict->insert({1, "one"});
00124     this->dict->update({1, "uno"});
00125     EXPECT_EQ(this->dict->at(1), "uno");
00126
00127     // 'update' em uma chave não existente deve lançar uma exceção
00128     EXPECT_THROW(this->dict->update({2, "dos"}), std::out_of_range);
00129 }
```

6.38.2.24 TYPED_TEST_P() [13/15]

```
TYPED_TEST_P (
             GeneralStressTest ,
             LargeRandomInsertAndRemove )
```

Definido na linha 439 do ficheiro Tests.cpp.

```
00440 {
00441     const int num_elements = 5000;
00442     std::vector<int> keys(num_elements);
00443     std::iota(keys.begin(), keys.end(), 0); // Preenche com 0, 1, 2, ...
00444
00445     // Embaralha as chaves para garantir inserção em ordem aleatória
00446     std::random_device rd;
00447     std::mt19937 g(rd());
00448     std::shuffle(keys.begin(), keys.end(), g);
00449
00450     // Insere todos os elementos
00451     for (int key : keys)
00452     {
00453         this->dict->insert({key, "value_" + std::to_string(key)});
00454     }
00455     ASSERT_EQ(this->dict->size(), static_cast<size_t>(num_elements));
00456
00457     // Verifica se todos foram inseridos
00458     for (int key : keys)
00459     {
00460         ASSERT_TRUE(this->dict->contains(key));
00461     }
00462
00463     // Embaralha novamente para remover em outra ordem aleatória
00464     std::shuffle(keys.begin(), keys.end(), g);
00465
00466     // Remove metade dos elementos
00467     for (size_t i = 0; i < keys.size() / 2; ++i)
00468     {
00469         this->dict->remove(keys[i]);
00470     }
00471     ASSERT_EQ(this->dict->size(), static_cast<size_t>(num_elements - keys.size() / 2));
00472
00473     // Verifica quais foram removidos e quais permaneceram
00474     for (size_t i = 0; i < keys.size(); ++i)
00475     {
00476         if (i < keys.size() / 2)
00477         {
00478             ASSERT_FALSE(this->dict->contains(keys[i]));
00479         }
00480         else
00481         {
00482             ASSERT_TRUE(this->dict->contains(keys[i]));
00483         }
00484     }
00485
00486     // Limpa a estrutura
00487     this->dict->clear();
00488     ASSERT_EQ(this->dict->size(), static_cast<size_t>(0));
00489     ASSERT_TRUE(this->dict->empty());
00490 }
```

6.38.2.25 TYPED_TEST_P() [14/15]

```
TYPED_TEST_P (
             HashTableStressTest ,
```

```
            HighCollisionRate )
```

Testa o comportamento sob alta taxa de colisão Para isso, precisamos de uma função de hash que possamos controlar. Como não podemos, vamos simular inserindo múltiplos de um valor inicial que, dependendo do tamanho da tabela, provavelmente colidirão.

Definido na linha 338 do ficheiro Tests.cpp.

```
00339 {
00340     // Supondo um tamanho inicial pequeno (e.g., 10),
00341     // inserir múltiplos de 10 causará colisões.
00342     const int num_elements = 20;
00343     const int step = 10; // Chaves: 0, 10, 20, 30...
00344     for (int i = 0; i < num_elements; ++i)
00345     {
00346         this->hashTable->insert({i * step, "value_" + std::to_string(i * step)});
00347     }
00348
00349     EXPECT_EQ(this->hashTable->size(), static_cast<size_t>(num_elements));
00350
00351     // Verifica se todos os elementos inseridos podem ser encontrados
00352     for (int i = 0; i < num_elements; ++i)
00353     {
00354         EXPECT_TRUE(this->hashTable->contains(i * step));
00355         EXPECT_EQ(this->hashTable->at(i * step), "value_" + std::to_string(i * step));
00356     }
00357
00358     // Tenta remover alguns elementos e verifica a consistência
00359     this->hashTable->remove(0);
00360     this->hashTable->remove(50);
00361     EXPECT_FALSE(this->hashTable->contains(0));
00362     EXPECT_FALSE(this->hashTable->contains(50));
00363     EXPECT_TRUE(this->hashTable->contains(10)); // Vizinho
00364     EXPECT_EQ(this->hashTable->size(), static_cast<size_t>(num_elements - 2));
00365 }
```

6.38.2.26 TYPED_TEST_P() [15/15]

```
TYPED_TEST_P (
            HashTableStressTest ,
            RehashingOnHighLoadFactor )
```

Testa a funcionalidade de redimensionamento (rehashing)

Definido na linha 368 do ficheiro Tests.cpp.

```
00369 {
00370     // Insere um número grande de elementos para forçar o rehash.
00371     // O rehash geralmente ocorre quando o fator de carga (size/capacity)
00372     // ultrapassa um limiar (e.g., 0.75).
00373     const int num_elements = 100;
00374     for (int i = 0; i < num_elements; ++i)
00375     {
00376         this->hashTable->insert({i, "value_" + std::to_string(i)});
00377     }
00378
00379     EXPECT_EQ(this->hashTable->size(), static_cast<size_t>(num_elements));
00380
00381     // Verifica se todos os dados permanecem consistentes após múltiplos rehashes.
00382     for (int i = 0; i < num_elements; ++i)
00383     {
00384         EXPECT_TRUE(this->hashTable->contains(i));
00385     }
00386
00387     // Remove metade dos elementos
00388     for (int i = 0; i < num_elements; i += 2)
00389     {
00390         this->hashTable->remove(i);
00391     }
00392
00393     EXPECT_EQ(this->hashTable->size(), static_cast<size_t>(num_elements / 2));
00394
00395     // Verifica se os elementos certos foram removidos e os outros permanecem
00396     for (int i = 0; i < num_elements; ++i)
00397     {
00398         if (i % 2 == 0)
00399         {
00400             EXPECT_FALSE(this->hashTable->contains(i));
00401         }
00402         else
00403         {
00404             EXPECT_TRUE(this->hashTable->contains(i));
00405         }
00406     }
00407 }
```

6.38.2.27 TYPED_TEST_SUITE_P() [1/3]

```
TYPED_TEST_SUITE_P (
            DictionaryTest )
```

Declaração da suíte de testes tipados.

6.38.2.28 TYPED_TEST_SUITE_P() [2/3]

```
TYPED_TEST_SUITE_P (
            GeneralStressTest )
```

6.38.2.29 TYPED_TEST_SUITE_P() [3/3]

```
TYPED_TEST_SUITE_P (
            HashTableStressTest )
```

6.39 Tests.cpp

Ir para a documentação deste ficheiro.

```
00001 #include "gtest/gtest.h"
00002 #include "dictionary/Dictionary.hpp"
00003 #include "dictionary/avl_tree/AVLTree.hpp"
00004 #include "dictionary/rb_tree/RedBlackTree.hpp"
00005 #include "dictionary/hash_table_c/ChainedHashTable.hpp"
00006 #include "dictionary/hash_table_o/OpenHashTable.hpp"
00007
00008 #include <string>
00009 #include <vector>
00010 #include <algorithm>
00011 #include <numeric>
00012 #include <random>
00013
00015 template <typename T>
00016 class DictionaryTest : public ::testing::Test
00017 {
00018 protected:
00019     std::unique_ptr<Dictionary<int, std::string>> dict;
00020
00021     void SetUp() override
00022     {
00023         // Cria uma nova instância da implementação de dicionário para cada teste
00024         dict = std::make_unique<T>();
00025     }
00026
00027     void TearDown() override
00028     {
00029         // unique_ptr lida com a desalocação automaticamente
00030     }
00031 };
00032
00034 TYPED_TEST_SUITE_P(DictionaryTest);
00035
00037 TYPED_TEST_P(DictionaryTest, DefaultConstructor)
00038 {
00039     EXPECT_TRUE(this->dict->empty());
00040     EXPECT_EQ(this->dict->size(), static_cast<size_t>(0));
00041 }
00042
00044 TYPED_TEST_P(DictionaryTest, InsertAndSize)
00045 {
00046     this->dict->insert({1, "one"});
00047     EXPECT_FALSE(this->dict->empty());
00048     EXPECT_EQ(this->dict->size(), static_cast<size_t>(1));
00049 }
00050
00052 TYPED_TEST_P(DictionaryTest, InsertDuplicates)
00053 {
00054     this->dict->insert({1, "one"});
00055     this->dict->insert({1, "uno"}); // Chave duplicada
00056     EXPECT_EQ(this->dict->size(), static_cast<size_t>(1));
00057     EXPECT_EQ(this->dict->at(1), "one"); // O valor original deve ser mantido
00058 }
00059
00061 TYPED_TEST_P(DictionaryTest, Contains)
00062 {
00063     this->dict->insert({10, "ten"});
00064     this->dict->insert({20, "twenty"});
00065     EXPECT_TRUE(this->dict->contains(10));
```

```
00066     EXPECT_TRUE(this->dict->contains(20));
00067     EXPECT_FALSE(this->dict->contains(30));
00068 }
00069
00071 TYPED_TEST_P(DictionaryTest, Remove)
00072 {
00073     this->dict->insert({1, "one"});
00074     this->dict->insert({2, "two"});
00075     this->dict->insert({3, "three"});
00076
00077     this->dict->remove(2);
00078     EXPECT_EQ(this->dict->size(), static_cast<size_t>(2));
00079     EXPECT_FALSE(this->dict->contains(2));
00080     EXPECT_TRUE(this->dict->contains(1));
00081     EXPECT_TRUE(this->dict->contains(3));
00082
00083     // Tenta remover um elemento não existente
00084     this->dict->remove(42);
00085     EXPECT_EQ(this->dict->size(), static_cast<size_t>(2));
00086 }
00087
00089 TYPED_TEST_P(DictionaryTest, RemoveNodeWithTwoChildren)
00090 {
00091     // A ordem de inserção pode influenciar a estrutura da árvore
00092     this->dict->insert({20, "twenty"});
00093     this->dict->insert({10, "ten"});
00094     this->dict->insert({30, "thirty"});
00095     this->dict->insert({5, "five"});
00096     this->dict->insert({15, "fifteen"});
00097
00098     // Remover a raiz (20), que tem dois filhos
00099     this->dict->remove(20);
00100     EXPECT_EQ(this->dict->size(), static_cast<size_t>(4));
00101     EXPECT_FALSE(this->dict->contains(20));
00102     EXPECT_TRUE(this->dict->contains(10));
00103     EXPECT_TRUE(this->dict->contains(30));
00104     EXPECT_TRUE(this->dict->contains(5));
00105     EXPECT_TRUE(this->dict->contains(15));
00106 }
00107
00109 TYPED_TEST_P(DictionaryTest, At)
00110 {
00111     this->dict->insert({1, "one"});
00112     EXPECT_EQ(this->dict->at(1), "one");
00113     EXPECT_THROW(this->dict->at(2), std::out_of_range);
00114
00115     // Modificar valor através da referência
00116     this->dict->at(1) = "uno";
00117     EXPECT_EQ(this->dict->at(1), "uno");
00118 }
00119
00121 TYPED_TEST_P(DictionaryTest, Update)
00122 {
00123     this->dict->insert({1, "one"});
00124     this->dict->update({1, "uno"});
00125     EXPECT_EQ(this->dict->at(1), "uno");
00126
00127     // 'update' em uma chave não existente deve lançar uma exceção
00128     EXPECT_THROW(this->dict->update({2, "dos"}), std::out_of_range);
00129 }
00130
00132 TYPED_TEST_P(DictionaryTest, BracketOperator)
00133 {
00134     (*this->dict)[1] = "one"; // Inserir
00135     EXPECT_EQ(this->dict->size(), static_cast<size_t>(1));
00136     EXPECT_EQ((*this->dict)[1], "one");
00137
00138     (*this->dict)[1] = "uno"; // Atualizar
00139     EXPECT_EQ(this->dict->size(), static_cast<size_t>(1));
00140     EXPECT_EQ((*this->dict)[1], "uno");
00141
00142     (*this->dict)[2] = "two"; // Inserir
00143     EXPECT_EQ(this->dict->size(), static_cast<size_t>(2));
00144     EXPECT_EQ((*this->dict)[2], "two");
00145 }
00146
00148 TYPED_TEST_P(DictionaryTest, Clear)
00149 {
00150     this->dict->insert({1, "one"});
00151     this->dict->insert({2, "two"});
00152     this->dict->clear();
00153     EXPECT_TRUE(this->dict->empty());
00154     EXPECT_EQ(this->dict->size(), static_cast<size_t>(0));
00155     EXPECT_FALSE(this->dict->contains(1));
00156 }
00157
00159 TYPED_TEST_P(DictionaryTest, Clone)
```

```
00160 {
00161     this->dict->insert({1, "one"});
00162     this->dict->insert({2, "two"});
00163 
00164     auto clone = this->dict->clone();
00165     ASSERT_NE(clone, nullptr);
00166 
00167     // Verifica se não são o mesmo objeto
00168     EXPECT_NE(clone.get(), this->dict.get());
00169 
00170     // Verifica se o conteúdo é o mesmo
00171     EXPECT_EQ(clone->size(), this->dict->size());
00172     EXPECT_TRUE(clone->contains(1));
00173     EXPECT_EQ(clone->at(1), "one");
00174     EXPECT_TRUE(clone->contains(2));
00175     EXPECT_EQ(clone->at(2), "two");
00176 
00177     // Modifica o original e verifica se o clone não foi afetado (deep copy)
00178     this->dict->remove(1);
00179     EXPECT_FALSE(this->dict->contains(1));
00180     EXPECT_TRUE(clone->contains(1)); // O clone ainda deve ter o elemento
00181     EXPECT_EQ(clone->size(), static_cast<size_t>(2));
00182 }
00183 
00185 TYPED_TEST_P(DictionaryTest, ForEach)
00186 {
00187     this->dict->insert({3, "three"});
00188     this->dict->insert({1, "one"});
00189     this->dict->insert({2, "two"});
00190 
00191     std::vector<int> keys;
00192     this->dict->forEach([&keys](const std::pair<int, std::string> &pair)
00193                         { keys.push_back(pair.first); });
00194 
00195     // Para árvores, a ordem deve ser mantida. Para a tabela hash, não.
00196     // Então só verificamos se os elementos estão presentes.
00197     EXPECT_EQ(keys.size(), static_cast<size_t>(3));
00198     std::sort(keys.begin(), keys.end());
00199     EXPECT_EQ(keys[0], 1);
00200     EXPECT_EQ(keys[1], 2);
00201     EXPECT_EQ(keys[2], 3);
00202 }
00203 
00205 REGISTER_TYPED_TEST_SUITE_P(DictionaryTest,
00206                             DefaultConstructor,
00207                             InsertAndSize,
00208                             InsertDuplicates,
00209                             Contains,
00210                             Remove,
00211                             RemoveNodeWithTwoChildren,
00212                             At,
00213                             Update,
00214                             BracketOperator,
00215                             Clear,
00216                             Clone,
00217                             ForEach);
00218 
00219 //================================================================================
00220 // TESTES ESPECÍFICOS PARA ÁRVORE AVL
00221 // Foco: Forçar todos os tipos de rotações na inserção e remoção.
00222 //================================================================================
00223 class AVLTreeSpecificTest : public ::testing::Test
00224 {
00225 protected:
00226     AVLTree<int, std::string> avl;
00227 };
00228 
00230 TEST_F(AVLTreeSpecificTest, InsertTriggersSingleRightRotation)
00231 {
00232     avl.insert({30, "thirty"});
00233     avl.insert({20, "twenty"});
00234     avl.insert({10, "ten"}); // Causa desbalanceamento LL no nó 30
00235 
00236     // Após a rotação, 20 deve ser a nova raiz.
00237     // A verificação é feita checando o tamanho e a presença das chaves,
00238     // mas o principal é garantir que a inserção não quebrou a estrutura.
00239     EXPECT_EQ(avl.size(), static_cast<size_t>(3));
00240     EXPECT_TRUE(avl.contains(10));
00241     EXPECT_TRUE(avl.contains(20));
00242     EXPECT_TRUE(avl.contains(30));
00243     // Um teste mais profundo (caixa-branca) poderia verificar quem é a raiz.
00244 }
00245 
00247 TEST_F(AVLTreeSpecificTest, InsertTriggersSingleLeftRotation)
00248 {
00249     avl.insert({10, "ten"});
00250     avl.insert({20, "twenty"});
```

```
00251     avl.insert({30, "thirty"}); // Causa desbalanceamento RR no nó 10
00252
00253     EXPECT_EQ(avl.size(), static_cast<size_t>(3));
00254     EXPECT_TRUE(avl.contains(10));
00255     EXPECT_TRUE(avl.contains(20));
00256     EXPECT_TRUE(avl.contains(30));
00257 }
00258
00260 TEST_F(AVLTreeSpecificTest, InsertTriggersRightLeftRotation)
00261 {
00262     avl.insert({10, "ten"});
00263     avl.insert({30, "thirty"});
00264     avl.insert({20, "twenty"}); // Causa desbalanceamento RL no nó 10
00265
00266     EXPECT_EQ(avl.size(), static_cast<size_t>(3));
00267     EXPECT_TRUE(avl.contains(10));
00268     EXPECT_TRUE(avl.contains(20));
00269     EXPECT_TRUE(avl.contains(30));
00270 }
00271
00273 TEST_F(AVLTreeSpecificTest, InsertTriggersLeftRightRotation)
00274 {
00275     avl.insert({30, "thirty"});
00276     avl.insert({10, "ten"});
00277     avl.insert({20, "twenty"}); // Causa desbalanceamento LR no nó 30
00278
00279     EXPECT_EQ(avl.size(), static_cast<size_t>(3));
00280     EXPECT_TRUE(avl.contains(10));
00281     EXPECT_TRUE(avl.contains(20));
00282     EXPECT_TRUE(avl.contains(30));
00283 }
00284
00286 TEST_F(AVLTreeSpecificTest, RemovalTriggersRebalancing)
00287 {
00288     // Cria uma árvore maior e mais complexa
00289     int keys[] = {40, 20, 60, 10, 30, 50, 70, 5, 15, 25, 35};
00290     for (int key : keys)
00291     {
00292         avl.insert({key, std::to_string(key)});
00293     }
00294     ASSERT_EQ(avl.size(), static_cast<size_t>(11));
00295
00296     // A remoção de uma chave distante (e.g., 70) pode forçar
00297     // o rebalanceamento em um nó ancestral.
00298     avl.remove(70);
00299     EXPECT_EQ(avl.size(), static_cast<size_t>(10));
00300     EXPECT_FALSE(avl.contains(70));
00301
00302     avl.remove(50);
00303     EXPECT_EQ(avl.size(), static_cast<size_t>(9));
00304     EXPECT_FALSE(avl.contains(50));
00305
00306     // Remover a raiz antiga deve funcionar
00307     avl.remove(40);
00308     EXPECT_EQ(avl.size(), static_cast<size_t>(8));
00309     EXPECT_FALSE(avl.contains(40));
00310 }
00311
00312 //============================================================================
00313 // TESTES ESPECÍFICOS PARA TABELAS HASH (AMBAS IMPLEMENTAÇÕES)
00314 // Foco: Forçar colisões e redimensionamento (rehashing).
00315 //============================================================================
00316
00319 template <typename T>
00320 class HashTableStressTest : public ::testing::Test
00321 {
00322 protected:
00323     std::unique_ptr<Dictionary<int, std::string» hashTable;
00324
00325     void SetUp() override
00326     {
00327         hashTable = std::make_unique<T>();
00328     }
00329 };
00330
00331 TYPED_TEST_SUITE_P(HashTableStressTest);
00332
00338 TYPED_TEST_P(HashTableStressTest, HighCollisionRate)
00339 {
00340     // Supondo um tamanho inicial pequeno (e.g., 10),
00341     // inserir múltiplos de 10 causará colisões.
00342     const int num_elements = 20;
00343     const int step = 10; // Chaves: 0, 10, 20, 30...
00344     for (int i = 0; i < num_elements; ++i)
00345     {
00346         this->hashTable->insert({i * step, "value_" + std::to_string(i * step)});
00347     }
```

```
00348
00349     EXPECT_EQ(this->hashTable->size(), static_cast<size_t>(num_elements));
00350
00351     // Verifica se todos os elementos inseridos podem ser encontrados
00352     for (int i = 0; i < num_elements; ++i)
00353     {
00354         EXPECT_TRUE(this->hashTable->contains(i * step));
00355         EXPECT_EQ(this->hashTable->at(i * step), "value_" + std::to_string(i * step));
00356     }
00357
00358     // Tenta remover alguns elementos e verifica a consistência
00359     this->hashTable->remove(0);
00360     this->hashTable->remove(50);
00361     EXPECT_FALSE(this->hashTable->contains(0));
00362     EXPECT_FALSE(this->hashTable->contains(50));
00363     EXPECT_TRUE(this->hashTable->contains(10)); // Vizinho
00364     EXPECT_EQ(this->hashTable->size(), static_cast<size_t>(num_elements - 2));
00365 }
00366
00368 TYPED_TEST_P(HashTableStressTest, RehashingOnHighLoadFactor)
00369 {
00370     // Insere um número grande de elementos para forçar o rehash.
00371     // O rehash geralmente ocorre quando o fator de carga (size/capacity)
00372     // ultrapassa um limiar (e.g., 0.75).
00373     const int num_elements = 100;
00374     for (int i = 0; i < num_elements; ++i)
00375     {
00376         this->hashTable->insert({i, "value_" + std::to_string(i)});
00377     }
00378
00379     EXPECT_EQ(this->hashTable->size(), static_cast<size_t>(num_elements));
00380
00381     // Verifica se todos os dados permanecem consistentes após múltiplos rehashes.
00382     for (int i = 0; i < num_elements; ++i)
00383     {
00384         EXPECT_TRUE(this->hashTable->contains(i));
00385     }
00386
00387     // Remove metade dos elementos
00388     for (int i = 0; i < num_elements; i += 2)
00389     {
00390         this->hashTable->remove(i);
00391     }
00392
00393     EXPECT_EQ(this->hashTable->size(), static_cast<size_t>(num_elements / 2));
00394
00395     // Verifica se os elementos certos foram removidos e os outros permanecem
00396     for (int i = 0; i < num_elements; ++i)
00397     {
00398         if (i % 2 == 0)
00399         {
00400             EXPECT_FALSE(this->hashTable->contains(i));
00401         }
00402         else
00403         {
00404             EXPECT_TRUE(this->hashTable->contains(i));
00405         }
00406     }
00407 }
00408
00409 REGISTER_TYPED_TEST_SUITE_P(HashTableStressTest,
00410                             HighCollisionRate,
00411                             RehashingOnHighLoadFactor);
00412
00413 using HashTableImplementations = ::testing::Types<
00414     ChainedHashTable<int, std::string>,
00415     OpenHashTable<int, std::string>>;
00416
00417 INSTANTIATE_TYPED_TEST_SUITE_P(MyHashImplementations, HashTableStressTest, HashTableImplementations);
00418
00419 //================================================================================
00420 // TESTE DE ESTRESSE GERAL (PARA TODAS AS IMPLEMENTAÇÕES)
00421 // Foco: Inserir e remover um grande volume de dados aleatórios.
00422 // Isso é ótimo para pegar vazamentos de memória e problemas de estabilidade.
00423 //================================================================================
00424
00425 template <typename T>
00426 class GeneralStressTest : public ::testing::Test
00427 {
00428 protected:
00429     std::unique_ptr<Dictionary<int, std::string» dict;
00430
00431     void SetUp() override
00432     {
00433         dict = std::make_unique<T>();
00434     }
00435 };
```

```
00436
00437 TYPED_TEST_SUITE_P(GeneralStressTest);
00438
00439 TYPED_TEST_P(GeneralStressTest, LargeRandomInsertAndRemove)
00440 {
00441     const int num_elements = 5000;
00442     std::vector<int> keys(num_elements);
00443     std::iota(keys.begin(), keys.end(), 0); // Preenche com 0, 1, 2, ...
00444
00445     // Embaralha as chaves para garantir inserção em ordem aleatória
00446     std::random_device rd;
00447     std::mt19937 g(rd());
00448     std::shuffle(keys.begin(), keys.end(), g);
00449
00450     // Insere todos os elementos
00451     for (int key : keys)
00452     {
00453         this->dict->insert({key, "value_" + std::to_string(key)});
00454     }
00455     ASSERT_EQ(this->dict->size(), static_cast<size_t>(num_elements));
00456
00457     // Verifica se todos foram inseridos
00458     for (int key : keys)
00459     {
00460         ASSERT_TRUE(this->dict->contains(key));
00461     }
00462
00463     // Embaralha novamente para remover em outra ordem aleatória
00464     std::shuffle(keys.begin(), keys.end(), g);
00465
00466     // Remove metade dos elementos
00467     for (size_t i = 0; i < keys.size() / 2; ++i)
00468     {
00469         this->dict->remove(keys[i]);
00470     }
00471     ASSERT_EQ(this->dict->size(), static_cast<size_t>(num_elements - keys.size() / 2));
00472
00473     // Verifica quais foram removidos e quais permaneceram
00474     for (size_t i = 0; i < keys.size(); ++i)
00475     {
00476         if (i < keys.size() / 2)
00477         {
00478             ASSERT_FALSE(this->dict->contains(keys[i]));
00479         }
00480         else
00481         {
00482             ASSERT_TRUE(this->dict->contains(keys[i]));
00483         }
00484     }
00485
00486     // Limpa a estrutura
00487     this->dict->clear();
00488     ASSERT_EQ(this->dict->size(), static_cast<size_t>(0));
00489     ASSERT_TRUE(this->dict->empty());
00490 }
00491
00492 REGISTER_TYPED_TEST_SUITE_P(GeneralStressTest,
00493                             LargeRandomInsertAndRemove);
00494
00495 using Implementations = ::testing::Types<
00496     AVLTree<int, std::string>,
00497     RedBlackTree<int, std::string>,
00498     ChainedHashTable<int, std::string>,
00499     OpenHashTable<int, std::string>>;
00500
00502 INSTANTIATE_TYPED_TEST_SUITE_P(MyImplementations, DictionaryTest, Implementations);
00503
00505 INSTANTIATE_TYPED_TEST_SUITE_P(MyGeneralStress, GeneralStressTest, Implementations);
```

6.40 Referência ao ficheiro tests/Tests.d

6.41 Tests.d

Ir para a documentação deste ficheiro.

```
00001 tests/Tests.o: tests/Tests.cpp \
00002  lib/googletest/googletest/include/gtest/gtest.h \
00003  lib/googletest/googletest/include/gtest/gtest-assertion-result.h \
00004  lib/googletest/googletest/include/gtest/gtest-message.h \
00005  lib/googletest/googletest/include/gtest/internal/gtest-port.h \
00006  lib/googletest/googletest/include/gtest/internal/custom/gtest-port.h \
00007  lib/googletest/googletest/include/gtest/internal/gtest-port-arch.h \
00008  lib/googletest/googletest/include/gtest/gtest-death-test.h \
```

```
00009  lib/googletest/googletest/include/gtest/internal/gtest-death-test-internal.h \
00010  lib/googletest/googletest/include/gtest/gtest-matchers.h \
00011  lib/googletest/googletest/include/gtest/gtest-printers.h \
00012  lib/googletest/googletest/include/gtest/internal/gtest-internal.h \
00013  lib/googletest/googletest/include/gtest/internal/gtest-filepath.h \
00014  lib/googletest/googletest/include/gtest/internal/gtest-string.h \
00015  lib/googletest/googletest/include/gtest/internal/gtest-type-util.h \
00016  lib/googletest/googletest/include/gtest/internal/custom/gtest-printers.h \
00017  lib/googletest/googletest/include/gtest/gtest-param-test.h \
00018  lib/googletest/googletest/include/gtest/internal/gtest-param-util.h \
00019  lib/googletest/googletest/include/gtest/gtest-test-part.h \
00020  lib/googletest/googletest/include/gtest/gtest-typed-test.h \
00021  lib/googletest/googletest/include/gtest/gtest_pred_impl.h \
00022  lib/googletest/googletest/include/gtest/gtest_prod.h \
00023  include/dictionary/Dictionary.hpp \
00024  include/dictionary/avl_tree/AVLTree.hpp \
00025  include/dictionary/avl_tree/Node.hpp \
00026  include/dictionary/avl_tree/IteratorAVL.hpp \
00027  include/dictionary/rb_tree/RedBlackTree.hpp \
00028  include/dictionary/rb_tree/NodeRB.hpp \
00029  include/dictionary/rb_tree/IteratorRB.hpp \
00030  include/dictionary/hash_table_c/ChainedHashTable.hpp \
00031  include/dictionary/hash_table_o/OpenHashTable.hpp \
00032  include/dictionary/hash_table_o/Slot.hpp
00033 lib/googletest/googletest/include/gtest/gtest.h:
00034 lib/googletest/googletest/include/gtest/gtest-assertion-result.h:
00035 lib/googletest/googletest/include/gtest/gtest-message.h:
00036 lib/googletest/googletest/include/gtest/internal/gtest-port.h:
00037 lib/googletest/googletest/include/gtest/internal/custom/gtest-port.h:
00038 lib/googletest/googletest/include/gtest/internal/gtest-port-arch.h:
00039 lib/googletest/googletest/include/gtest/gtest-death-test.h:
00040 lib/googletest/googletest/include/gtest/internal/gtest-death-test-internal.h:
00041 lib/googletest/googletest/include/gtest/gtest-matchers.h:
00042 lib/googletest/googletest/include/gtest/gtest-printers.h:
00043 lib/googletest/googletest/include/gtest/internal/gtest-internal.h:
00044 lib/googletest/googletest/include/gtest/internal/gtest-filepath.h:
00045 lib/googletest/googletest/include/gtest/internal/gtest-string.h:
00046 lib/googletest/googletest/include/gtest/internal/gtest-type-util.h:
00047 lib/googletest/googletest/include/gtest/internal/custom/gtest-printers.h:
00048 lib/googletest/googletest/include/gtest/gtest-param-test.h:
00049 lib/googletest/googletest/include/gtest/internal/gtest-param-util.h:
00050 lib/googletest/googletest/include/gtest/gtest-test-part.h:
00051 lib/googletest/googletest/include/gtest/gtest-typed-test.h:
00052 lib/googletest/googletest/include/gtest/gtest_pred_impl.h:
00053 lib/googletest/googletest/include/gtest/gtest_prod.h:
00054 include/dictionary/Dictionary.hpp:
00055 include/dictionary/avl_tree/AVLTree.hpp:
00056 include/dictionary/avl_tree/Node.hpp:
00057 include/dictionary/avl_tree/IteratorAVL.hpp:
00058 include/dictionary/rb_tree/RedBlackTree.hpp:
00059 include/dictionary/rb_tree/NodeRB.hpp:
00060 include/dictionary/rb_tree/IteratorRB.hpp:
00061 include/dictionary/hash_table_c/ChainedHashTable.hpp:
00062 include/dictionary/hash_table_o/OpenHashTable.hpp:
00063 include/dictionary/hash_table_o/Slot.hpp:
```

Índice