# Implementação e Análise Comparativa de Dicionários com Estruturas de Dados Avançadas

**Francisco Kelvy Monteiro de Lima - 569497**

[1] Ciência da Computação – Universidade Federal do Ceará (UFC)

kelvyemergencial@gmail.com

***Abstract.*** *This paper explores the implementation of a frequency counter using various advanced data structures in C++. The study focuses on comparing the performance of AVL Trees, Red-Black Trees, and Hash Tables (with external chaining and open addressing) as the underlying dictionary structures. Balanced trees guarantee logarithmic complexity (O(logn)) for insertion, search, and deletion, while hash tables typically offer constant-time access (O(1)) on average. The documentation delves into the theoretical concepts, generic implementations of each structure (including collision resolution and key occurrence counting), and the advantages/disadvantages of each approach. The aim is to provide a comprehensive guide for selecting the most suitable data structure based on specific data management application requirements.*

***Resumo.*** *Este trabalho explora a implementação de um contador de frequências utilizando diversas estruturas de dados avançadas em C++. O estudo foca na comparação de desempenho de Árvores AVL, Árvores Rubro-Negras e Tabelas Hash (com encadeamento exterior e endereçamento aberto) como base para um dicionário. As árvores balanceadas garantem complexidade logarítmica (O(logn)) para inserção, busca e remoção, enquanto as tabelas hash oferecem, em média, acesso constante (O(1)). A documentação detalha os conceitos teóricos, as implementações genéricas de cada estrutura (incluindo o tratamento de colisões e a contagem de ocorrências para chaves), e as vantagens/desvantagens de cada abordagem. O objetivo é fornecer um guia compreensível para a seleção da estrutura de dados mais adequada, considerando os requisitos específicos de cada aplicação de gestão de dados.*

## 1. Introdução

A gestão eficiente e rápida de dados é um alicerce fundamental no desenvolvimento de softwares cada vez mais robustos e performáticos. Em diversas aplicações, a necessidade de armazenar e recuperar informações associadas a chaves únicas é constante, tornando a estrutura de dicionário (ou mapa, em algumas nomenclaturas) um artefato indispensável. Um dicionário permite a inserção, busca e remoção de pares chave-valor, oferecendo acesso direto aos dados a partir de suas respectivas chaves.

Neste trabalho, visamos abordar a implementação de um dicionário em C++ explorando e comparando o desempenho em diferentes estruturas de dados, sendo elas: Árvores AVL, Árvores Rubro-Negras e Tabelas Hash (com tratamento de colisão por endereçamento exterior e tratamento de colisão por endereçamento aberto). As Árvores

AVL e Rubro-Negras são tipos de árvores binárias de busca autobalanceadas, garantindo complexidade logarítmica para as operações básicas que também serão demonstradas neste documento, mesmo em cenários de inserção ou remoção desfavoráveis. Por outro lado, as Tabelas Hash oferecem, em média, acesso constante aos dados, utilizando funções de hashing para mapear chaves a posições em um array.

Para as Tabelas Hash, serão exploradas duas abordagens distintas para a resolução de colisões: o tratamento de exceção (ou encadeamento separado) e o chaveamento aberto (ou endereçamento aberto). O tratamento de exceção lida com colisões armazenando múltiplos elementos em uma mesma posição da tabela através de listas encadeadas, enquanto o chaveamento aberto busca uma nova posição na própria tabela em caso de colisão, utilizando diferentes estratégias de sondagem.

Através desta documentação, será destrinchado conceitos teóricos de cada estrutura, junto a suas implementações em C++ e analisaremos as vantagens e desvantagens de cada uma no contexto de um dicionário, considerando métricas de tempo e espaço. O objetivo é fornecer um guia compreensivo para a escolha da estrutura de dados mais adequada, dependendo dos requisitos específicos de cada aplicação.

## 2. Introdução a usabilidade do progama

## 3. Árvore AVL

### 3.1. Introdução

Árvores AVL são um tipo especial de árvore binária de busca (ABB) que se autobalanceiam. Isso é super importante porque em ABBs comuns, se você inserir os dados numa ordem dita ”ruim”, a exemplo de inserir todos os elementos de um lado só, seja ela a direita ou esquerda, a árvore pode virar uma linha reta, se assemelhando a uma lista encadeada, com isso resultando em buscas mais lentas e ineficientes.

Com isso, na AVL garantimos que a diferença de altura entre a subárvore esquerda e a subárvore direita de qualquer nó nunca seja maior que 1. Para manter esse equilíbrio, após cada inserção ou remoção, a árvore verifica se algum nó ficou desbalanceado. Se sim, ela realiza rotações (movimentos especiais dos nós) para restaurar o balanço. Graças a isso, operações como buscar, inserir ou remover um item são sempre muito rápidas, levando um tempo proporcional ao logaritmo do número de itens na árvore **(O(logn))**, mesmo com a inserção massiva de dados. É essa característica que faz com que as Árvores AVL sejam uma escolha robusta para cenários onde o desempenho é crucial.

### 3.2. O Nó da Árvore AVL

```
    #ifndef NODE_H
#define NODE_H

#include <iostream>

template <typename T>
struct AVLNode {
    T key;
    int height;
```

```
    int contador; // contador de inserções
    Node* left;
    Node* right;

    // Construtor padrão
    Node(const T& k)
        : key(k), height(1), contador(1), left(nullptr), right(nullptr)
};

#endif // NODE_H
```

O nó em sua Árvore AVL, agora genérico (template ¡typename T¿), é a peça fundamental que armazena os dados. Ele contém a chave (key) (que pode ser de qualquer tipo comparável), a altura (height) de sua subárvore para o balanceamento, e um novo contador (contador) que permite rastrear múltiplas ocorrências da mesma chave. Além disso, possui os ponteiros left e right para seus filhos. O construtor inicializa a chave, define a altura e o contador como 1, e aponta os filhos para nulo. Essa estrutura flexível e completa permite que sua AVL gerencie eficientemente diversos tipos de dados, enquanto mantém o auto-balanceamento e a contagem de elementos duplicados.

### 3.3. Inserção e Funções Auxiliares

A inserção de um novo nó em uma Árvore AVL começa de forma semelhante à de uma Árvore Binária de Busca (ABB) comum. O novo elemento é sempre adicionado como uma folha, seguindo a regra de que chaves menores vão para a esquerda e chaves maiores vão para a direita.

No entanto, o diferencial da AVL surge após essa inserção inicial. Para manter a propriedade de auto-balanceamento que garante a eficiência da árvore, é crucial verificar se a adição do novo nó causou algum desequilíbrio. Esse processo envolve recalcular as alturas dos nós afetados, do nó inserido até a raiz, e identificar qualquer nó cujo fator de balanceamento (a diferença entre as alturas das suas subárvores esquerda e direita) exceda o limite permitido de ±1. Se um desequilíbrio for detectado, uma ou mais rotações (simples ou duplas) serão executadas para restaurar o balanço da árvore, garantindo que ela permaneça otimizada para futuras operações.

```
    void Insert(const KeyType& key, const ValueType& value)
```

A função Insert atua como a interface pública para adicionar novos dados à sua árvore. Quando você a invoca, ela imediatamente delega a tarefa à sua contraparte recursiva e privada, insert, passando a raiz da árvore e o par chave-valor a ser inserido.Observe que ela atribui o resultado de insert de volta à root. Isso é fundamental porque, durante o processo de inserção e balanceamento, a própria raiz da árvore pode mudar se ocorrerem rotações no nível superior.

```
    AVLNode<KeyType, ValueType>*
    _insert(AVLNode<KeyType, ValueType>*
    node, const KeyType& key, const ValueType& value)
```

A função AVLNode¡KeyType, ValueType¿* insert(AVLNode¡KeyType, ValueType¿* node, const KeyType key, const ValueType value) é o método recursivo que gerencia a adição de novos elementos (pares chave-valor) na sua árvore AVL, garantindo que

ela permaneça balanceada. Localização e Inserção: Ela percorre a árvore, comparando a key a ser inserida com as chaves existentes, descendo à esquerda ou à direita. Se encontrar um local vazio (!node), um novo nó é criado e inserido ali. O tamanho da árvore (size) é incrementado. Se a key já existir, o value do nó existente é simplesmente atualizado, sem criar um novo nó ou alterar a estrutura da árvore. Atualização da Altura: Conforme a recursão retorna, a altura de cada nó é recalculada com base nas alturas de seus filhos. Balanceamento (Rotações AVL): Após a altura ser atualizada, o fator de balanceamento de cada nó é verificado. Se um nó estiver desbalanceado (fator de balanceamento maior que 1 ou menor que -1), as rotações AVL apropriadas (simples ou duplas) são executadas para restaurar o equilíbrio da subárvore, garantindo a eficiência da árvore. As rotações são contabilizadas em mrotations. Em resumo, insert insere/atualiza um elemento, atualiza alturas e rebalanceia a árvore AVL via rotações, se necessário, garantindo sua eficiência O(logn).

```
int height(AVLNode<KeyType, ValueType>* node) const
```

Uma função utilitária simples que retorna a altura de um dado nó. Se o nó for nulo, sua altura é 0.

```
int balance(AVLNode<KeyType, ValueType>* node) const
```

Calcula o fator de balanceamento de um nó. A diferença crucial entre as alturas das subárvores esquerda e direita. É este valor que acusa um desbalanceamento.

```
AVLNode<KeyType, ValueType>*
right_rotation(AVLNode<KeyType,
ValueType>* y)
```

Executa a rotação mais básica para a direita. Essencial para corrigir desbalanceamentos onde a subárvore esquerda está muito alta (LL ou parte de LR). Ela ”gira” o nó y para a direita em torno de seu filho esquerdo x.

```
AVLNode<KeyType, ValueType>*
left_rotation(AVLNode<KeyType,
ValueType>* x):
```

Executa a rotação mais básica para a esquerda. Essencial para corrigir desbalanceamentos onde a subárvore direita está muito alta (RR ou parte de RL). Ela ”gira” o nó x para a esquerda em torno de seu filho direito y.

### 3.4. Busca na Árvore AVL

A busca em uma Árvore AVL funciona como em qualquer Árvore Binária de Busca: você começa na raiz e vai para a esquerda se a chave procurada for menor, e para a direita se for maior, até encontrar o elemento ou um caminho vazio. O grande diferencial da AVL é que ela se mantém balanceada. Isso garante que a busca seja sempre muito rápida e eficiente, com tempo de execução de O(logn), sem o risco de ficar lenta como em árvores desbalanceadas.

```
bool Contains(const KeyType& key) const
```

Ela serve como uma porta de entrada pública para a busca de existência. Sua principal responsabilidade é iniciar a busca a partir da raiz da árvore (root), delegando o trabalho real à função auxiliar recursiva contains. O const no final indica que esta função não modifica o estado do objeto Set, apenas o consulta

```
bool _contains(AVLNode<KeyType, ValueType>*
node, const KeyType& key) const
```

Esta é a função privada e recursiva que implementa a lógica real da busca. Ela também é const.Se o node atual for nullptr, significa que a busca chegou ao fim sem encontrar a key, então retorna false. Se a key procurada for igual à chave do node atual, o elemento foi encontrado e retorna true.Caso contrário, a busca continua recursivamente: para a subárvore esquerda (node-¿left) se a key for menor, ou para a subárvore direita (node-¿right) se for maior. Graças ao balanceamento da AVL, esta operação tem uma complexidade de tempo de O(logn), o que a torna altamente eficiente.

```
ValueType getCount(const KeyType& key) const
```

Diferente de contains que é recursiva, getCount é iterativa. Ela começa na root e segue o caminho apropriado (esquerda se menor, direita se maior) até encontrar a key desejada ou um nullptr. Se a key for encontrada, ela retorna o value associado àquela chave. Se a key não for encontrada (o loop termina porque curr se tornou nullptr), ela retorna um valor padrão (ValueType). Para tipos numéricos, isso geralmente é 0; para ponteiros, é nullptr; para objetos, é um objeto default-construído. Isso evita lançar uma exceção e fornece um valor ”seguro” quando a chave não está presente. Cada comparação de chave durante a travessia incrementa mcomparisons.

```
std::pair<KeyType, ValueType> Minimum() const:
```

Retorna o menor elemento (par chave-valor) da árvore. Internamente, chama findmin(root)

```
AVLNode<KeyType, ValueType>*
find_min(AVLNode<KeyType,
ValueType>* node) const
```

Esta função auxiliar encontra o nó com a menor chave em uma dada subárvore. Como o menor elemento em uma BST/AVL está sempre no caminho mais à esquerda, ela simplesmente percorre os filhos esquerdos até chegar a um nó sem filho esquerdo. Cada passo incrementa mcomparisons. Lança std::runtimeerror se a árvore estiver vazia.

```
std::pair<KeyType, ValueType> Maximum() const
```

Retorna o maior elemento (par chave-valor) da árvore. Internamente, chama findmax(root).

```
AVLNode<KeyType, ValueType>*
find_max(AVLNode<KeyType, ValueType>* node)
const
```

Similar a findmin, mas percorre os filhos direitos para encontrar o nó com a maior chave. Cada passo incrementa mcomparisons. Lança std::runtimeerror se a árvore estiver vazia.

```
std::pair<KeyType, ValueType>
Successor(const KeyType& key) const:
```

Encontra o elemento cuja chave é a próxima em ordem crescente após a key fornecida.

```
AVLNode<KeyType, ValueType>*
find_successor(AVLNode<KeyType, ValueType>*
node, const KeyType& key) const
```

Esta função auxiliar busca o sucessor in-order. Ela percorre a árvore, mantendo um ponteiro succ para o nó mais à esquerda que é maior que key e que foi encontrado até agora. Se a key do nó atual for menor ou igual à key buscada, ela vai para a direita; caso contrário, o nó atual é um potencial sucessor, então succ é atualizado e a busca continua à esquerda. Incrementa mcomparisons. Lança std::runtimeerror se não houver sucessor.

```
std::pair<KeyType, ValueType>
Predecessor(const KeyType& key)
const
```

Encontra o elemento cuja chave é a anterior em ordem crescente à key fornecida.

```
AVLNode<KeyType, ValueType>* f
ind_predecessor(AVLNode<KeyType,
ValueType>* node, const KeyType& key) const
```

Similar a findsuccessor, mas busca o predecessor. Ela percorre a árvore, mantendo um ponteiro pred para o nó mais à direita que é menor que key e que foi encontrado até agora. Se a key do nó atual for maior ou igual à key buscada, ela vai para a esquerda; caso contrário, o nó atual é um potencial predecessor, então pred é atualizado e a busca continua à direita. Incrementa mcomparisons. Lança std::runtimeerror se não houver predecessor.

```
bool Empty() const
```

Retorna true se a árvore não contém elementos (root == nullptr), e false caso contrário

```
int Size() const
```

Retorna o número total de elementos (pares chave-valor únicos) atualmente armazenados na árvore

```
void PrintInOrder(std::function<void
(const KeyType&, const ValueType&)> func) const
```

Permite que você percorra a árvore em ordem crescente de chaves (percurso in-order) e execute uma função (func) para cada par chave-valor. Isso é útil para exibir os elementos ordenadamente ou para realizar alguma operação em cada um deles.

```
void in_order_print(AVLNode<KeyType,
ValueType>* node, const
std::function<void(const KeyType&,
const ValueType&)>& func) const
```

A função auxiliar recursiva que implementa o percurso in-order: visita a subárvore esquerda, processa o nó atual, depois visita a subárvore direita.

```
std::vector<std::pair<KeyType,
ValueType>> ToVector() const
```

Converte todos os elementos da árvore em um std::vector de std::pair¡KeyType, ValueType¿, com os elementos já ordenados por chave. void inordercollect(AVLNode¡KeyType, ValueType¿* node, std::vector¡std::pair¡KeyType, ValueType¿¿ vec) const: A função auxiliar recursiva que preenche o vetor durante um percurso in-order.

```
void bshow() const
```

Esta função é inestimável para depuração e compreensão visual da estrutura da sua árvore AVL. Ela imprime uma representação gráfica da árvore no console, mostrando a chave, o valor, a altura e o fator de balanceamento de cada nó. Isso ajuda a verificar se a árvore está realmente balanceada e se as operações estão funcionando como esperado.

### 3.5. Remoção na Árvore AVL

A remoção de um nó em uma Árvore AVL segue a lógica básica de uma Árvore Binária de Busca. O desafio principal surge após a remoção: a árvore pode ficar desbalanceada. Para corrigir isso, o algoritmo verifica o fator de balanceamento dos nós afetados, da remoção até a raiz. Se houver desequilíbrio, rotações (simples ou duplas) são realizadas para restaurar o balanço da árvore. Esse processo garante que, mesmo após a remoção, a operação mantenha uma complexidade de tempo de O(logn), crucial para a eficiência da AVL.

```
void Erase(const KeyType& key)
```

Esta é a interface que você utiliza para remover um par chave-valor específico da árvore. Ela é direta: recebe a key do elemento a ser removido e invoca a função recursiva erase, passando a raiz da árvore.Assim como na inserção, o resultado de erase é atribuído de volta à root. Isso é crucial porque, após a remoção e as potenciais rotações de rebalanceamento, a raiz da árvore pode ter sido alterada.

```
AVLNode<KeyType, ValueType>*
_erase(AVLNode<KeyType, ValueType>*
node, const KeyType& key)
```

Aqui é onde a complexidade da remoção se manifesta. Esta função recursiva busca o nó a ser removido e, após a remoção, garante que a propriedade AVL seja restaurada através de rebalanceamentos.

```
void Clear()
```

Ela efetivamente ”esvazia” a árvore. Chama a função auxiliar destroy(root) para recursivamente deletar todos os nós da árvore, liberando a memória alocada. Redefine root para nullptr, indicando que a árvore está vazia. Reseta o size para 0 e os contadores de mcomparisons e mrotations.

```
void destroy(AVLNode<KeyType, ValueType>* node)
```

Esta função privada e recursiva é usada para liberar a memória de todos os nós de uma subárvore, a partir de um node dado. Ela é tipicamente utilizada no destrutor da classe ( Set()) ou em uma função Clear() para garantir que não haja vazamento de memória quando o conjunto é descartado ou esvaziado.

```
AVLNode<KeyType, ValueType>*
find_min(AVLNode<KeyType, ValueType>* node)
const
```

### 3.6. Comparações: Principal e rotações

A classe Set utiliza duas variáveis do tipo long long para monitorar a eficiência das operações na Árvore AVL, registrando o número de comparações principais(comparaçes entre chaves) e rotações(quantas vezes ocorre uma rotação).

```
mutable long long comparacoes_principal
```

Para a variavel comparacoesprincipal (do tipo mutable long long): Esta variável conta as comparações de chaves que acontecem durante as operações essenciais da árvore AVL, como inserção (insert), verificação de existência (contains), e busca por sucessores e predecessores (findsuccessor, findpredecessor). O modificador mutable é importante aqui, pois permite que essa contagem seja atualizada mesmo em métodos marcados como const, que normalmente não poderiam modificar membros do objeto. Isso é útil para funções que não mudam o ”estado” lógico do conjunto, mas que precisam registrar uma ação para fins de análise de desempenho.

```
long long comparacoes_rotacoes
```

Para a variavel comparacoesrotacoes (do tipo long long): Este contador é específico para as operações de rebalanceamento da árvore. Ele registra o número de rotações (sejam elas à direita ou à esquerda) que ocorrem para manter a propriedade AVL após inserções ou remoções de elementos. Cada vez que uma função de rotação (rightrotation ou leftrotation) é executada, comparacoesrotacoes é incrementado. Isso nos dá uma medida direta do custo de manutenção do balanceamento da árvore.

Para que você possa acessar os valores desses contadores e redefini-los para novas medições, a classe Set oferece as seguintes funções:

```
long long getComparacoes_principal()
const { return comparacoes_principal; }
```

Retorna o número atual de comparações de chaves registradas por comparacoesprincipal. Use esta função para saber quantas comparações foram feitas desde o último reset, tendo asim como retorno um valor long long que representa a contagem de comparações principais.

```
void resetComparacoes() { comparacoes_principal = 0; }
```

Zera o contador comparacoesprincipal. Isso é ideal para iniciar uma nova medição, garantindo que as comparações de operações anteriores não interfiram na análise de uma operação específica.

```
long long getComparacoes_rotacoes()
const { return comparacoes_rotacoes; }
```

Esta função retorna o número atual de rotações registradas por comparacoesrotacoes, retornando um valor long long que representa a contagem de rotações.

```
void resetComparacoesRotacoes()
{ comparacoes_rotacoes = 0; }
```

Zera o contador comparacoesrotacoes. Similar ao resetComparacoes(), permite que você isole a contagem de rotações para analisar o impacto do rebalanceamento em cenários específicos.

### 3.7. Considerações Finais Sobre AVL

A Árvore AVL é uma Árvore Binária de Busca autobalanceada. Sua principal vantagem é garantir que todas as operações (inserção, busca e remoção) tenham uma complexidade de tempo de O(logn), independentemente da ordem dos dados. Embora sua implementação envolva a complexidade das rotações para manter o balanceamento, essa característica assegura um desempenho consistente e previsível. Isso faz da AVL uma escolha robusta e eficiente para dicionários, especialmente onde a velocidade e a confiabilidade das operações são críticas.

## 4. Árvore Rubro-Negra

### 4.1. Introdução

Árvores Rubro-Negras são outro tipo de árvore binária de busca autobalanceada, como as AVL, que garantem operações eficientes (O(logn)). A diferença é que elas mantêm o balanço seguindo cinco propriedades específicas de ”cores” (vermelho ou preto) para cada nó, em vez de fatores de balanceamento estritos. Após cada inserção ou remoção, a árvore usa rotações e recolorações para restaurar essas propriedades. Essa abordagem pode resultar em menos rotações que as AVL em alguns casos, tornando-as ligeiramente mais eficientes para muitas modificações.

### 4.2. O Nó da Árvore Rubro-Negra

O nó de uma Árvore Rubro-Negra é a sua unidade básica e crucial. Além da chave (key) que armazena o dado (permitindo diferentes tipos, como indicado pelo template typename T), cada nó possui uma cor (color), que pode ser vermelha ou preta. Essa cor é fundamental para o sistema de balanceamento da árvore. Ele também inclui um contador de ocorrências, útil para chaves duplicadas. Por fim, para auxiliar nas operações de balanceamento, cada nó mantém ponteiros para seus filhos esquerdos (left) e direitos (right), e um ponteiro para seu pai (parent). O construtor facilita a criação de novos nós com essas propriedades iniciais, e um destrutor simples informa quando um nó é removido. Essa estrutura detalhada do nó é o que permite à Árvore Rubro-Negra manter seu rigoroso equilíbrio e garantir operações eficientes.

```
    #ifndef NODE_HPP
#define NODE_HPP
#include <iostream>

#define RED true
#define BLACK false

template <typename T>
class RBNode {
    public:

    bool color;
    int ocorrencias = 1;
    T key;
    Node* left;
    Node* right;
    Node* parent;

    Node(T key, bool color, Node* l, Node* r, Node* parent){
        this->key = key;
        this->color = color;
        left = l;
        right = r;
        this->parent = parent;
    }

    ~Node() {
    std::cout << "O nó (" << key.first << ", " << key.second << ") foi
}

};

#endif
```

### 4.3. Inserção Na Árvore Rubro-Negra

A inserção de um novo nó em uma Árvore Rubro-Negra começa de forma semelhante à de uma Árvore Binária de Busca (ABB) comum: o novo nó é adicionado como uma folha, seguindo as regras de ordenação de chaves (menores à esquerda, maiores à direita). O diferencial é que, inicialmente, todo novo nó é inserido como vermelho.

Essa cor inicial é estratégica. Embora possa violar as propriedades da Árvore Rubro-Negra (especialmente a que proíbe dois nós vermelhos consecutivos), ela simplifica o processo. Após a inserção, o algoritmo de inserção se concentra em restaurar as propriedades da árvore. Isso é feito através de uma série de verificações e, se necessário, rotações (iguais às da AVL) e recolorações de nós. O objetivo é garantir que a árvore retorne a um estado balanceado e consistente com todas as cinco propriedades das Árvores Rubro-Negras, mantendo a eficiência logarítmica (O(logn)) para a operação.

```
void insert(const Key& key, const Value& value)
```

Esta é a função pública que você chama para adicionar um novo par (key, value) à árvore. Ela orquestra o processo de encontrar o local correto, criar o novo nó e, em seguida, chamar o mecanismo de fixup para manter as propriedades da RBTree.

```
void insertFixup(RBNode<Pair>* z)
```

Esta função é a inteligência por trás do auto-balanceamento da RBTree após uma inserção. Ela é chamada quando um nó z (o nó recém-inserido, que é VERMELHO) pode ter causado uma violação. A única violação possível ao inserir um nó VERMELHO é a Propriedade 4: "Se um nó é vermelho, então ambos os seus filhos são pretos". Se o pai de z também for VERMELHO, temos uma dupla vermelha, o que é uma violação.

```
void insertFixup(Node<T>* z):
Correção Pós-Inserção (Privada)
```

Esta é a função privada e crucial que restaura as propriedades da Árvore Rubro-Negra após a inserção de um nó vermelho. Ela recebe o nó recém-inserido (z) e, através de uma série de verificações (casos de violação), executa rotações (esquerda e direita) e recolorações de nós. O objetivo é eliminar qualquer sequência de dois nós vermelhos e garantir que o número de nós pretos nos caminhos até as folhas seja consistente, mantendo assim o balanceamento.

```
void leftRotate(RBNode<Pair>* x)
```

Realiza uma rotação simples à esquerda. Essa operação reestrutura um segmento da árvore para mover x para baixo e seu filho direito y para cima, mantendo a ordem dos elementos. Incrementa comparacoesrotacoes.

```
void rightRotate(RBNode<Pair>* y)
```

Realiza uma rotação simples à direita. Esta é a operação simétrica à leftRotate, movendo y para baixo e seu filho esquerdo x para cima. Incrementa comparacoesrotacoes.

```
RBNode<Pair>* create_nil_node()
```

Cria e retorna o nó sentinela nil. Este nó é fundamental em RB Trees; ele representa todas as "folhas" vazias da árvore, e é sempre PRETO. Todos os ponteiros nulos em uma implementação típica de RBTree apontam para este único nó sentinela.

### 4.4. Busca Na Árvore Rubro-Negra

A busca em uma Árvore Rubro-Negra é idêntica à busca em qualquer Árvore Binária de Busca comum: você começa na raiz e se move para a esquerda se a chave procurada for menor, ou para a direita se for maior, até encontrar o nó desejado ou um ponto nulo. O benefício crucial das Árvores Rubro-Negras é que seu balanceamento intrínseco (mantido pelas regras de cores e rotações) garante que essa busca seja sempre extremamente eficiente, com uma complexidade de tempo de O(logn), mesmo em árvores muito grandes.

```
bool contains(const Key& key) const
```

Esta é a função pública que o usuário final utiliza para verificar a existência de uma key no conjunto. Sendo const, ela não altera o estado da árvore. Sua principal função é iniciar o processo de busca, chamando a função auxiliar search a partir da raiz da árvore e retornando o resultado booleano (verdadeiro se a chave for encontrada, falso caso contrário).

```
RBNode<Pair>* search(RBNode<Pair>*
node, const Key& key) const
```

Esta função privada e recursiva é a que encontra a chave na árvore. Ela avança pela árvore (esquerda para chaves menores, direita para maiores) até achar o nó com a key ou um ponto vazio. Sua eficiência é garantida em O(logn), graças ao balanceamento da Árvore Rubro-Negra.

```
RBNode<Pair>* searchNode(const Key& key) const
```

Esta função permite que o usuário obtenha um ponteiro direto para o nó que contém a chave, se ela existir. Internamente, ela também chama search(root, key). No entanto, em vez de retornar nil (que é um detalhe de implementação interna), ela o traduz para nullptr, que é uma convenção mais comum para indicar um ponteiro ”nulo” ou ”não encontrado” em interfaces públicas de C++. Isso dá ao usuário acesso ao próprio nó para inspecionar seu valor e número de ocorrências.

```
bool empty() const
```

Embora não seja uma busca por uma chave específica, esta função consulta o estado da árvore para saber se ela contém elementos. Ela retorna true se a root aponta para o nó sentinela nil, indicando que a árvore está vazia.

### 4.5. Remoção Na Árvore Rubro-Negra

A remoção de um nó em uma Árvore Rubro-Negra é o processo mais complexo, pois, ao retirar um elemento, as cinco propriedades de balanceamento da árvore podem ser violadas. O algoritmo identifica o nó a ser removido (que pode ser substituído por seu sucessor ou predecessor se tiver dois filhos) e, após a remoção física, a árvore precisa ser rebalanceada. Isso é feito através de uma série de rotações e recolorações específicas, aplicadas de forma iterativa ou recursiva, para garantir que todas as propriedades da Árvore Rubro-Negra sejam restauradas. Assim como nas outras operações, a remoção mantém uma complexidade de tempo de O(logn), o que assegura a eficiência da estrutura.

```
void remove(const Key& key)
```

Esta é a função pública que o usuário do conjunto chama para remover uma chave específica da árvore. Ela inicia o processo de remoção, atuando como uma interface para a lógica interna e garantindo que a raiz da árvore seja atualizada após as modificações.

```
void removeFixup(RBNode<Pair>* x)
```

Esta função é a mais complexa de todas em árvores Rubro-Negras. Ela é ativada quando um nó PRETO é removido, o que pode causar um desbalanceamento no número de nós pretos nos caminhos (Propriedade 5). O nó x é o ponto de partida do desbalanceamento (ele ”subiu” para o lugar de um nó preto).

```
void transplant(RBNode<Pair>* u,
RBNode<Pair>* v)
```

Esta é uma função auxiliar crucial. Ela substitui a subárvore enraizada em u pela subárvore enraizada em v. Ela lida com a ligação dos pais de u para v, efetivamente movendo v para a posição de u.

```
RBNode<Pair>* minimum(RBNode<Pair>* node)
```

Usada para encontrar o sucessor in-order, um passo vital na remoção de nós com dois filhos. Percorre para a esquerda até encontrar o nó mais à esquerda.

```
void clearInternal(RBNode<Pair>* node)
```

Chamada por clear() e pelo destrutor ( rbtree()). Esta função recursiva libera a memória de todos os nós da árvore em um percurso pós-ordem, garantindo que não haja vazamentos de memória.

### 4.6. Comparações: Principal e rotações

A classe RBTree utiliza duas variáveis do tipo long long para monitorar a eficiência das operações na Árvore Rubro-Negra, registrando o número de comparações principais(comparaçes entre chaves) e rotações(quantas vezes ocorre uma rotação).

```
mutable long long comparacoes_principal
```

Para a variavel comparacoesprincipal (do tipo mutable long long): Esta variável conta as comparações de chaves que acontecem durante as operações essenciais da árvore Rubro-negra, como inserção (insert), verificação de existência (contains), e busca por sucessores e predecessores (findsuccessor, findpredecessor). O modificador mutable é importante aqui, pois permite que essa contagem seja atualizada mesmo em métodos marcados como const, que normalmente não poderiam modificar membros do objeto. Isso é útil para funções que não mudam o ”estado” lógico do conjunto, mas que precisam registrar uma ação para fins de análise de desempenho.

```
long long comparacoes_rotacoes
```

Para a variavel comparacoesrotacoes (do tipo long long): Este contador é específico para as operações de rebalanceamento da árvore. Ele registra o número de rotações (sejam elas à direita ou à esquerda) que ocorrem para manter a propriedade Rubro-negra após inserções ou remoções de elementos. Cada vez que uma função de rotação (rightrotation ou leftrotation) é executada, comparacoesrotacoes é incrementado. Isso nos dá uma medida direta do custo de manutenção do balanceamento da árvore.

Para que você possa acessar os valores desses contadores e redefini-los para novas medições, a classe RBTree oferece as seguintes funções:

```
long long getComparacoes_principal()
const { return comparacoes_principal; }
```

Retorna o número atual de comparações de chaves registradas por comparacoesprincipal. Use esta função para saber quantas comparações foram feitas desde o último reset, tendo asim como retorno um valor long long que representa a contagem de comparações principais.

```
void resetComparacoes() { comparacoes_principal = 0; }
```

Zera o contador comparacoesprincipal. Isso é ideal para iniciar uma nova medição, garantindo que as comparações de operações anteriores não interfiram na análise de uma operação específica.

```
long long getComparacoes_rotacoes()
const { return comparacoes_rotacoes; }
```

Esta função retorna o número atual de rotações registradas por comparacoesrotacoes, retornando um valor long long que representa a contagem de rotações.

```
void resetComparacoesRotacoes() { comparacoes_rotacoes = 0; }
```

Zera o contador comparacoesrotacoes. Similar ao resetComparacoes(), permite que você isole a contagem de rotações para analisar o impacto do rebalanceamento em cenários específicos.

### 4.7. Considerações Finais Árvore Rubro - Negra

A Árvore Rubro-Negra (RBN) é uma estrutura de dados autobalanceada fundamental, especialmente eficiente para implementar dicionários (como o std::map do C++). Ela garante que operações como inserção, busca e remoção tenham sempre uma complexidade de tempo de O(logn), o que a torna altamente performática, mesmo com muitos dados.

Diferente de outras árvores, a RBN mantém seu balanço através de propriedades de ”cores” (vermelho/preto) e um conjunto de rotações e recolorações após cada modificação. Embora essa lógica possa parecer complexa, ela resulta em um desempenho muito consistente e previsível, sendo por isso a escolha padrão em muitas bibliotecas e sistemas que exigem eficiência e confiabilidade na manipulação de dados associativos.

## 5. Tabelas Hash com Endereçamento Exterior

### 5.1. Introdução a Hash Por Colisão

As Tabelas Hash, também conhecidas como tabelas de dispersão, são uma das estruturas de dados mais eficientes para implementar dicionários, oferecendo, em média, acesso quase constante (O(1)) para operações de inserção, busca e remoção. Sua principal característica é a utilização de uma função de hashing, que transforma uma chave de entrada em um índice numérico, mapeando-a diretamente para uma posição em um vetor (ou array).

No entanto, o principal desafio das Tabelas Hash são as colisões: quando a função de hashing gera o mesmo índice para duas chaves diferentes. Para lidar com isso, existem diversas estratégias, e uma das mais comuns e eficazes é o endereçamento exterior, também conhecido como encadeamento separado ou encadeamento.

Nesta abordagem, em vez de armazenar diretamente o elemento na posição do vetor, cada posição da Tabela Hash (chamada de ”balde” ou bucket) aponta para uma estrutura de dados secundária, geralmente uma lista encadeada (ou, em casos mais avançados, uma árvore balanceada para lidar com muitas colisões). Quando uma colisão ocorre, o novo elemento é simplesmente adicionado ao final da lista encadeada naquele índice. Para

buscar um elemento, a função de hashing aponta para o balde correto, e então a lista encadeada associada é percorrida até que a chave seja encontrada ou o final da lista seja atingido. Essa estratégia combina a velocidade de acesso direto da Tabela Hash com a flexibilidade das listas encadeadas para gerenciar múltiplos elementos no mesmo índice.

### 5.2. Estrutura da Hash Por Colisão

```
    template <typename Key, typename Value>
struct Elemento {
    Key chave;
    Value valor;
    int contador;
    Elemento(const Key& k, const Value& v) : chave(k),
    valor(v), contador(1) {}
};
```

O Elemento é a unidade básica que armazena dados em sua Tabela Hash. Sendo genérico (template ¡typename Key, typename Value¿), ele pode guardar qualquer tipo de chave e seu valor correspondente. Ele também inclui um contador, que rastreia quantas vezes a mesma chave foi ”inserida”, transformando a tabela em um multiconjunto. O construtor inicializa a chave e o valor, e define o contador como 1 para a primeira ocorrência.

### 5.3. Inserção na Hash Por Colisão

```
    void add(const KeyType& key, const ValueType& value)
```

Esta é a função pública que você chama para adicionar um novo par (key, value) à sua tabela hash. Ela é responsável por calcular o slot, verificar se a chave já existe, adicionar o novo elemento e gerenciar o fator de carga.

```
    size_t hash_code(const KeyType& key) const
```

Esta função pega uma chave e a transforma em um índice de bucket. Ela usa um std::hash padrão (que é genérico para muitos tipos básicos) e aplica o operador módulo pelo mtablesize. A qualidade da função hash é crucial para distribuir uniformemente as chaves pelos buckets e minimizar colisões

```
    float load_factor() const
```

Calcula o fator de carga atual da tabela, que é a razão entre o número de elementos (mnumberofelements) e o tamanho da tabela (mtablesize). Este valor é usado para decidir quando o rehash é necessário.

```
    size_t get_next_prime(size_t x)
```

Uma função utilitária para encontrar o próximo número primo maior ou igual a x. Usar tamanhos de tabela primos geralmente resulta em uma melhor distribuição dos elementos e menos colisões, melhorando o desempenho geral.

### 5.4. Busca na Hash Por Colisão

```
bool contains(const KeyType& key) const
```

Esta é a função mais simples para saber se um elemento com uma determinada chave está na sua tabela hash.

```
ValueType count(const KeyType& key) const
```

Embora o nome count possa sugerir uma contagem de ocorrências, na sua implementação, essa função é usada para recuperar o value associado a uma key específica. Isso a torna similar a um método get em um mapa ou dicionário.

### 5.5. Remoção na Hash Por Colisão

```
void remove(const KeyType& key)
```

Esta é a função pública que o usuário chama para remover uma chave (k) específica da tabela hash. Cálculo do Slot: Assim como nas operações de inserção e busca, o primeiro passo é determinar o índice do balde (slot = hashcode(k)) onde a chave k deveria estar. Busca e Remoção na Lista Encadeada: A função, então, itera através da std::list naquele slot usando um iterador (auto it). Ela procura por um Elemento cuja chave seja igual a k. Se o Elemento com a chave k for encontrado na lista, ele é removido fisicamente da std::list usando mtable[slot].erase(it). Após a remoção bem-sucedida, o mnumberofelements é decrementado para refletir a nova contagem de elementos na tabela. A função retorna true para indicar que a remoção foi realizada com sucesso. Chave Não Encontrada: Se a chave k não for encontrada na lista após percorrê-la por completo, a função retorna false, indicando que o elemento não estava presente na tabela para ser removido. Tempo de Execução: Em média, a complexidade é O(1). O acesso ao balde é O(1), e a busca e remoção em uma std::list também são O(L) (onde L é o comprimento da lista) no pior caso. Com um bom fator de carga e uma boa função de hashing, L tende a ser pequeno, mantendo a operação próxima de O(1) na maioria das vezes. No pior caso (todos os elementos colidem no mesmo balde), pode se degradar para O(N), onde N é o número total de elementos.

```
size_t hash_code(const Key& k) const
```

Esta função privada é usada para calcular o índice hash da chave que se deseja remover. É o ponto de partida para saber em qual balde (std::list) a busca e remoção devem ocorrer.

### 5.6. Comparações: Principal e rerashs

No contexto de uma Tabela Hash com Encadeamento as variáveis comparacoesprincipal e contadorrehash, juntamente com suas respectivas funções get, são cruciais para analisar e otimizar o desempenho da estrutura de dados. Elas nos fornecem métricas quantitativas sobre como a tabela está se comportando sob diferentes cenários de uso.

```
long long comparacoes_principal
```

A variável comparacoesprincipal é um contador que registra o número de comparações diretas de chaves que a tabela hash precisa realizar durante operações como add (adição), contains (verificação de existência) e search (busca).

`long long contador_rehash`

A variável contadorrehash é um contador que rastreia quantas vezes a tabela hash foi redimensionada e reconstruída (rehash) desde sua criação ou desde o último reset do contador.

### 5.7. Considerações finais

As Tabelas Hash com encadeamento exterior (ChainedHashTable) são uma das estruturas de dados mais eficientes para a implementação de dicionários (também conhecidos como mapas ou conjuntos), oferecendo um desempenho médio de tempo constante (O(1)) para as operações de inserção, busca e remoção. Sua popularidade deriva da forma inteligente como resolvem o problema das colisões.

A grande vantagem do encadeamento exterior é sua simplicidade e robustez no tratamento de colisões: quando duas chaves diferentes ”caem” no mesmo balde (índice hash), elas são simplesmente adicionadas a uma lista encadeada (std::list em sua implementação) naquele balde. Isso evita a complexidade de ter que ”sondar” por outro local na tabela principal, como ocorre no endereçamento aberto. Pontos chave da sua implementação: Tipos Genéricos (template ¡typename Key, typename Value¿): A capacidade de armazenar qualquer tipo de chave e valor torna sua ChainedHashTable extremamente versátil. Contador de Ocorrências: A inclusão do campo contador no Elemento transforma sua tabela hash em um multiconjunto, permitindo registrar quantas vezes uma mesma chave foi inserida, o que é um recurso valioso para certas aplicações. Redimensionamento Automático (rehash): A tabela se autoajusta. Quando o fator de carga (mnumberofelements / mtablesize) ultrapassa o limite (mmaxloadfactor), um rehash é acionado para dobrar o tamanho da tabela (para o próximo número primo) e redistribuir os elementos. Essa operação, embora O(N), é crucial para manter as listas dos baldes curtas e garantir a eficiência O(1) na maioria das operações. Função de Hashing Personalizável: A opção de fornecer uma std::hash¡Key¿ personalizada (typename Hash = std::hash¡Key¿) oferece flexibilidade e permite otimizar a distribuição das chaves para tipos de dados complexos, um fator crítico para o bom desempenho. Em resumo, sua ChainedHashTable é uma implementação sólida de uma tabela hash. Ela equilibra a velocidade de acesso direto com a robustez do encadeamento para gerenciar colisões, tornando-a uma excelente escolha para aplicações que exigem eficiência e escalabilidade na manipulação de grandes volumes de dados.

## 6. Tabelas Hash com Endereçamento Aberto

### 6.1. Introdução a Hash de Endereçamento Aberto

As Tabelas Hash, com sua promessa de acesso rápido (O(1) em média), são excelentes para implementar dicionários. Elas usam uma função de hashing para converter uma chave em um índice, apontando para um local direto em um vetor. No entanto, quando duas chaves diferentes geram o mesmo índice – uma colisão – precisamos de uma estratégia para resolvê-la.

O endereçamento aberto (também conhecido como sondagem ou chaveamento aberto) é uma das abordagens para lidar com colisões. Diferente do endereçamento exterior (que usa listas encadeadas), no endereçamento aberto, todos os elementos são armazenados diretamente no próprio vetor da Tabela Hash.

Quando ocorre uma colisão, em vez de criar uma lista, a tabela busca uma nova posição vazia no próprio vetor. Existem diferentes métodos para essa busca, chamados de sondagem:

Sondagem Linear: Procura sequencialmente as próximas posições (i+1,i+2,...) até encontrar um espaço vazio.

Sondagem Quadrática: Aumenta o passo de busca quadraticamente (i+12,i+22,...).

Hash Duplo: Usa uma segunda função de hashing para determinar o tamanho dos passos.

A busca, inserção e remoção no endereçamento aberto envolvem essa ”sondagem” para encontrar o item ou um local adequado. A eficiência dessa técnica depende muito da função de hashing e da estratégia de sondagem, que devem minimizar o número de ”passos” necessários para resolver colisões e evitar o ”agrupamento” de elementos na tabela.

### 6.2. Slot e sua Estrutura Na hash De Endereçamento Aberto

```
struct Slot {
    std::optional<Key> chave;
    std::optional<Value> valor;
    int contador = 0; // contador de inserções
    Estado estado = Estado::VAZIO;
    Slot() = default;
};

std::vector<Slot> m_table;
size_t m_table_size;
size_t m_number_of_elements;
float m_max_load_factor;
Hash m_hashing;
```

A Tabela Hash com endereçamento aberto é construída em torno de um vetor de Slots (mtable). Cada Slot é a unidade fundamental, capaz de armazenar uma chave (key) e um valor (value), ambos encapsulados em std::optional para indicar sua presença. Um contador (contador) adicional em cada slot permite gerenciar ocorrências de chaves, e um estado (estado) (Vazio, Ocupado, Removido) é crucial para o comportamento da sondagem. A Tabela Hash geral é gerenciada por seu tamanho (mtablesize), o número de elementos (mnumberofelements), um fator de carga máximo (mmaxloadfactor) que dispara o redimensionamento para manter a eficiência, e um objeto para a função de hashing (mhashing), que mapeia chaves para posições no vetor. Essa organização permite o armazenamento direto dos elementos no vetor e o tratamento de colisões pela busca por posições alternativas.

### 6.3. Inserção Na hash De Endereçamento Aberto

A inserção de um elemento em uma Tabela Hash que utiliza endereçamento aberto é um processo direto, mas que exige uma atenção especial à resolução de colisões. Ao contrário

do encadeamento separado, onde cada ”balde” da tabela pode conter uma lista de itens, no endereçamento aberto todos os elementos são armazenados diretamente no próprio vetor da tabela.

Quando uma nova chave e seu valor correspondente precisam ser inseridos, o primeiro passo é calcular a posição ideal para eles usando a função de hashing. Se essa posição estiver vazia, o elemento é inserido ali. No entanto, se a posição já estiver ocupada (ocorreu uma colisão), a tabela entra em modo de sondagem: ela busca sequencialmente uma próxima posição disponível dentro do próprio vetor, seguindo uma estratégia predefinida (como sondagem linear, quadrática ou hash duplo). O elemento é então inserido na primeira posição vazia encontrada. Além disso, a tabela monitora seu fator de carga; se ele exceder um limite, um redimensionamento (rehash) é acionado para manter a eficiência, realocando todos os elementos para um vetor maior. Esse método garante que a inserção seja, em média, muito rápida (O(1)), mas sua performance no pior caso pode depender da qualidade da função de hashing e da estratégia de sondagem.

```
bool insert(const Key& k, const Value& v)
```

Esta é a função pública que o usuário final utiliza para inserir um novo par chave-valor ou atualizar um valor existente associado a uma chave. Se a chave (k) já estiver presente na tabela, o valor (v) correspondente é atualizado, e um contador interno de ocorrências dessa chave é incrementado. Caso a chave seja nova, ela é inserida junto com seu valor e o contador é inicializado em 1. Um aspecto crucial dessa função é a capacidade de realizar um rehash automático se o fator de carga da tabela ultrapassar um limite predefinido, garantindo que a eficiência seja mantida mesmo com o aumento do número de elementos.

```
size_t hash_code(const Key& k) const
```

Esta função auxiliar é responsável por calcular o índice inicial onde uma chave (k) deve ser tentada no vetor da tabela hash. Ela pega a chave e, por meio de uma função de hashing, a transforma em um valor numérico que corresponde a uma posição dentro do mtablesize. Essa é a base para o acesso rápido na tabela.

```
int aux_hash_search(const Key& k) const
```

Essa função auxiliar é utilizada para localizar o índice de uma chave específica dentro da tabela. Em uma tabela com endereçamento aberto, isso pode envolver uma sondagem (procurar posições alternativas) para resolver colisões. Ela é fundamental para que add possa verificar se a chave já existe antes de tentar uma inserção ou atualização, e tem uma complexidade de tempo média de O(1), podendo chegar a O(n) no pior caso (todas as chaves em uma longa cadeia de sondagem).

```
void rehash(size_t new_size)
```

Quando a Tabela Hash fica muito ”cheia” (ou seja, seu fator de carga excede o mmaxloadfactor), a função rehash é chamada para redimensionar a tabela. Ela cria um novo vetor de slots com um newsize maior e, em seguida, reinsere todos os elementos válidos do vetor antigo para as novas posições no vetor maior, utilizando a função de hashing e as regras de sondagem novamente. Este processo é vital para manter o desempenho de O(1) das operações e minimizar a chance de colisões excessivas, embora seja uma operação que pode ter uma complexidade de tempo de O(n) quando ocorre.

### 6.4. Busca Na hash De Endereçamento Aberto

A busca por um elemento em uma Tabela Hash que utiliza endereçamento aberto é uma operação que, idealmente, é extremamente rápida. Ela começa da mesma forma que a inserção: a função de hashing é aplicada à chave desejada para calcular sua posição inicial no vetor da tabela.

Se o elemento estiver exatamente nessa posição e a chave corresponder, a busca é instantânea. Contudo, devido às colisões (quando diferentes chaves mapeiam para o mesmo índice), a busca pode não ser tão direta. Nesses casos, o algoritmo entra em modo de sondagem, percorrendo outras posições no vetor da tabela, seguindo a mesma estratégia usada durante a inserção (linear, quadrática, ou hash duplo), até que a chave procurada seja encontrada ou um slot vazio seja atingido. O slot vazio indica que a chave não está na tabela. Embora a busca mantenha uma complexidade de tempo média de O(1), a eficiência no pior caso pode degradar-se para O(n) se houver muitos elementos agrupados ou se a tabela estiver muito cheia, exigindo muitos ”saltos” para encontrar ou confirmar a ausência de um item.

```
int aux_hash_search(const Key& k) const
```

Esta é uma função privada que realiza a busca interna (sondagem) por uma key na tabela. Ela começa na posição indicada pela função de hashing e, se encontrar colisões, continua procurando posições alternativas (sondando) até localizar a key ou um slot vazio que indique que a chave não está presente. Se a chave for encontrada, ela retorna o índice da posição; caso contrário, retorna -1. Esta função é a base para contains e at.

```
Value& at(const Key& k)
```

Esta função pública permite acessar diretamente o valor associado a uma key. Ela usa auxhashsearch para encontrar a key. Se a chave for localizada, retorna uma referência modificável (Value) para o valor, permitindo sua alteração. Se a chave não for encontrada após a sondagem, esta função lança uma exceção, indicando que o elemento não existe na tabela.

```
const Value& at(const Key& k) const
```

Esta é a versão const da função at. Ela também retorna uma referência, mas não modificável (const Value), para o valor associado a uma key. É usada quando você quer apenas ler o valor sem a intenção de modificá-lo, garantindo a segurança de tipos em contextos onde a tabela não deve ser alterada. Assim como a versão não const, ela lança uma exceção se a chave não for encontrada.

### 6.5. Remoção Na hash De Endereçamento Aberto

A remoção de um item em uma Tabela Hash que usa endereçamento aberto não é tão simples quanto apenas apagar o dado. Para não ”quebrar” as buscas futuras (já que elas podem ter que ”sondar” por posições), o slot do item removido geralmente não é esvaziado. Em vez disso, ele é marcado logicamente como REMOVIDO. Isso permite que as operações de busca e inserção saibam que podem ”passar” por aquele slot (busca) ou que ele pode ser usado para uma nova inserção (inserção), sem interromper a cadeia de sondagem. Essa abordagem garante que a remoção mantenha a eficiência média de O(1), embora o gerenciamento de slots REMOVIDO exija atenção para evitar o declínio de desempenho com o tempo e um rehash pode ser necessário.

```
bool remove(const Key& k)
```

Essa é a função pública que o usuário do seu dicionário chamará para remover uma chave (k) da tabela. Ela não remove fisicamente o elemento do slot. Em vez disso, após encontrar a chave (geralmente usando uma lógica similar à de auxhashsearch), o Slot correspondente à chave é marcado com um estado REMOVIDO. Isso permite que futuras operações de busca continuem ”passando” por esse slot (como se ele ainda estivesse ocupado) até encontrar a chave desejada ou um slot verdadeiramente vazio. O contador de elementos (mnumberofelements) é decrementado. A função retorna true se a chave foi encontrada e marcada para remoção, e false se a chave não foi encontrada na tabela. Essa remoção lógica é crucial para não quebrar as cadeias de sondagem.

```
size_t hash_code(const Key& k) const
```

Esta função auxiliar tem o papel de calcular o índice inicial no vetor da tabela onde a busca pela key a ser removida deve começar. É o primeiro passo para localizar o slot correto antes de aplicar a remoção.

```
int aux_hash_search(const Key& k) const
```

Embora listada aqui como auxiliar para remoção, esta função, na sua essência, realiza a busca por uma chave específica na tabela usando a estratégia de sondagem. Ela é vital para localizar o índice da key que se deseja remover. Uma vez que o auxhashsearch retorna o índice da chave (ou -1 se não encontrar), a função remove pode proceder com a marcação do slot.

### 6.6. Comparações: Principal e rerashs

No contexto de uma Tabela Hash de Endereçamento Aberto (como a HashAberto que você forneceu), as variáveis comparacoesprincipal e contadorrehash, juntamente com suas respectivas funções get, são cruciais para analisar e otimizar o desempenho da estrutura de dados. Elas nos fornecem métricas quantitativas sobre como a tabela está se comportando sob diferentes cenários de uso.

```
long long comparacoes_principal
```

A variável comparacoesprincipal é um contador que registra o número de comparações diretas de chaves que a tabela hash precisa realizar durante operações como add (adição), contains (verificação de existência) e remove (remoção).

```
long long contador_rehash
```

A variável contadorrehash é um contador que rastreia quantas vezes a tabela hash foi redimensionada e reconstruída (rehash) desde sua criação ou desde o último reset do contador.

### 6.7. Considerações finais

Ao final da discussão sobre as Tabelas Hash com endereçamento aberto, fica claro que esta é uma estrutura de dados de alto desempenho, especialmente valorizada por sua capacidade de oferecer acesso quase constante (O(1) em média) para as operações de inserção, busca e remoção. Sua força reside na habilidade de mapear chaves diretamente para posições de memória através de uma função de hashing.

O principal desafio e a característica definidora do endereçamento aberto é o tratamento de colisões. A estratégia de sondagem (linear, quadrática ou hash duplo) permite que todos os elementos residam no próprio vetor da tabela, o que pode levar a um melhor uso da cache da CPU em comparação com o encadeamento separado. No entanto, a remoção exige um tratamento cuidadoso com a marcação lógica de slots (REMOVIDO) para não comprometer futuras buscas.

A eficiência de uma Tabela Hash com endereçamento aberto é fortemente influenciada pela qualidade da função de hashing e pelo fator de carga. Uma boa função minimiza colisões, e um gerenciamento adequado do fator de carga (com redimensionamento ou rehash quando necessário) é crucial para evitar a degradação de desempenho, que no pior caso pode regredir para O(n).

Em suma, quando bem implementada, uma Tabela Hash com endereçamento aberto é uma solução extremamente rápida e robusta para dicionários, ideal para cenários onde a velocidade de acesso aos dados é a prioridade máxima.

## 7. Notas e Considerações para a proxima parte do trabalho

A segunda etapa do projeto focará na aplicação prática e na eficiência das estruturas de dados. irei implementar a leitura de arquivos, o que exige tratamento robusto de strings (normalização, remoção de caracteres especiais, tokenização) para garantir dados precisos. Em seguida, farei testes mais robustos com grandes volumes de dados. Isso permitirá avaliar o desempenho, a escalabilidade e identificar gargalos, comprovando a eficiência do programa em condições reais. Por fim, os resultados dos testes guiarão ajustes em funções e na estrutura de acesso aos arquivos, como otimizações em funções hash ou balanceamento da árvore, para garantir a melhor performance.

## 8. Resultados

Este relatório apresenta uma análise comparativa entre diferentes estruturas de dados utilizadas na construção de um dicionário para contagem de frequência de palavras em arquivos de texto. As estruturas implementadas foram:

- Árvore AVL;
- Árvore Rubro-Negra;
- Tabela Hash com Encadeamento (Chained Hashing);
- Tabela Hash com Endereçamento Aberto (Open Addressing).

Para a realização dos testes com as diferentes estruturas de dados, foi utilizado como entrada um arquivo de texto contendo o conteúdo completo da Bíblia na versão King James (KJV). O arquivo foi obtido gratuitamente no seguinte endereço: `https://openbible.com/textfiles/kjv.txt`. Esse arquivo foi escolhido por possuir um grande volume de texto, com diversidade de palavras e pontuações, o que permite avaliar de forma mais realista o desempenho das estruturas em situações de uso intensivo.

As comparações foram feitas com base em três métricas principais:

1. Tempo de montagem (inserção de todas as palavras);
2. Número de comparações de chaves realizadas;
3. Número de rotações (para árvores) ou colisões (para hash).

**Table 1. Resumo dos resultados de desempenho por estrutura**

| Estrutura | Tempo (s) | Comparações | Rotações/Rerashs |
|---|---|---|---|
| Árvore AVL | 2.618 | 27.087.165 | 9.641 (rotações) |
| Rubro-Negra | 1.684 | 15.428.870 | 8.113 (rotações) |
| Hash Encadeado | 0.865 | 1.038.360 | 10 (Rerashs) |
| Hash Aberto | 1.448 | 5.692.128 | 10 (Rerashs) |

### 8.1. Análise dos Resultados

A Tabela 1 mostra claramente que as estruturas baseadas em **tabelas hash** foram significativamente mais rápidas do que as baseadas em árvores. O tempo de montagem da estrutura com **hash encadeado** foi o menor (menos de 1 segundo), seguido da **hash com endereçamento aberto**. Ambas apresentaram um número muito reduzido de colisões, evidenciando a boa dispersão da função hash utilizada. Esse desempenho superior está de acordo com a complexidade média dessas estruturas, que é $\mathcal{O}(1)$ para inserções e buscas, desde que a função hash seja eficiente e o fator de carga controlado.

Por outro lado, as **estruturas de árvore**, embora mais lentas, oferecem vantagens importantes: garantem a ordenação dos elementos e possibilitam operações eficientes sobre intervalos de chaves, como buscas por prefixo ou faixas alfabéticas. Ambas as árvores balanceadas — **AVL** e **Rubro-Negra** — mantêm a altura da árvore limitada, garantindo complexidade $\mathcal{O}(\log n)$ para inserções e buscas. A **AVL** realizou mais comparações e rotações do que a **Rubro-Negra**, o que condiz com sua política de balanceamento mais rigorosa, que visa manter a árvore o mais equilibrada possível.

## 9. Conclusão

A escolha da estrutura ideal depende do uso pretendido:

- Para aplicações com foco em desempenho de inserção e busca simples, as estruturas **hash** são preferíveis.
- Para aplicações que exigem **ordenação ou buscas por intervalo**, as **árvores** (especialmente a Rubro-Negra) são mais adequadas.

## 10. Referencias


- **Título:** *Algoritmos: Teoria e Prática*
- **Autores:** Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein
- **Editora:** Campus / Elsevier
- **Observação:** Considerada a **bíblia dos algoritmos e estruturas de dados**. Aborda com profundidade tópicos como Árvores Rubro-Negras, sendo uma referência essencial para o entendimento de estruturas complexas.

- **Título:** *Estruturas de Dados e Seus Algoritmos*
- **Autores:** Jayme Luiz Szwarcfiter e Lishon S. Markenzon
- **Editora:** LTC
- **Observação:** Um **clássico da literatura brasileira**, oferece uma abordagem clara e didática com foco em exemplos e implementação em C/C++.

- **Título:** *Estruturas de Dados com C++*
- **Autor:** Adam Drozdek
- **Editora:** Cengage Learning
- **Observação:** Conhecido pela sua **clareza e foco em implementações práticas** de estruturas de dados utilizando C++.

- **Título:** *Data Structures and Algorithm Analysis in C++*
- **Autor:** Mark Allen Weiss **Editora:** Pearson
- **Observação:** Um livro respeitado pela sua **análise aprofundada da complexidade** de tempo e espaço das estruturas e algoritmos, com exemplos em C++.