UFMG
UNIVERSIDADE FEDERAL DE MINAS GERAIS

**Universidade Federal de Minas Gerais**

**Turma:** Ciência da Computação **Prof.:** Gisele e Wagner

**Nome:** Rubia Alice Moreira de Souza

**Matrícula:** 2022043507


# TP3 - Dicionários

Belo Horizonte

2022

# Sumário

# 1. Introdução

Este trabalho consiste em implementar e avaliar dois tipos de dicionários. Um deles é baseado na estrutura hash e foi utilizado com a carga de trabalho típica de no máximo 75% da tabela preenchida. Já o outro utiliza a estrutura de dados de uma árvore balanceada AVL.

O programa recebe um conjunto de verbetes a partir de um arquivo texto. Cada verbete é identificado por dois atributos: A palavra e o tipo associado a ele. Para verbetes de um mesmo tipo com mesma palavra, os significados deles são agrupados em uma única entrada.

O objetivo deste trabalho é estudar as diferentes implementações de dicionário usando as duas estruturas de dados e compará-las a fim de verificar quando é mais vantajoso usar cada uma.

# 2. Método

## 2.1. Classes

Inicialmente, temos um namespace Utils, que contém funções utilitárias para manipular as strings que serão lidas no arquivo de entrada:

| **Utils** |
|---|
| toLower(referencia : string) : string |
| extrairColchetes(referencia : string) : string |
| formatarPalavra(referencia : string) : string |
| trimEnd(referencia : string&) : void |

Foi criada uma classe abstrata do tipo Dicionario, a fim de padronizar a assinatura dos métodos dos dois tipos de implementação que ele poderá ter:

| **Dicionario** | |
|---|---|
| **Atributos** | **Métodos** |

| | |
|---|---|
| | inserir(verbete : Verbete) : void<br><br>pesquisar(palavra : string, tipo : TiposVerbete) : Verbete*<br><br>atualizar(verbete : Verbete) : void<br><br>remover(palavra : string, tipo : TiposVerbete) : void<br><br>imprimir() : void<br>imprimirEmArquivo(arquivo : ofstream) : void<br><br>limpar() : void<br><br>estaVazia() : bool<br><br>removerVerbetesComMaisDeUmSignificado() : void |

Para o dicionário baseado na árvore AVL, foi necessário criar uma classe nodo que representa uma entrada na árvore. Cada nodo sabe a sua altura a fim de evitar o cálculo recursivo da altura sempre que for necessário verificar o fator de balanceamento ou atualizar a altura de um caminho na árvore.

| **Nodo** | |
|---|---|
| **Atributos** | **Métodos** |
| esquerdo : Nodo*<br>direito : Nodo*<br><br>conteudo : Verbete*<br>altura : int | setNodoEsquerdo(esquerdo : Nodo*) : void<br>getNodoEsquerdo() : Nodo*&<br><br>setNodoDireito(direito : Nodo*) : void<br>getNodoDireito() : Nodo*&<br><br>setConteudo(conteudo : Verbete*) : void<br>getConteudo() : Verbete* |

| | |
|---|---|
| | getAltura() : int |

Com relação ao dicionário com árvore AVL, ele realiza a implementação de todos os métodos da classe abstrata pai a partir de métodos auxiliares recursivos. Além disso, os métodos atualizarAlturaNodo(), calcularFatorBalanceamento(), rotacionarParaEsquerda(), rotacionarParaDireita() são auxiliares para manter o balanceamento da árvore na inserção e remoção de elementos.

| **DicionarioArvoreAVL : Dicionario** | |
|---|---|
| **Atributos** | **Métodos** |
| raiz : Nodo*<br>quantidadeElementos : unsigned int | imprimirArvoreHorizontal() : void<br><br>inserirRecursivamente(nodo : Nodo*, verbete : Verbete&) : Nodo*<br><br>pesquisarRecursivamente(nodo : Nodo*&, palavra : string, tipo : TiposVerbete) : Verbete*<br><br>atualizarRecursivamente(nodo : Nodo*&, verbete : Verbete&) : void<br><br>removerRecursivamente(nodo : Nodo*, palavra : string, tipo : TiposVerbete) : Nodo*<br><br>encontrarMenorValorSubArvoreDireita(subarvore : Nodo*) : Nodo*<br><br>void imprimirRecursivamente(nodo : Nodo*&)<br>imprimirEmArquivoRecursivo(Nodo*& nodo, std::ofstream* arquivo) : void<br>imprimirArvoreHorizontal(nodo : Nodo*&, espacamento : unsigned int) : void<br><br>limparRecursivamente(nodo : Nodo*&) : void<br><br>buscarVerbetesComMaisDeUmSignificadoRecursivo(verbetes : Verbete[], nodo : |

| | |
|---|---|
| | Nodo*, index : unsigned int&) : void<br><br>atualizarAlturaNodo(nodo : Nodo*&) : void<br>calcularFatorBalanceamento(nodo : Nodo*&) : int<br><br>rotacionarParaEsquerda(nodo : Nodo*) : Nodo*<br>rotacionarParaDireita(nodo : Nodo*) : Nodo* |

Já o dicionário hash apenas possui o método auxiliar hash(), que realiza o mapeamento de uma string para uma determinada lista de verbetes. O método considera o resto da divisão entre, a soma dos valores inteiros ASCII de cada caracter presente na string multiplicado pela sua posição, e o tamanho do array de listas de verbetes, para determinar o index da lista em que o verbete estará.

| **DicionarioHash : Dicionario** | |
|---|---|
| **Atributos** | **Métodos** |
| QUANTIDADE_LISTAS : unsigned int<br><br>listasVerbetes[QUANTIDADE_LISTAS] : Lista<Verbete*><br><br>tamanho : unsigned int | hash(palavra : string) : unsigned int |

Como estruturas auxiliares para o dicionário hash, foi necessário utilizar uma lista duplamente encadeada a fim de realizar o endereçamento no hash. Portanto, quando há algum conflito de chaves, os elementos são inseridos no final da lista. Ambas estruturas auxiliares, Celula e Lista, são especificadas abaixo:

| **Celula<T>** | |
|---|---|
| **Atributos** | **Métodos** |
| proxima : Celula*<br>anterior : Celula* | setValor(valor : T) : void<br>getValor() : T |

| | |
|---|---|
| valor : T | setProximaCelula(Celula<T>* proxima) : void<br>getProximaCelula() : Celula<T>*<br><br>setAnteriorCelula(Celula<T>* anterior) : void<br>getAnteriorCelula() : Celula<T>* |

| **Lista<T>** | |
|---|---|
| **Atributos** | **Métodos** |
| inicio : Celula<T>*<br>fim : Celula<T>*<br><br>tamanho : unsigned int | adicionarFim(valor : T*) : void<br>adicionarComeco(valor : T*) : void<br>adicionarPosicao(valor : T*, posicao unsigned int) : void<br><br>get(posicao : unsigned int) : T*<br><br>removerItem(item : T) : void<br>removerFim() : void<br>removerComeco() : void<br>remover(posicao : unsigned int) : void<br>remover(iterador : Celula<T>*) : void<br><br>limpar() : void |

Além disso, foi necessário utilizar um algoritmo de ordenação para ajustar a saída final que é impressa pelo dicionário hash. No caso, foi escolhido o algoritmo QuickSort, implementado em uma classe estática, para realizar essa função.

| **QuickSort** | |
|---|---|
| **Atributos** | **Métodos** |
| | ordenarCrescente(vetor : Verbete*[], tamanho : unsigned int) : void<br>ordenarDecrescente(vetor : Verbete*[], const tamanho : unsigned int) : void |

| | |
|---|---|
| | quickSortCrescente(vetor : Verbete*[], posicaoInicial : int, poiscaoFinal : int) : void<br>particionarCrescente(vetor Verbete*[], posicaoInicial : int, poiscaoFinal : int, indexEsquerda : int*, indexDireita : int*) : void<br><br>quickSortDecrescente(vetor Verbete*[], posicaoInicial : int, poiscaoFinal : int) : void<br>void particionarDecrescente(vetor Verbete*[], posicaoInicial : int, poiscaoFinal : int, indexEsquerda : int*, indexDireita : int*) : void |

A fim de melhorar a legibilidade do código, todos os tipos de verbetes foram mapeados para a enumeração abaixo:

| **TiposVerbete** | |
|---|---|
| **Atributos** | **Métodos** |
| ADJETIVO = 'a'<br>NOME = 'n'<br>VERBO = 'v' | converterCharParaTiposVerbete(tipo : char) : TiposVerbete |

Por fim, temos a classe DTO verbete. Ela possui como identificador a sua palavra e o tipo. Além disso, para cada verbete de um mesmo tipo e palavra, o seu significado é inserido na lista de significados.

| **Verbete** | |
|---|---|
| **Atributos** | **Métodos** |
| palavra : string<br>tipoVerbete : TiposVerbete<br>significados : Lista<std::string>* | imprimir() : void<br>imprimirEmArquivo(arquivo : ofstream*) : void<br><br>setPalavra(palavra : string) : void<br>getPalavra() : string<br><br>setTipo(tipo : TiposVerbete) : void |

| | |
|---|---|
| | setTipo(tipo : char) : void<br>getTipo() : TiposVerbete<br><br>adicionarSignificado(significado : string) : void<br>adicionarSignificado(significados : Lista<string>&) : void<br>getSignificados() : Lista<string>*<br><br>getQuantidadeSignificados() unsigned int |

## 2.2. Configurações de Ambiente

O código foi executado nas seguintes configurações de ambiente:

- **Sistemas operacionais:** Windows, Linux e na WSL do Windows utilizando a versão Ubuntu 20.04 LTS
- **Linguagem utilizada:** C++11
- **Compilador utilizado:** g++ da GNU Compiler Collection
- **Processador utilizado:** Intel(R) Core(™) i5-7200U CPU @ 2.50GHz, 2701 Mhz, 2 Núcleo(s), 4 Processador(es) Lógico(s)
- **Quantidade de memória RAM disponível:** 8Gb

# 3. Análise de Complexidade

## 3.1. Dicionário Árvore AVL

### 3.1.1. Inserir

O comportamento para inserção em uma árvore AVL é semelhante à inserção em uma árvore binária. Como a árvore AVL mantém seu balanceamento a cada inserção e remoção, para encontrar o nodo em que será inserido o novo conteúdo, haverá um custo de O(log(n)). Além disso, após a inserção pode ser necessário realizar rotações. Como as rotações são apenas manipulações de ponteiros, o seu custo é constante, ou seja, O(1). Por fim, pode ser necessário atualizar a altura dos nodos a partir do nodo inserido. No pior caso, esse custo será de O(log(n)), pois seria necessário percorrer desde o nó folha até a raiz atualizando as alturas. Assim, o custo final seria de O(log(n)) + O(log(n)) + O(1) = O(log(n)).

### 3.1.2. Pesquisar

Para encontrar um elemento na árvore AVL, o custo será O(log(n)) no pior caso, pois, como a árvore sempre se mantém balanceada, sempre há uma distribuição igualitária entre os

caminhos esquerdo e direito a ser seguidos pela árvore, evitando assim alturas desiguais que levariam a custos maiores que log(n).

Observação: No caso do projeto, em que é possível haver dois verbetes com mesma palavra, mas tipos diferentes, é necessário verificar tanto o caminho a esquerda quanto o caminho a direita, caso seja buscado um verbete com essa característica. Isso pois, ao remover ou inserir na árvore, pode ser que alguma rotação desloque o posicionamento desse verbete com tipo diferente e palavra igual, que é usada como chave. Assim, no pior caso, haveria dois custos de log(n) (ou seja 2log(n)), um para percorrer a subárvore à esquerda e outro para a sub-árvore à direita. No entanto, o custo assintótico final ainda seria O(log(n)).

#### 3.1.3. Atualizar

Novamente, o custo para atualizar um elemento na árvore AVL é O(log(n)), pois, assim como na pesquisa, devemos encontrar o elemento que deseja ser atualizado e para isso realizamos uma busca ao longo da árvore balanceada. Ao encontrar um elemento que tenha a mesma palavra e tipo, todas as operações são constantes, pois apenas ocorrem atribuições, que sobre escrevem os valores antigos.

#### 3.1.4. Remover

Para remover um elemento da árvore, inicialmente, é necessário fazer uma busca para encontrá-lo na árvore. Caso seja um verbete com mais de um tipo, temos duas vezes o custo de log(n) para procurar, tanto na subárvore à direita quanto à esquerda. Além disso, caso também seja um nó com mais de um filho, devemos encontrar o menor elemento na subárvore à direita para realizar a troca com o elemento atual que será removido, gerando um custo de no máximo log(n). Depois, realizamos manipulações de ponteiro de custo constante, para realizar a remoção, e a possível rotação, para ajustar o balanceamento da árvore, devemos atualizar a altura de cada nodo a partir do removido até a raiz, gerando um custo de log(n). Portanto, no final temos O(2log(n)) + O(log(n)) + O(1) + O(log(n)) = O(log(n)).

Observação: O cálculo do fator de balanceamento de cada nodo é constante devido ao fato de cada nodo já saber a sua altura previamente.

#### 3.1.5. ImprimirEmArquivo e Imprimir

O custo para imprimir os elementos da árvore AVL é O(n), pois é necessário percorrer todos os elementos presentes na árvore.

#### 3.1.6. Limpar

Assim como a impressão, limpar a árvore exige que voĉe passe por todos elementos pelo menos uma vez. Portanto, o custo também será de O(n).

#### 3.1.7. estaVazia

O custo para verificar se a árvore está vazia é constante, pois basta verificar se a raiz é nula.

#### 3.1.8. removerVerbetesComMaisDeUmSignificado

Para remover todos os verbetes com mais de um significado, foi necessário criar um array auxiliar para armazená-los. Assim, há um custo de memória extra O(k), sendo “k” a quantidade de verbetes com mais de um significado e “k” é menor ou igual a “n”, que é a quantidade de elementos.

Após isso, é necessário percorrer todos elementos da árvore a fim de verificar se ele possui ou não mais de um significado e armazená-lo no vetor se necessário. Assim, como devemos percorrer todos elementos, temos um custo de O(n).

Então, para cada verbete colocado no array auxiliar, devemos realizar a remoção dele na árvore. Como uma remoção possui o custo de log(n) e iremos realizá-la para cada elemento no array auxiliar, teremos um custo de O(klog(n)).

Portanto, o custo computacional final será de: O(n) + O(klog(n)) = O(klog(n)), sendo “k” a quantidade de verbetes com mais de um significado e “n” a quantidade de elementos na árvore AVL.

### 3.2. Dicionário Hash

#### 3.2.1. Inserir

Para inserir um elemento no hash, o custo sempre será constante, O(1), pois a função de mapeamento hash tem custo constante e a inserção de um elemento ao final da lista também sempre tem custo constante.

#### 3.2.2. Pesquisar

O custo para procurar um elemento no hash depende diretamente do quão preenchido ele está. No melhor caso, em que não houve colisões e os elementos estão cumprindo o fator de carga de 75, o custo de acesso é constante, pois sempre será o primeiro elemento da lista alvo.

No entanto, para os casos em que houveram colisões, o custo passa a depender da quantidade de elementos presentes na lista. Ou seja, a quantidade de elementos que tiveram a mesma colisão. No pior caso, será necessário percorrer toda a lista para encontrar o elemento desejado. Assim, o custo assintótico seria o custo constante de encontrar o index hash mais o custo de percorrer a lista: O(1) + O(n) = O(n), sendo “n” a quantidade de elementos presentes na lista alvo.

### 3.2.3. Atualizar

Assim como na pesquisa, para atualizar um elemento, teremos um custo variável de acordo com o preenchimento da tabela hash. No melhor caso, cada lista possui apenas um elemento e o acesso para encontrar e atualizar um elemento será constante, sendo apenas necessário calcular o mapeamento do index.

Caso haja colisões, também devemos percorrer a lista até encontrar o elemento na lista para atualizá-lo. Como a atualização são apenas operações de atribuições, o seu custo é constante. Assim, o custo será: O(1) + O(n) + O(1) = O(n), sendo “n” a quantidade de elementos presentes na lista alvo.

### 3.2.4. Remover

Novamente, o custo irá depender do preenchimento da tabela. O melhor caso, o acesso será direto e terá custo constante O(1).

Já para o caso de colisões será necessário percorrer a lista com um iterador até encontrar o elemento que será removido, gerando como custo, no pior caso, de O(n), sendo “n” a quantidade de elementos presentes na lista alvo. O custo de remoção de um elemento na lista é constante, pois a partir do iterador é possível unificar a célula anterior e seguinte.

### 3.2.5. ImprimirEmArquivo e Imprimir

Para imprimir todos elementos da tabela hash, será necessário passar por cada elemento em cada lista. Assim, o custo será de O(n), sendo “n” a quantidade de elementos inseridos na tabela hash.

Observação: Neste trabalho, como é necessário ordenar a saída, tive que criar um vetor auxiliar para armazenar os elementos do hash e então ordená-lo, utilizando o algoritmo QuickSort. Assim, há um custo de memória extra de O(n) e também um custo computacional extra para ordenar os elementos pelo QuickSort, que no melhor caso é O(nlog(n)) e no pior caso será $O(n^2)$. Portanto, o custo assintótico para impressão dos dados no trabalho será de O(nlog(n)) no melhor caso e caso médio e $O(n^2)$ no pior caso.

### 3.2.6. Limpar

Para limpar a tabela hash, é necessário percorrer todas as listas chamando o método de limpar cada uma delas. Considerando um todo, o processo é como se passássemos por cada elemento da tabela hash. Assim, o custo também será O(n), sendo “n” a quantidade de elementos.

### 3.2.7. estaVazia

O custo para verificar se a tabela hash está vazia depende da quantidade de listas alocadas na estrutura. Para cada lista, devemos verificar se ela está vazia, que utiliza um método de custo constante. Assim, o custo é O(k), sendo “k” a quantidade de listas presentes no hash.

#### 3.2.8. removerVerbetesComMaisDeUmSignificado

Como conseguimos acessar mais diretamente e sequencialmente os elementos no hash do que na árvore, o custo para remover os verbetes com mais de um significado consiste em apenas percorrer todos elementos em cada lista utilizando um iterador, que tem custo de remoção constante. Assim, o custo dependerá de percorrer cada elemento, sendo então O(n).

### 3.3. Comparação Dicionário Hash e Árvore AVL

Após a implementação deste trabalho, foi possível notar que tanto a árvore quanto o hash possuem condições ideais para serem utilizados. A árvore é uma estrutura que parece ser mais indicada quando não conseguimos estimar a quantidade de dados que devemos armazenar. Já a tabela hash é ideal quando conseguimos ter essas estimativas para que possamos garantir um fator de carga ideal a fim de ter os custos de acesso constantes para a maioria dos seus métodos.

Além disso, a árvore é mais interessante caso haja a necessidade de definir algum tipo de ordenação para os elementos, pois a própria estrutura já é capaz de mantê-los dessa forma. Para o hash, isso não é diretamente viável, pois iria atrapalhar o mapeamento estabelecido pela função hash. Assim, é necessário estruturas auxiliares para conseguir colocar os elementos do hash em uma determinada ordem.

Por outro lado, o hash demonstra ser uma estrutura mais interessante para poder inserir, atualizar e deletar dados, quando seu fator de carga está abaixo de 75%, pois o custo de acesso é constante. Além disso, é mais fácil navegar entre todos os elementos da estrutura.

Para as situações apresentadas neste trabalho, o hash demonstrou ter mais facilidade na manipulação dos dados, como na remoção de todos os verbetes que possuem mais de um significado. Já a árvore apresentou uma impressão mais simples, por já conseguir colocar os elementos em ordem sem auxílio de alguma outra estrutura.

Portanto, é possível notar que uma estrutura hash é mais recomendada para quando conseguimos prever a quantidade de dados que serão armazenados, a fim de ajustar o fator de carga, e possui mais facilidade em manipular as informações. Um ponto negativo da estrutura é ter uma parte de memória não utilizada a fim de manter o fator de carga abaixo de um certo valor. Já a árvore é ideal quando não temos uma noção prévia da quantidade de dados e caso haja a necessidade de ordenação ou organização deles. Além disso, não há “desperdício de memória” ao utilizar essa estrutura.

# 4. Estratégias de Robustez

Para identificar e tratar possíveis erros do sistema são utilizadas exceções customizadas. Todas elas herdam da classe genérica Excecao.h, que herda de runtime_error.

As exceções são relacionadas a duas classes:

- **ExcecoesLista:** Realiza as verificações de exceções na busca e remoção de elementos com base em index inválidos: IndexInvalidoException.

- **ExcecoesMain:** Realiza as verificações de entradas do programa.
    - **ArquivoEntradaNaoFornecidoException:** É lançada caso o arquivo de entrada não seja fornecido.
    - **ArquivoEntradaNaoAbertoException:** É lançada caso o caminho ou nome do arquivo de entrada seja inválido.
    - **ArquivoSaidaNaoFornecidoException:** É lançada caso o arquivo de saída não seja informado.
    - **ArquivoSaidaNaoAbertoException:** É lançada caso o caminho ou nome do arquivo de saída seja inválido.
    - **ArgumentoInvalidoException:** É lançada ao não informar um tipo para o dicionário ou informar um tipo inválido.

# 5. Conclusões

Neste trabalho, realizamos a implementação de um dicionário a partir de duas estruturas de dados distintas, sendo elas, uma árvore AVL e um hash. Portanto, foi possível a partir disso compreender melhor o desenvolvimento e funcionamento dessas estruturas, além de explicitar as vantagens de cada uma.

# 6. Bibliografia

Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Favtutor. AVL Tree - Insertion, Deletion and Rotation with Python Code. Disponível em: <https://favtutor.com/blogs/avl-tree-python>. Acesso em: 12/12/2022.

Geeks for Geeks. Complexity of different operations in Binary tree, Binary Search Tree and AVL tree. Disponível em: <https://www.geeksforgeeks.org/complexity-different-operations-binary-tree-binary-search-tree-avl-tree/>. Acesso em: 12/12/2022.

Opengenus. Time & Space Complexity of AVL Tree operations. Disponível em: <https://iq.opengenus.org/time-complexity-of-avl-tree/>. Acesso em: 12/12/2022.

Stackoverflow. Hash table runtime complexity (insert, search and delete). Disponível em: <https://stackoverflow.com/questions/9214353/hash-table-runtime-complexity-insert-search-and-delete>. Acesso em: 12/12/2022.

Stackoverflow. Hash table is always O(n) time for lookup?. Disponível em: <https://stackoverflow.com/questions/40034552/hash-table-is-always-on-time-for-lookup>. Acesso em: 12/12/2022.

# Apêndice 1. Instruções para Compilação

Para executar o código do trabalho e disponibilizado no github, você deve realizar os seguintes passos em qualquer terminal de preferência:

1. Acesse a pasta raiz (UFMG-ED-Dicionario) do projeto, onde está presente o Makefile.
2. Execute o comando make.
3. Acesse a pasta “bin” do projeto.
4. Inicie o executável “run” considerando as seguintes flags.
    - **4.1. [Obrigatória]** -i “caminho/nome_arquivo_entrada.txt”
        - **4.1.1.** Informe o caminho e nome do arquivo de entrada que contém os verbetes que serão inseridos no dicionário em linhas diferentes no arquivo.
            - 4.1.1.1. Os verbetes devem estar no formato: tipo [palavra] significado, sendo o significado opcional
            - 4.1.1.2. Ex.: “a [abundant] plentiful”
    - **4.2. [Obrigatória]** -o “caminho/nome_saida.txt”
        - **4.2.1.** Informe o caminho e nome do arquivo resultante que irá os verbetes de mesmo tipo e palavra agrupados, além da lista de verbetes que não possuem nenhum significado.
    - **4.3. [Obrigatória]** -t
        - **4.3.1.** Esta flag define qual tipo de dicionário será utilizado. Os possíveis valores para ela são:
            - 4.3.1.1. “arv”: Executa o programa utilizando um dicionário com base em Árvore AVL (DicionarioArvoreAVL)
            - 4.3.1.2. “hash”: Executa o programa utilizando um dicionário com base em Tabela Hash (DicionarioHash)
5. Após informar as flags obrigatórias um arquivo .txt deve ser criado no local e com o nome informado no parâmetro -o.
6. O arquivo gerado deve apresentar, inicialmente, todos os verbetes com seus respectivos significados em ordem alfabética e posteriormente todos os verbetes que não possuem nenhum significado, também em ordem alfabética.

- Exemplo de comando para executar dicionário com Árvore AVL:

```
./run -i ./entrada.txt -o ./saida.txt -t arv;
```

- Exemplo de comando para executar dicionário com Tabela Hash:

```
./run -i ./entrada.txt -o ./saida.txt -t hash;
```