# Trabalho Prático 3

**Thaís Ferreira da Silva - 2021092571**

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

thaisfds@ufmg.br

## 1 Introdução

O problema proposto foi a implementação de um dicionário através de duas estruturas diferentes estudadas em sala de aula, uma arvore AVL e um hash. O dicionário deve ser capaz de executar a seguinte sequencia de tarefas:

- Criar um dicionario
- Inserir no dicionário criado todos os verbetes e seus significados a partir do arquivo de entrada
- Imprimir o dicionário resultante em ordem alfabética dos verbetes e ordem de inserção dos significados dos verbetes
- Remover todos os verbetes que tem pelo menos um significado
- Imprimir novamente o dicionário
- Destruir o dicionário

Para isso, é necessário que o dicionário seja uma classe abstrata com funções de inserir, remover e imprimir um verbete com o seu tipo e seus significados. O usuário terá que entrar com um arquivo .txt contendo os verbetes a serem adicionados e escolher durante a compilação o tipo de estrutura que o dicionário utilizará (arv para arvore AVL, e hash).

## 2 Método

O programa foi desenvolvido na linguagem C++, compilado pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificações:

- Sistema Operacional: Linux Ubuntu 22.04
- Processador: 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz 2.80 GHz
- RAM: 8,00 GB (utilizável: 7,80 GB)

### 2.1 Namespace

Para armazenar alguns dados importantes que seriam utilizados em diversos locais, achei mais prático criar um namespace par armazenar os dados de entrada.

- **Entradas:** Esse namespace é responsável por armazenar os dados de entrada inseridos ao compilar o código, sendo eles:
  - Nome do arquivo de entrada .txt com os verbetes a serem adicionados;
  - Nome do arquivo de saída .txt onde será impresso os verbetes adicionados antes e depois da remoção daqueles com significado.
  - Tipo escolhido pelo usuário para implementação do dicionário.

### 2.2 Classes

O programa é composto das seguintes classes principais: Lista, Verbete, Dicionario, Hash, e ArvoreAVL.

- **Lista:** Uma lista encadeada abstrata construída com template e typename que pode ser utilizada para qualquer tipo de variável. É possível adicionar e remover elementos em qualquer posição da lista. A ideia de se utilizar uma classe abstrata surgiu com a necessidade de implementar 2 listas com funções semelhantes, a lista de verbetes do hash e a lista de significados de um verbete.
- **Verbete:** Essa classe armazena o tipo do verbete, o nome do verbete, e uma lista de significados. Decidi por implementar o TAD de significado através da lista que já implementei. Essa classe permite setar um tipo e uma palavra, além de poder adicionar um significado e imprimir o mesmo.
- **Dicionario:** Essa classe abstrata contem as principais funcionalidade do dicionário como Inserir, Remover e Imprimir um verbete.
- **Hash:** Classe derivada do dicionário, que implementa o dicionário através de um hash. Alem das funçoes do pai, ele também calcula a função de hash utilizada.
- **ArvoreAVL:** Derivada da classe dicionário, ela também tem as funções de uma arvore, então ela é capaz de calcular o fator de balanceamento, balancear a arvore e realizar as rotações necessárias.

### 2.3 Funções

O programa é composto de diversas classes, então aqui estão as principais funçoes das principais classes citadas anteriormente.

- **Lista**
  - **adicionarInicio:** Essa função recebe um dado e adiciona ele na primeira posição da lista, fazendo com que o **novoNode** tenha seu próximo apontando para o **head/cabeça**, e passe a ocupar a posição do **head/cabeça**.
  - **adicionarNaPosicao:** Semelhante a função **acionarInicio**. Recebe um dado e a posição que devemos inserir ele na lista. Para realizar essa inserção, é necessário utilizar dois **Node** auxiliares que armazenaram o endereço do ultimo item analisado e do que está sendo analisado no momento, depois realiza pequenas manipulações para encaixar o dado. O novo nódulo criado aponta para o item analisado, e o ultimo item analisado passa a apontar para o item apontado pela analise atual.
  - **remover:** A função remove o primeiro elemento da lista e retorna o dado armazenado pela primeira posição. Para isso, se o **head/cabeça** não for vazio, armazenamos o seu

dado em uma variável e depois fazemos o **head/cabeça** ser o próximo elemento da lista.

- **removerNaPosicao:** Semelhante a função **remover**. Recebe a posição do dado que deseja remover, percorre os elementos em busca da posição desejada, e assim como ocorre no **adicionarNaPosicao** utilizamos de duas variáveis auxiliares para fazer a ultima analise apontar para o próximo nódulo da analise, e depois removemos a analise da lista.

- **Verbete**
  - **ImprimeSignificados:** Essa função é capaz de imprimir o verbete e seus significados em um um arquivo. O loop percorre todos os significados na lista e imprime os seus valores caso a lista exista.
  - **temSignificados:** Apenas retorna se a lista de significados possui algum elemento.
  - **addSignificado:** Adiciona um novo significado no verbete
- **Dicionario** Todas as classes do dicionário são asbstratas e são melhor definidas no hash e na arvoreAVL.
- **Hash**
  - **FuncHash:** A função hash que criei considera a primeira letra da palavra para separar no hash e retorna um numero de 0 a 51.
  - **Pesquisa:** Recebe o tipo e a palavra a ser pesquisada e busca através de um loop o verbete no hash, retornando null se não for encontrado.
  - **Insere:** cria um auxiliar que é utilizado para armazenar o ultimo verbete analisado ao percorrer o loop, dessa forma ele busca a posição na lista onde deve-se encaixar o novo verbete. Caso o verbete já exista no hash, o programa apenas adiciona um novo significado ao verbete existente.
  - **Remove:** Percorre todo o hash e caso encontre o verbete com o mesmo tipo e nome que inserido. Deleta o verbete do hash.
  - **RemoveComSignificado:** Realiza a mesma tarefa que o Remove, entretanto ao invés de retirar o verbete que possui o mesmo tipo e nome inserido, ele remove aqueles verbetes que possuem pelo menos um significado.
  - **Imprimir:** Imprime todos os verbetes presente no hash e seus significados caso existam.
- **ArvoreAVL**
  - **FatorBalanceamento:** Retorna o fator de balanceamento considerando quantos nos existem pra cada lado. Retorna 0 se o nodolo não existe, ou o valor da altura da esquerda + 1 - altura da direita + 1;
  - **Altura:** Retorna 1 se o nodulo não existe, retorna 0 caso esse no não possua filhos, ou a altura do maior nodulo a esqueda ou a direita +‘;
  - **RotacaoDireita:** Manda o pai para a direita e sobe o seu primeiro filho a esquerda na arvore.
  - **RotacaoEsquerda:** Manda o pai para a esquerda e sobe o seu primeiro filho a direita na arvore.
  - **RotacaoDDireita:** Realiza uma rotação pra esquerda no nodulo a esqueda e uma a direita com o nodulo atual.

- **RotacaoDEsquerda:** Realiza uma rotação pra direita no nodulo a direita e uma a esquerda com o nodulo atual.
- **BalancearRecursivo:** Através do fator de balanceamento realiza as rotações necessárias. Caso o fator seja 2, ele deve realizar uma rotação normal ou dupla para a direita, acaso esse valor do fator seja negativo, a rotação deve ocorrer para a esquerda.
- **PesquisaRecursivo:** Percorre toda a arvore com o tipo, e nome do verbete em busca do verbete escolhido. Caso encontrado retorna o verbete, e caso contrario retorna null;
- **InsereRecursivo:** Percorre toda a arvore adicionando o novo verbete a esquerda ou a direita de acordo com o percurso na arvore. Caso o verbete ja exista, ela adiciona o significado ao verbete ja existente.
- **RemoveRecursivo:** Percorre recursivamente a arvore em busca do nódulo a ser deletado. Caso ele exista realiza as manipulações necessárias par remover o nódulo, e depois rebalanceá a arvore se necessário.
- **RemoveComSignificadoRecursivo:** Realiza a mesma tarefa que o removeRecursivo, mas leva em consideração apenas se o verbete possui ou não um significado. Apagando assim os verbetes com significado.
- **ImprimirRecursivo:** Percorre a arvore imprimindo o verbete e seus significados caso existam.

# 3 Análise de Complexidade

## 3.1 Tempo

Uma arvore AVL por ser uma arvore binária percorre ou para a direita ou para a esqueda dos nódulos. Sendo assim, o tamanho sempre é dividido pela metade o que gera no pior caso um custo O(log n) nas funçoes de inserção, remoção e pesquisa. Já em relação ao balanceamento, podemos separar em 2 casos, quando inserimos um novo verbete, e quando removemos um novo verbete. Ao inserir um novo verbete, basta balancear o galho onde ele foi inserido gerando um custo O(log n), já na remoção é necessário percorrer toda a arvore para realizar o balanceamentro o que gera um custo O(n).

Em relação ao Hash, o custo será calculado de acordo com o tamanho das linhas contidas. Considerando que todas as listas não estão sempre completar podemos dizer que a complexidade do hash sera O(1+n/m) onde n é são os verbetes, e m a quantidade de listas dentro da tabela do hash.

É importante ressaltar que o hash desenvolvido por mim não é o mais eficiente, entretanto a ideia de divisão que tive levava em consideração a primeira letra do verbete a ser adicionado.

## 3.2 Espaço

Em relação a complexidade de espaço a analise deve ser feita pala a lista de significados, para a arvore AVL e para o hash, que são as estruturas onde temos alocação de espaço no programa. Sendo assim:

A lista de significados possui uma complexidade O(n) onde n é o numero de significados adicionados na lista encadeada. Ou seja a sua complexidade é proporcional ao numero de significado adicionados.

Já a arvore AVL possui complexidade de espaço também O(n) onde n é o número de verbetes adicionados na arvore binária. Ou seja a sua complexidade é proporcional ao número de nós presentes na estrutura.

Por fim o Hash possui complexidade de espaço O(n) onde n é o número de verbetes adicionados no hash. Ou seja a sua complexidade é proporcional ao número de verbetes presentes nos 52 espaços do hash.

# 4 Estratégias de Robustez

Para tornar o programa mais robusto foi utilizado do assert para verificar as principais exceções do programa. Foram implementadas algumas medidas para:

- Verificar se o tipo do verbete é valido (a,n,v)
- Verificar se o nome do verbete inserido não é vazio
- Verificar a o tipo de dicionario escolhido pelo usuário (arv ou hash)
- Verificar a extensão dos arquivos de entra e saída

Além disso, foram implementados outras estratégias de robustez onde o assert não foi necessário. As três estrategias foram:

- Definir um nome de saída padrão caso não seja inserido pelo usuário (output.txt)
- Define o tipo de dicionario como arvore AVL caso o nome inserido não seja válido

# 5 Conclusão

Nesse trabalho prático tivemos que implementar um dicionário através de duas estruturas diferentes, uma com um Hash e a outra com uma Árvore AVL.

Infelizmente por falta de tempo não foi possível realizar uma analise experimental, mas ao analisar com meus colegar consegui perceber que o hash é superior a arvore AVL se for entrado uma boa função de hash que evite o maior número de colisões possíveis. Entretando para a minha implementação de hash, prevalecente a arvore é melhor, pois implementei 52 tabelas para o hash em casos de teste onde existem mais de 1000 verbetes diferentes a serem inseridos no dicionário.

Por fim, pude aprender bastante sobre as duas formas de implementação, e quando um método pode ser considerado melhor do que outro. Acredito que essa prática foi extremamente útil para fixar o conteúdo visto em sala de aula, consolidando assim o hash e a arvore avl;

## References

DOS, C. Árvore AVL. Disponível em: <https://pt.wikipedia.org/wiki/%C3%81rvore_AVL>. Acesso em: 14 dez. 2022.

DOS, C. Árvore AVL. Disponível em: <https://pt.wikipedia.org/wiki/%C3%81rvore_AVL>. Acesso em: 14 dez. 2022.

Slides das aulas de Estrutura de Dados

## Instruções de Compilação e Execução

1. Abrir o terminal
2. Entrar na pasta tp
   Comando: cd / <caminho para a pasta /tp >
3. Compilar o programa com o Makefile disponibilizado na pasta
   Comando: make
4. Executar o programa
   Comando: ./bin/<nome do arquivo.out>
   - i <nome do arquivo de entrada .txt> (nome do arquivo de entrada)
   - o <nome do arquivo de saida .tx> (nome do arquivo de saída)
   - t <tipo de dicionario> (arv para arvore AVL, hash para o Hash)
5. Após terminar de utilizar o programa, delete os arquivos desnecessários
   Comando: make clean