

RAFAEL BALDASSO AUDIBERT

**Trabalho final para
Classificação e Pesquisa de Dados - INF01124
FEDERAL WORKER BLAMER**

Brasil

Dezembro de 2018

RAFAEL BALDASSO AUDIBERT

Trabalho final para
Classificação e Pesquisa de Dados - INF01124
FEDERAL WORKER BLAMER

Trabalho final para a cadeira de Classificação
e Pesquisa de Dados do Instituto de Informá-
tica - UFRGS

Universidade Federal do Rio Grande do Sul – UFRGS

Instituto de Informática

Classificação e Pesquisa de Dados - INF01124

Brasil

Dezembro de 2018

Resumo

Esse é o trabalho final para a cadeira de Classificação e Pesquisa de Dados, ministrada pelo [Professor Renan de Queiroz Maffei](#), no Instituto de Informática @ UFRGS.

O objetivo deste trabalho consistia em se aprofundar nos métodos, estruturas e algoritmos estudados na disciplina, em especial com o armazenamento e a manipulação de dados em arquivos com dados salvos no formato binário (i.e., não textual), a indexação, a pesquisa e a ordenação de dados.

O trabalho deveria envolver a coleta (ou extração) de dados brutos, o armazenamento em estruturas de arquivo (indexadas e em formato binário), a pesquisa e a ordenação de dados.

Nesse trabalho, foi utilizado como base de dados as informações que podem ser obtidas no Portal de Transparência do Governo Federal Brasileiro, mais especificamente os dados referentes aos servidores públicos federais. O trabalho visa fazer o trabalho de um banco de dados onde podem ser pesquisados funcionários para conseguir acessar alguns dados deles facilmente, principalmente focados na remuneração dos mesmos. É possível pesquisar no banco de dados através do nome do funcionário, da sua posição ou do órgão governamental do qual o mesmo se encontra em exercício.

Palavras-chaves: Classificação de Dados. Governo Federal. Servidor Público. Rust.

Sumário

1	INTRODUÇÃO	5
2	IMPLEMENTAÇÃO	7
2.1	Frameworks e Bibliotecas Adicionais	7
2.2	Parse	7
2.3	Record	8
2.4	Trie	9
2.5	Inserção de um Record	12
3	GUIA DE USO	13
3.1	CLI	13
3.2	Interativo	14
4	CONCLUSÕES FINAIS	17
	REFERÊNCIAS	19

1 Introdução

O problema que esse trabalho, apelidado de Federal Worker Blamer, visa solucionar é a pesquisa eficiente dos dados referentes a remuneração e dados pessoais dos servidores públicos federais. Todos os dados¹ se encontram disponibilizados no Portal da Transparência² do Governo Federal do Brasil, porém estão disponíveis através de uma interface visual lenta que mostra informações demais, muitas vezes irrelevantes.

Através da construção de um banco de dados local e a utilização de uma trie, formalmente uma árvore de prefixos, e a idéia de "arquivos invertidos"[1], é possível acessar os dados muito mais rapidamente (caso o queira, já que caso o usuário queira indexar muitas coisas, o sistema pode demorar um pouco para realizar a pesquisa na primeira vez que a mesma é feita).

O programa também suporta a capacidade de inserir os dados de um novo funcionário público no banco de dados, indexando-o corretamente nas tries, bastando preencher os mesmos dados que são guardados para os servidores públicos federais.

Por padrão, o programa tem a capacidade de pesquisar os funcionários através do seu nome, o seu cargo, ou o órgão no qual o mesmo se encontra em exercício. Utilizando a CLI (que será explicada mais adiante) podemos realizar uma pesquisa que filtra pelas três diferentes indexações.

¹ <http://www.portaltransparencia.gov.br/download-de-dados/servidores>

² <http://www.portaltransparencia.gov.br>

2 Implementação

O programa foi feito utilizando uma linguagem (consideravelmente nova) chamada Rust¹, voltada para desenvolvimento de aplicações críticas e sistemas embarcados "que roda incrivelmente rápido, previne falhas de segmentação, e garante segurança entre threads"[2], que não possui *Garbage Collector*, o que impede *memory leaks*, já que toda vez que você sai do escopo de uma variável, sua posição de memória é desalocada, e todos os ponteiros que a referenciam são invalidados. Como o trabalho tinha o intuito de ser ainda mais rápido que o sistema disponibilizado pelo Portal da Transparência, era necessário uma linguagem de baixo nível, porém capaz de realizar coisas poderosas (criação de threads, alocação dinâmica de estruturas, liberação de memória alocada, etc.) de maneira mais simples e segura.

2.1 Frameworks e Bibliotecas Adicionais

Foram utilizadas 4 bibliotecas adicionais que podem ser encontradas no topo do arquivo *main.rs*, que são: *prettytable*², *text_io*³, *clap*⁴ e *csv*⁵, todas disponíveis no repositório oficial dos pacotes (crates) de Rust, Crates.io⁶. As 3 primeiras são responsáveis apenas pela criação da interface, I/O do console e criação de um sistema de CLI, respectivamente. A biblioteca *csv* é a única realmente utilizada para geração de código, e é a responsável por ler dos arquivos *.csv* e realizar o parse dos mesmos para manipulação durante a etapa de criação do banco de dados, a ser descrita posteriormente.

2.2 Parse

O programa cria um banco de dados (guardados no formato binário, com os dados distribuídos serialmente) a partir de 2 arquivos disponibilizados pelo Governo Federal, com nomes no formato *<ano><mes>_Remuneracao.csv* e *<ano><mes>_Cadastro.csv*, que contém informações sobre a remuneração dos funcionários e dados sobre órgão de exercício, datas de afastamento, cargos, etc, respectivamente. Para poder criar esse banco de dados, primeiro é necessário *parsear* o arquivo referente a remuneração, já que ele não se encontra ordenado da mesma maneira que o arquivo referente aos dados, o que impede a realização do merge dos dois arquivos rapidamente em

¹ <https://www.rust-lang.org/pt-BR/>

² <https://crates.io/crates/prettytable-rs>

³ https://crates.io/crates/text_io

⁴ <https://crates.io/crates/clap>


⁵ <https://crates.io/crates/csv>

⁶ <https://crates.io>

running time, então primeiramente é necessário rodar um script em Python que ordena os arquivos de maneira similar, de um jeito que o programa em Rust consiga realizar o merge de maneira eficiente.

2.3 Record

As estruturas que são utilizadas no banco de dados e armazenam todas as informações sobre os funcionários públicos são chamadas de *records* e elas armazenam várias informações que são parseadas dos 2 arquivos *.csv* que são passados para o programa. A sua estrutura é a seguinte:



```
pub struct Record {  
    pub nome: Vec<u8>,  
    pub id: Vec<u8>,  
    pub cpf: Vec<u8>,  
    pub descricao_cargo: Vec<u8>,  
    pub orgao_exercicio: Vec<u8>,  
    pub remuneracao_basica_bruta_rs: Vec<u8>,  
    pub gratificacao_natalina_rs: Vec<u8>,  
    pub ferias_rs: Vec<u8>,  
    pub outras_remuneracoes_eventuais_rs: Vec<u8>,  
    pub irrf_rs: Vec<u8>,  
    pub pss_rgps_rs: Vec<u8>,  
    pub demais_deducoes_rs: Vec<u8>,  
    pub remuneracao_apos_deducoes_obrigatorias_rs: Vec<u8>,  
    pub total_verbas_indenizatorias_rs: Vec<u8>,  
    pub data_inicio_afastamento: Vec<u8>,  
    pub data_termino_afastamento: Vec<u8>,  
    pub jornada_trabalho: Vec<u8>,  
    pub data_ingresso_cargo: Vec<u8>,  
    pub data_ingresso_orgao: Vec<u8>,  
}
```

Figura 1 – Representação em Rust da estrutura de um Record


Cada dado representado nessa estrutura possui como tipo *Vec<u8>* que poderia ser traduzido para C++ como *std::vector<uint8_t>*. Basicamente, ele funciona como um *std::vector<char>* porém realmente armazenando o valor ASCII do caractere como *uint8_t* facilitando a utilização de funções built-in do Rust para conversão de *Vec<u8>* para *String*. Foi adotado um padrão para o tamanho máximo que cada um desses campos pode ter, dessa maneira cada uma dessas estruturas ocupa exatamente 323 bytes no arquivo do

banco de dados, dispostos serialmente, um após o outro, assim que são parseados dos arquivos *.csv*.

A maior dificuldade enfrentada nessa etapa, foi decidir como salvar isso no banco de dados, já que se cada um dos *Record* contivesse tamanho diferente, teríamos uma complexidade de $\Theta(n)$ para encontrarmos um dado dentro do arquivo, já que precisaríamos percorrer cada um dos *Record* para saber onde se encontrava o início do próximo. Como eles possuem todos o mesmo tamanho, a partir do conhecimento da posição que um dado encontra dentro do arquivo de banco de dados, temos uma busca em $\Theta(1)$, já que sabemos exatamente quantas posições "pular" para chegar ao *Record*.

2.4 Trie

Após a criação do banco de dados, são gerados os arquivos binários referentes as *tries* que servem como indexação para o arquivo binário serial do banco de dados. As *tries* possuem a seguinte estrutura em Rust:



```
pub struct Node {  
    chars: HashMap<char, u32>,  
    val: Vec<u32>,  
    address: u32,  
}  
  
pub struct Trie {  
    nodes: Arena,  
    root: u32,  
}  
  
pub struct Arena {  
    nodes: Vec<Node>,  
}
```

Figura 2 – Representação em Rust da estrutura da Trie

Os tipos dessa estrutura poderiam tentar ser traduzidos para C++ da seguinte maneira: `HashMap<_, _>` pode ser traduzido para `std::unordered_map<_, _>`, `Vec<_>` pode ser traduzido para `std::vector<_>`, e `u32` pode ser traduzido para `unsigned int`^{[3][4]}. Basicamente, na minha *Trie* eu posso uma estrutura chamada *Arena* que armazena todos os nodos que se encontram na minha trie, além de um `u32` que guarda qual posição da arena se encontra o meu nodo raiz. Cada um dos nodos possui uma *HashMap* que é uma Tabela Hash de caracteres para um `u32` que significa "O nodo do caractere *x* se encontra na posição *y* da minha arena". Eu também possuo um `Vec<u32>` guardando todos os valores possíveis para aquele nodo, que são a posição dentro do arquivo do banco de dados que correspondem à aquele nodo. O último campo, é utilizado quando eu vou salvar a minha trie no arquivo binário, que basicamente corresponde qual o endereço físico dela dentro do disco. O processo para calcular isso será explicado abaixo, junto com a representação na

memória de cada nodo da *Arena*.

Basicamente, o que é guardado no arquivo binário é a estrutura *Arena*, já que por padrão o nodo *root* se encontra na posição 0 da mesma, e não precisa ser escrito no arquivo binário.

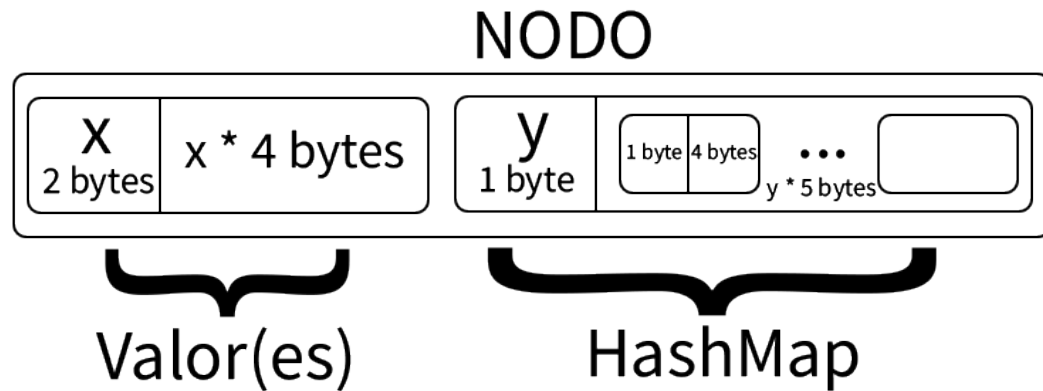


Figura 3 – Representação de um nodo no arquivo da Trie

Os nodos são salvos no arquivo na ordem que estão na *Arena* dentro da Trie que está em memória. Todos os campos estão salvos no formato *little-endian*⁷. Os 2 primeiros bytes (que chamaremos de *X*) representam o tamanho do vetor de valores, e em seguida temos $X * 4$ bytes, sendo que cada 4 bytes representam um dos valores que estavam armazenados nesse nodo e representam uma posição dentro do arquivo do banco de dados. Na segunda parte do nodo, começamos com 1 byte (que, conseqüentemente, chamaremos de *Y*) que representa a quantidade de filhos que esse nodo da *Trie* tinha. Os próximos $Y * 5$ bytes são uma representação da *HashMap* que cada nodo possuía na memória. Cada posição da *HashMap* é representado por 1 byte, com o valor em ASCII daquele caractere, e 4 bytes representando qual posição dentro do próprio arquivo binário da *Trie* se encontra o nodo correspondente a esse caractere. Para sabermos onde o nodo vai estar dentro do próprio arquivo, seria necessário escrevermos o nodo, sabermos onde ele foi escrito, e voltar preenchendo todos esses campos novamente, sendo necessário escrever o arquivo 2 vezes, o que geraria um tempo de execução imenso, porém, foi utilizado uma estratégia em que eu calculo previamente o tamanho que cada um dos nodos vai ocupar na árvore, já que eu sou capaz de fazer as contas descritas acima, e conseqüentemente eu consigo saber em que posição meu filho vai ocupar quando ele for escrito daqui alguns milissegundos, calculando o offset para cada nodo presente dentro da *Arena* enquanto ela ainda está na memória.

Uma parte interessante de como o programa executa a criação desses arquivos com as tries, é que ele utiliza 1 thread para cada trie que está sendo criada. Por padrão, o programa indexa os campos de *nome*, *descricao_cargo* e *orgao_exercicio*, poupando

⁷ <https://en.wikipedia.org/wiki/Endianness>

cerca de 55% do tempo na criação das tries. Ele inicializa as 3 threads ao mesmo tempo, e somente da continuidade à execução do programa assim que as 3 finalizarem, para evitarmos de termos dados incompatíveis no disco. Porém, é necessário tomar cuidado com quais campos estamos indexando, já que com a base de dados do governo a thread que indexa os nomes ocupa cerca de 5GB de memória RAM já que ela mantém toda a trie na memória antes de escrevê-la (poderia ser otimizado para criar a trie diretamente em disco, mas o algoritmo seria muito complexo, já que precisaria aumentar o tamanho das *Hash Tables* no disco enquanto está escrevendo no mesmo, provavelmente levando mais tempo do que já leva atualmente).

2.5 Inserção de um Record

Para realizar a inserção de um novo "funcionário" é criado um novo *Record* que é inserido no final do arquivo do banco de dados. Para a indexação desse nome na trie, é criado um arquivo adicional de overflow de Trie, que contém somente os valores adicionados após a geração inicial da Trie. Como essa Trie é muito menor, ela pode rapidamente ser carregada na memória, incrementada com a indexação do novo funcionário e salva na memória. Dentro do programa existe a opção de recriar a estrutura da trie principal, fazendo com que o arquivo principal da *Trie* seja refeito, e o arquivo de overflow seja excluído. Em uma aplicação real, provavelmente assim que a *Trie* de overflow chegasse em um determinado tamanho (na minha opinião, 512 bytes), um novo processo seria criado para realizar a união dos mesmos, fazendo o programa rodar em plano de fundo, fazendo com que o arquivo principal de *Trie* ficasse constantemente atualizado, acelerando a procura já que não seria necessário procurar cada nome nos 2 arquivos.

3 Guia de Uso

O programa tem 2 modos de uso: *CLI* e *Interativo*.

3.1 CLI

Para saber todas as opções de rodá-lo em modo *CLI* você pode chamar o programa com a flag `-h` ou `-help` e ele lhe dará uma explicação detalhada sobre os modos de utilizar a mesma, conforme figura abaixo.

```
Federal Worker Blamer 1.0.0
Rafael B. Audibert <rbaudibert@inf.ufrgs.br>
Search some data about the brazilian federal workers

USAGE:
  trabalho_final_cpd [FLAGS] [OPTIONS]

FLAGS:
  -h, --help      Prints help information
  -i              Runs the program in the interactive mode
  -n              Asks to insert a new entry in the database
  -o              Runs the program CLI searches using OR instead of AND searches
  -s              Runs the program using the prefix_search (CAREFUL)
  -t              Asks to remake the trie, based in the DATABASE.BIN file
  -V, --version   Prints version information

OPTIONS:
  -a, --agency_name <agency_name>  Chooses the agency name which will be searched in the database
  -c, --csv <csv> <csv>             Sets the CSV files which will be used to generate the database
  -e, --entry <entry>               Chooses the entry which will be searched in the database
  -p, --person_name <person_name>   Chooses the person name which will be searched in the database
  -r, --role_name <role_name>       Chooses the role which will be searched in the database
```

Figura 4 – Imagem do texto gerado pela flag `-h` da CLI

Segue as opções disponíveis e como utilizar elas:

- `h`, `help` ⇒ Mostra informações a respeito do programa, e explica sobre a utilização da CLI;
- `i` ⇒ Roda o programa no modo interativo;
- `n` ⇒ Roda o programa para inserir um novo funcionário no banco de dados;
- `o` ⇒ Roda o modo CLI utilizando buscar com OR, ao contrário de AND (será explicado mais adiante);
- `s` ⇒ Roda o programa utilizando a função de árvore de prefixos da trie (tome cuidado com sua pesquisa, ela pode retornar muitos resultados);

- `t` \Rightarrow Roda o programa para refazer a trie, fazendo o merge dela com os arquivos de overflow;
- `V`, `version` \Rightarrow Mostra informações a respeito da versão do programa (1.0.0);
- `a`, `agency_name` \Rightarrow Realiza uma pesquisa no banco de dados utilizando o index do nome do órgão em que o funcionário está em exercício. Recebe um parâmetro, que é o nome do órgão a ser pesquisado;
- `c`, `csv` \Rightarrow Cria o banco de dados. Recebe 2 parâmetros, que são os arquivos de remuneração e de cadastro, respectivamente, em formato `.csv`;
- `e`, `entry` \Rightarrow Realiza uma pesquisa no banco de dados acessando uma determinada posição do mesmo. Recebe um parâmetro, que é a posição a qual será pesquisada.
- `p`, `person_name` \Rightarrow Realiza uma pesquisa no banco de dados utilizando o index do nome do funcionário. Recebe um parâmetro, que é o nome do funcionário a ser pesquisado;
- `r`, `role_name` \Rightarrow Realiza uma pesquisa no banco de dados utilizando o index do cargo do funcionário. Recebe um parâmetro, que é o nome do cargo a ser pesquisado;

Caso voce utilize as opções *a*, *p* ou *r*, o programa retorna a pesquisa com um "AND"entre todos os valores retornados, por exemplo: `"federal_worker_blamer.exe -a 'PRESIDENCIA DA REPUBLICA' -p 'RAFAEL BALDASSO AUDIBERT'"`retorna todos os funcionários do órgão PRESIDENCIA DA REPUBLICA que possuam o nome RAFAEL BALDASSO AUDIBERT. Porém, caso você utilize o programa utilizando a flag `-o`então ele irá rodar o programa buscando todas as pessoas com aquele nome MAIS as pessoas que trabalham no dado órgão, executando uma operação de "OR"entre os valores retornados.

3.2 Interativo

Ao rodar o programa utilizando a flag `-i` você entra no modo interativo que conta com a seguinte tela:


```
===== FEDERAL WORKER BLAMER =====
Choose an option below:
1. Generate the database (ALL YOUR DATA WILL BE LOST)
2. Choose a person name to be searched in the database
3. Choose a role in the brazilian civil service to be searched in the database
4. Choose a brazilian federal agency to be searched in the database
5. Insert a new worker in the database
6. Rebuilds the database-indexes (CAREFUL, IT WILL TAKE A WHILE)
7. Exit
Your choice: 
```

Figura 5 – Captura de tela do modo interativo do programa

As opções disponíveis no mesmo, com explicações sobre as mesmas são as seguintes:

1. **Generate the database** ⇒ Caso você escolha essa opção, ele irá lhe pedir para colocar os 2 arquivos que serão utilizados para gerar o banco de dados, e o gerará. Também serão geradas as árvores trie que indexam o conteúdo do arquivo de banco de dados. Tome cuidado, essa opção apaga todo o banco de dados que você possui, e pode demorar um bom tempo para ser refeito, já que as árvores tries demoram para ser construídas, e consome muita memória, já que as árvores tries ficam instanciadas em memória AO MESMO TEMPO, ocupando vários GB de memória com um dataset muito grande;
2. **Choose a person name to be seached in the database** ⇒ Insira o nome de uma pessoa que você deseja buscar no banco de dados. Caso você tenha inicializado o programa com a flag -s, ele irá realizar a busca utilizando a funcionalidade de árvore de prefixos;
3. **Choose a role in the brazilian civil service to be seached in the database** ⇒ Insira o nome de um cargo do funcionalismo público brasileiro que você deseja buscar no banco de dados. Caso você tenha inicializado o programa com a flag -s, ele irá realizar a busca utilizando a funcionalidade de árvore de prefixos;
4. **Choose a brazilian federal agency to be seached in the database**⇒ Insira o nome de um órgão do governo federal brasileiro que você deseja buscar no banco de dados. Caso você tenha inicializado o programa com a flag -s, ele irá realizar a busca utilizando a funcionalidade de árvore de prefixos;
5. **Insert a new worker in the database** ⇒ Insira os dados referentes a um "novo trabalhador"que você gostaria de inserir no banco de dados. Serão pedidos todos os dados referentes a esse funcionário, e ele passará a estar disponível na pesquisa pelos 3 diferentes índices. Esse funcionário é indexado pela trie de overflow até que a opção 6 seja executada;

6. **Rebuilds the database-indexes** \Rightarrow Gera novamente as tries, utilizando os valores que se encontram no banco de dados. Quaisquer funcionários que estavam nas tries de overflow passarão a estar na trie principal. Tome cuidado, essa opção pode demorar um bom tempo para ser executada, já que as árvores tries demoram para ser construídas, e consome muita memória, já que as árvores tries ficam instanciadas em memória AO MESMO TEMPO, ocupando vários GB de memória com um dataset muito grande;
7. **Exit** \Rightarrow Sai do programa.

4 Conclusões Finais

Em um computador suficientemente rápido o programa satisfaz o problema que ele buscava resolver, sendo mais rápido do que o próprio site da Transparência do Governo Federal. Um contra da utilização desse programa é que a geração das tries é insuperavelmente demorada e custosa, tanto de processamento quanto de memória, RAM e disco (5GB e quase 200MB, respectivamente).

O programa tem certas limitações, porém. Uma delas é o fato de não conseguirmos pesquisar pessoas específicas caso não saibamos algum nome do meio. Por exemplo, para o nome RENAN DE QUEIROZ MAFFEI, podemos encontrá-lo a partir de RENAN, QUEIROZ, MAFFEI, RENAN DE QUEIROZ, QUEIROZ MAFFEI, RENAN DE QUEIROZ MAFFEI, mas se realizássemos uma pesquisa com RENAN MAFFEI, não encontraríamos os resultados desejados, já que o programa não faz a indexação de todas as possibilidades de nomes, já que o arquivo de index ficaria extremamente pesado, já que geramos um número maior que $n!$ combinações para um nome com n palavras.

Uma solução muito eficiente para esse problema foi encontrada, se baseando na indexação de cada uma das palavras do nome da pessoa (ex.: RENAN, QUEIROZ, DE, MAFFEI) e ao realizarmos uma pesquisa, (ex.: RENAN MAFFEI) separamos os nomes pesquisados, realizamos 2 consultas ao banco de dados, e utilizamos a interseção das mesmas, do mesmo jeito que é realizada a consulta por múltiplos filtros através da CLI. A implementação não seria custosa e pouco mudaria a lógica do programa, porém não foi implementada.

A utilização das árvores tries conforme aprendido na disciplina foi fundamental para a realização do trabalho. Mas importante ainda foi o conceito de *arquivos invertidos* já que, por padrão, iríamos colocar toda a estrutura no nodo da árvore, e não apenas um "*ponteiro*" para a sua localização física no disco, como foi implementado nesse programa. Caso os salários tivessem sido indexados, como eu tinha idéia no início, árvores B+ teriam sido usadas, já que elas são muito eficientes quando temos um *range* de valores que queremos acessar, porém as dificuldades exercidas pela linguagem (que eu nunca tinha usado antes) e a necessidade de trabalhar diretamente com a leitura de bytes acabou fazendo com que não houvesse tempo suficiente para implementar essa *feature*.

Referências

- [1] Wikipedia, *Inverted Index*. https://en.wikipedia.org/wiki/Inverted_index Citado na página 5.
- [2] Open source language, *Rust Lang*. <https://www.rust-lang.org/pt-BR/> Citado na página 7.
- [3] Tipos primitivos definidos na linguagem Rust, *Data Types - The Rust Programming Language*, <https://doc.rust-lang.org/book/second-edition/ch03-02-data-types.html> Citado na página 10.
- [4] Tipos primitivos definidos na linguagem C++, *cplusplus.com - C++ reference*, <http://www.cplusplus.com/> Citado na página 10.
- [5] Imortal, Gigante, Sagrado e Maravilhoso, *Stack Overflow*, <https://stackoverflow.com> Nenhuma citação no texto.
- [6] Rust reference, *Rust Cookbook - Ed. 2* <https://rust-lang-nursery.github.io/rust-cookbook/> Nenhuma citação no texto.