Trabalho Prático 1
Resolvedor de Expressão Numérica

Isabela Saenz Cardoso

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

isabelasaenz@ufmg.br
Matrícula: 2021040032

1. Introdução

Este sistema implementa uma árvore de expressão para resolver expressões matemáticas, e possui suporte a modificação da representação da expressão numérica, assim como a possibilidade de solução automática da expressão. O código é dividido em várias classes e funções, incluindo uma classe *Pilha* para armazenar elementos temporários durante a construção da árvore de expressão e classes *No* e *Arvore* para representar os nós e a estrutura da árvore de expressão.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection em um Subsistema Windows para Linux (WSL2).

2.1. Estruturas de dados

O sistema utiliza duas estruturas de dados principais: *Pilha* e *Arvore*. A classe *Pilha* é uma implementação de uma pilha genérica, que pode armazenar elementos de qualquer tipo. Já a classe *Arvore* representa uma árvore binária de expressões, onde cada nó armazena um valor numérico ou um operador.

2.1.1. Árvore Binária

A árvore binária é usada para armazenar e representar a expressão matemática. Cada nó da árvore armazena um número ou um operador e possui até dois filhos, representando os operandos da operação. A classe *No* é utilizada para representar os nós da árvore, e a classe *Arvore* é usada para gerenciar a árvore como um todo. A árvore binária permite realizar operações de inserção, remoção, busca e percurso em ordem, pós-ordem e pré-ordem.

2.1.2. Pilha

A pilha é uma estrutura de dados linear que segue o princípio de LIFO (Last In, First Out), ou seja, o último elemento inserido é o primeiro a ser removido. As pilhas são

utilizadas para auxiliar na construção da árvore binária a partir de uma expressão infixa ou posfixa. As pilhas são implementadas pela classe Pilha. As principais operações em uma pilha são *push* (inserção), *pop* (remoção) e *getTopo* (consulta do elemento no topo da pilha).

2.2. Classes

A classe *Expressao* apresentada no código é responsável por realizar diversas operações relacionadas a expressões matemáticas. As principais funcionalidades desta classe incluem:

- Ler uma expressão matemática infixa e construir uma árvore de expressão correspondente (*lerInfixa*).
- Ler uma expressão matemática posfixa e construir uma árvore de expressão correspondente (*lerPosfixa*).
- Obter a representação infixa da expressão (*Infixa*).
- Obter a representação posfixa da expressão (*Posfixa*).
- Resolver a expressão matemática e retornar o resultado (*resolve*).
- Verificar se a árvore de expressão está vazia (*vazia*).

2.2.1. lerInfixa

A função lerInfixa é responsável por ler uma expressão matemática infixa (e.g., 2 * (3 + 4)) e construir uma árvore de expressão correspondente. Ao final do processo, a raiz da árvore é atualizada com o resultado final. Para isso, a função utiliza duas pilhas: uma para armazenar os nós e outra para armazenar os operadores.

Durante o processo, a função verifica o tipo do token encontrado (número, parêntese esquerdo, parêntese direito ou operador) e realiza as operações correspondentes. Caso o token seja um número, ele é inserido na pilha de nós. Se for um parêntese esquerdo, é inserido na pilha de operadores. Se for um parêntese direito, são processados os operadores dentro dos parênteses. Já se for um operador, processa os operadores fora dos parênteses.

2.2.2. lerPosfixa

A função *lerPosfixa* é responsável por ler uma expressão matemática posfixa (e.g., 2 3 4 + *) e construir uma árvore de expressão correspondente. Neste caso, a função utiliza apenas uma pilha para armazenar os nós.

Assim como na função *lerInfixa*, a função verifica o tipo do token encontrado (número ou operador) e realiza as operações correspondentes. Caso o token seja um número, ele é inserido na pilha de nós. Se for um operador, realiza a operação com os dois últimos elementos da pilha de nós e insere o resultado novamente na pilha.

Ao final do processo, a árvore é verificada para garantir que possui apenas um elemento ou que está vazia. Caso contrário, a árvore é limpa.

2.2.3. Infixa e Posfixa

A função *Infixa* retorna a representação infixa da expressão matemática, utilizando a função *inOrdemParenteses* da árvore.

A função *Posfixa* retorna a representação posfixa da expressão matemática, utilizando a função *posOrdem* da árvore.

2.2.4. resolve

A função *resolve* é responsável por resolver a expressão matemática e retornar o resultado. Para isso, a função utiliza a notação posfixa da expressão e uma pilha auxiliar.

Durante o processo, a função verifica se o elemento é um número ou um operador. Se for um número, converte para double e insere na pilha auxiliar. Se for um operador, realiza a operação com os dois últimos elementos da pilha auxiliar e insere o resultado novamente na pilha. Ao final do processo, a função retorna o resultado da expressão.

2.2.5. vazia

A função *vazia* verifica se a árvore de expressão está vazia, retornando true se estiver vazia e false caso contrário.

3. Análise de complexidade

3.1. Expressao::lerInfixa

Tempo:

- O loop principal percorre a string de entrada uma vez, com tempo O(n), onde n é o tamanho da string.
- Dentro do loop principal, há operações de pilha, como *push*, *pop*, *getTopo* e *vazia*, todas com tempo O(1).
- As chamadas para a função *lerTipoString* também são O(1).

Portanto, a complexidade de tempo total é O(n).

Espaço:

- A função utiliza duas pilhas: *pilhaNo* e *pilhaOperador*.
- No pior caso, ambas as pilhas podem conter elementos na ordem de O(n).

Portanto, a complexidade de espaço total é O(n).

3.2. Expressao::lerPosfixa

Tempo:

- O loop principal percorre a string de entrada uma vez, com tempo O(n), onde n é o tamanho da string.
- As operações de pilha e as chamadas para a função *lerTipoString* são O(1).

Portanto, a complexidade de tempo total é O(n).

Espaço:
- A função utiliza uma pilha.
- No pior caso, a pilha pode conter elementos na ordem de O(n).

Portanto, a complexidade de espaço total é O(n).

3.3. Expressao::Infixa e Expressao::Posfixa

Tempo:
- As funções *inOrdemParenteses* e *posOrdem* são chamadas, que percorrem a árvore em tempo O(n), onde n é o número de nós.

Portanto, a complexidade de tempo total para ambas as funções é O(n).

Espaço:
- Não há uso significativo de espaço adicional nessas funções.
- A complexidade de espaço total é O(1).

3.3. Expressao::resolve

Tempo:
- A função utiliza a pilha posfixa da expressão, percorrendo-a em tempo O(n), onde n é o número de elementos na pilha.
- As operações de pilha e as chamadas para a função *lerTipoString* são O(1).

Portanto, a complexidade de tempo total é O(n).

Espaço:
- A função utiliza duas pilhas: *pilhaPosOrdem* e *pilhaAux*.
- No pior caso, ambas as pilhas podem conter elementos na ordem de O(n).

Portanto, a complexidade de espaço total é O(n).

3.4. Expressao::vazia

Tempo:
- A função chama o método *vazia* da árvore, que é O(1).

Portanto, a complexidade de tempo total é O(1).

Espaço:
- Não há uso significativo de espaço adicional nessa função.
- A complexidade de espaço total é O(1).

3.5. lerTipoString - exemplo usado na análise experimental

A complexidade da função lerTipoString pode ser determinada da seguinte forma:

- A complexidade de acessar o mapa instrucoes através da função *find* é O(log k), onde k é o número de elementos no mapa. Neste caso, k é uma constante (4 elementos), então a complexidade é O(1).
- A complexidade da função *isDouble* é O(n), onde n é o tamanho da string de entrada, pois a leitura da string é feita caractere por caractere.
- A complexidade da função *isChar* é O(1), pois verifica a igualdade do caractere com um conjunto fixo de caracteres.

Dado que a complexidade das operações envolvidas na função lerTipoString são O(1) e O(n), a complexidade geral da função é dominada pela função isDouble, resultando em uma complexidade de O(n).

Em resumo, a complexidade do programa é dominada pelas funções *lerInfixa*, *lerPosfixa* e *resolve*, todas com complexidade de tempo O(n) e complexidade de espaço O(n). As outras funções têm complexidades menores e não afetam significativamente a análise geral.

4. Estratégias de Robustez

Várias estratégias de robustez são para lidar com possíveis erros e garantir a correção do programa:

1. Checagem de árvore vazia antes de realizar operações. Isso evita tentativas de acesso a elementos que não existem, o que poderia causar erros ou comportamento indefinido

2. Limpeza da árvore antes de construir uma nova. Verifica se a árvore já possui elementos e, se necessário, realiza a limpeza (*arvore.clean();*) antes de construir uma nova árvore.

3. Tratamento de possíveis exceções utilizando blocos try-catch para tratar possíveis exceções que possam ocorrer durante a leitura da expressão infixa.

4. Prevenção de divisão por zero na função *resolve()*.

5. Verificação da pilha após a leitura. Na função *lerPosfixa()*, após processar a expressão, o código verifica se a pilha possui apenas um elemento e se a árvore não está vazia.

5. Análise Experimental

A análise experimental foi feita usando a ferramenta gprof, que serve para medir o tempo gastos pelas funções de um algoritmo. O primeiro arquivo de entrada possui 86 caracteres (*entdouble.s4.n5.p*), e o segundo possui 946 caracteres (*entdouble.s35.n50.i*).

O aumento no número de chamadas pode ser explicado pela natureza linear da função O(n). À medida que a entrada cresce, o número de chamadas da função também aumenta proporcionalmente.

O *flat profile* mostra a quantidade total de tempo que o programa gastou executando cada função.

Alguns dos resultados de *entdouble.s4.n5.p*:

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00      101     0.00     0.00  std::_Select1st<std::pair<std::__cxx11::basic_string<char, std::char_traits<ch
  0.00      0.00     0.00       92     0.00     0.00  __gnu_cxx::__aligned_membuf<std::pair<std::__cxx11::basic_string<char, std::ch
  0.00      0.00     0.00       92     0.00     0.00  __gnu_cxx::__aligned_membuf<std::pair<std::__cxx11::basic_string<char, std::ch
  0.00      0.00     0.00       92     0.00     0.00  std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char, std::char_traits
  0.00      0.00     0.00       91     0.00     0.00  std::_Rb_tree_const_iterator<std::pair<std::__cxx11::basic_string<char, std::c
  0.00      0.00     0.00       87     0.00     0.00  std::less<std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
  0.00      0.00     0.00       87     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       87     0.00     0.00  bool std::operator< <char, std::char_traits<char>, std::allocator<char> >(std
  0.00      0.00     0.00       67     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       56     0.00     0.00  std::remove_reference<std::__cxx11::basic_string<char, std::char_traits<char>,
  0.00      0.00     0.00       56     0.00     0.00  std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
  0.00      0.00     0.00       45     0.00     0.00  bool std::operator==<char, std::char_traits<char>, std::allocator<char> >(std
  0.00      0.00     0.00       44     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       36     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       32     0.00     0.00  std::_Rb_tree_iterator<std::pair<std::__cxx11::basic_string<char, std::char_tr
  0.00      0.00     0.00       32     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       24     0.00     0.00  std::map<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat
  0.00      0.00     0.00       24     0.00     0.00  std::map<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat
  0.00      0.00     0.00       24     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       24     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       24     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       24     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
```

```
  0.00      0.00     0.00       24     0.00     0.00  std::operator!=(std::_Rb_tree_const_iterator<std::pair<std::__cxx11::basic_str
  0.00      0.00     0.00       21     0.00     0.00  std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
  0.00      0.00     0.00       20     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       18     0.00     0.00  lerTipoString(std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       17     0.00     0.00  isDouble(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat
  0.00      0.00     0.00       16     0.00     0.00  __gnu_cxx::__aligned_membuf<std::pair<std::__cxx11::basic_string<char, std::ch
  0.00      0.00     0.00       16     0.00     0.00  __gnu_cxx::__aligned_membuf<std::pair<std::__cxx11::basic_string<char, std::ch
  0.00      0.00     0.00       16     0.00     0.00  std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char, std::char_traits
  0.00      0.00     0.00       16     0.00     0.00  std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
  0.00      0.00     0.00       16     0.00     0.00  operator new(unsigned long, void*)
  0.00      0.00     0.00       15     0.00     0.00  Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<
  0.00      0.00     0.00       14     0.00     0.00  std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
  0.00      0.00     0.00       14     0.00     0.00  std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
  0.00      0.00     0.00       10     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00       10     0.00     0.00  std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
  0.00      0.00     0.00        9     0.00     0.00  isChar(char)
  0.00      0.00     0.00        9     0.00     0.00  Arvore::vazia()
  0.00      0.00     0.00        8     0.00     0.00  __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::__cxx11::basic_stri
  0.00      0.00     0.00        8     0.00     0.00  void __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::__cxx11::basic
  0.00      0.00     0.00        8     0.00     0.00  __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::__cxx11::basic_stri
  0.00      0.00     0.00        8     0.00     0.00  void __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::__cxx11::basic
  0.00      0.00     0.00        8     0.00     0.00  __gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::__cxx11::basic_stri
  0.00      0.00     0.00        8     0.00     0.00  std::_Rb_tree_const_iterator<std::pair<std::__cxx11::basic_string<char, std::c
```

Podemos observar que a função *lerTipoString* foi chamada um total de 18 vezes, logo seguida por *isDouble*.

Alguns dos resultados de *entdouble.s35.n50.i*:

% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	3439	0.00	0.00	std::_Rb_tree_const_iterator<std::pair<std::__cxx11::basic_string<char, std::c
0.00	0.00	0.00	2612	0.00	0.00	std::_Select1st<std::pair<std::__cxx11::basic_string<char, std::char_traits<ch
0.00	0.00	0.00	2603	0.00	0.00	__gnu_cxx::__aligned_membuf<std::pair<std::__cxx11::basic_string<char, std::ch
0.00	0.00	0.00	2603	0.00	0.00	__gnu_cxx::__aligned_membuf<std::pair<std::__cxx11::basic_string<char, std::ch
0.00	0.00	0.00	2603	0.00	0.00	std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char, std::char_traits
0.00	0.00	0.00	2598	0.00	0.00	std::less<std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
0.00	0.00	0.00	2598	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
0.00	0.00	0.00	2598	0.00	0.00	bool std::operator< <char, std::char_traits<char>, std::allocator<char> >(std:
0.00	0.00	0.00	2578	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
0.00	0.00	0.00	2487	0.00	0.00	bool std::operator==<char, std::char_traits<char>, std::allocator<char> >(std:
0.00	0.00	0.00	1719	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
0.00	0.00	0.00	873	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
0.00	0.00	0.00	861	0.00	0.00	std::map<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat
0.00	0.00	0.00	861	0.00	0.00	std::map<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat
0.00	0.00	0.00	861	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
0.00	0.00	0.00	861	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
0.00	0.00	0.00	861	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
0.00	0.00	0.00	861	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al
0.00	0.00	0.00	861	0.00	0.00	std::operator==(std::_Rb_tree_const_iterator<std::pair<std::__cxx11::basic_str
0.00	0.00	0.00	861	0.00	0.00	std::operator!=(std::_Rb_tree_const_iterator<std::pair<std::__cxx11::basic_str
0.00	0.00	0.00	760	0.00	0.00	std::remove_reference<std::__cxx11::basic_string<char, std::char_traits<char>,
0.00	0.00	0.00	570	0.00	0.00	lerTipoString(std::__cxx11::basic_string<char, std::char_traits<char>, std::al
0.00	0.00	0.00	569	0.00	0.00	isDouble(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocat
0.00	0.00	0.00	473	0.00	0.00	isChar(char)
0.00	0.00	0.00	381	0.00	0.00	Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<
0.00	0.00	0.00	291	0.00	0.00	lerEntrada(std::__cxx11::basic_string<char, std::char_traits<char>, std::alloc
0.00	0.00	0.00	285	0.00	0.00	std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
0.00	0.00	0.00	237	0.00	0.00	Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<
0.00	0.00	0.00	237	0.00	0.00	Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<
0.00	0.00	0.00	190	0.00	0.00	std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
0.00	0.00	0.00	190	0.00	0.00	std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
0.00	0.00	0.00	189	0.00	0.00	Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<
0.00	0.00	0.00	142	0.00	0.00	bool std::operator!=<char, std::char_traits<char>, std::allocator<char> >(std:
0.00	0.00	0.00	95	0.00	0.00	No::~No()
0.00	0.00	0.00	95	0.00	0.00	Pilha<No*>::pop()
0.00	0.00	0.00	95	0.00	0.00	Pilha<No*>::push(No*)
0.00	0.00	0.00	95	0.00	0.00	Pilha<No*>::vazia()
0.00	0.00	0.00	95	0.00	0.00	Pilha<double>::pop()
0.00	0.00	0.00	95	0.00	0.00	Pilha<double>::push(double)
0.00	0.00	0.00	95	0.00	0.00	Pilha<double>::vazia()
0.00	0.00	0.00	95	0.00	0.00	std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>
0.00	0.00	0.00	94	0.00	0.00	precedencia(std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
0.00	0.00	0.00	56	0.00	0.00	std::pair<std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
0.00	0.00	0.00	49	0.00	0.00	std::setprecision(int)
0.00	0.00	0.00	48	0.00	0.00	No::No(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
0.00	0.00	0.00	47	0.00	0.00	No::No(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator
0.00	0.00	0.00	32	0.00	0.00	std::_Rb_tree_iterator<std::pair<std::__cxx11::basic_string<char, std::char_tr
0.00	0.00	0.00	32	0.00	0.00	std::_Rb_tree<std::__cxx11::basic_string<char, std::char_traits<char>, std::al

Dessa vez a função *lerTipoString* foi chamada um total de 570 vezes.

6. Conclusões

Neste trabalho prático, foi desenvolvido um programa em C++ que lida com expressões matemáticas representadas por árvores binárias. O programa é capaz de ler expressões matemáticas infixa e posfixa, convertê-las entre os dois formatos e resolver o valor da expressão.

As principais habilidades desenvolvidas foram a manipulação de árvores binárias e percorrendo-as em pré-ordem, em ordem e pós-ordem e a utilização de pilhas para processar expressões matemáticas

Além disso algumas das lições que podem ser aprendidas incluem a importância de utilizar classes e funções auxiliares para organizar e modularizar o código, utilização

de classes e templates em C++, como usar funções-membro e variáveis-membro em uma classe, tratamento de exceções e erros e leitura e processamento de arquivos de texto.

7. Bibliografia

Documentação da linguagem C++ . Disponível em: <https://cplusplus.com/doc/>.

GNU Project. GNU gprof manual - Chapter 4. Disponível em: <https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_chapter/gprof_4.html#SEC5>.

Microsoft. Learn C++. Disponível em: <https://learn.microsoft.com/en-us/cpp/?view=msvc-170>.

Paulo Feofiloff. DCC-IME-USP. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/bint.html>.

Wikipédia. Notação polonesa inversa. Disponível em: <https://pt.wikipedia.org/wiki/Notacão_polonesa_inversa>.

Wikipédia. Notação infixa. Disponível em: <https://pt.wikipedia.org/wiki/Notação_infixa>.

GeeksforGeeks. Expression Tree. Disponível em: <https://www.geeksforgeeks.org/expression-tree/>.

GeeksforGeeks. Evaluation of Postfix Expression. Disponível em: <https://www.geeksforgeeks.org/evaluation-of-postfix-expression/>.

GeeksforGeeks. Stack Data Structure. Disponível em: <https://www.geeksforgeeks.org/stack-data-structure/>.

GeeksforGeeks. Analysis of Algorithms | Big-O analysis. Disponível em: <https://www.geeksforgeeks.org/analysis-algorithms-big-o-analysis/>.

- Material apresentado em sala e disponível no Portal Minhas Turmas

8. Instruções para compilação e execução

A compilação e a execução do programa são feitas diretamente no terminal:

Compilar e executar: `make run`

Excluir objetos e arquivos executáveis: `make clean`