Trabalho Prático 1
Resolvedor de Expressão Numérica

Pedro Henrique Madeira de Oliveira Pereira

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

pedrohmop@ufmg.br

1 Introdução

Expressões numéricas são um conjunto de uma ou mais operações, que devem ser realizadas em uma determinada ordem para se obter o resultado correto. A forma mais usual de representar uma expressão numérica é chamada representação infixa, onde cada operador aparece entre os números (ou expressões) sobre os quais atua, havendo também o uso de parênteses para representar a precedência das operações. A notação polonesa inversa também é outra forma de representar expressões: também chamada de posfixa, essa representação coloca os operadores depois dos números, eliminando a necessidade dos parênteses.

Diante desse contexto, o problema proposto consiste em desenvolver um resolvedor de expressões infixas ou posfixas, lendo-as e armazenando-as devidamente utilizando as estruturas de dados aprendidas no curso.

2 Método

2.1 Linguagem de Programação

O programa foi desenvolvido utilizando C++.

2.2 Estruturas de Dados

A resolução do problema contou com a utilização de duas das principais estruturas de dados aprendidas: Árvore Binária e Pilha. A Arvore Binária foi utilizada para armazenar a expressão, seja infixa ou posfixa. Basicamente, os números das expressões são as folhas da árvore, enquanto os demais nós contém os operadores que irão atuar sobre os números ou sobre o(s) resultado(s) de outra(s) operação(ões).

Já a Pilha foi utilizada como estrutura de dados auxiliar, sendo extremamente útil para guardar valores no momento do armazenamento da expressão lida na forma de árvore binária, bem como no algoritmo para resolver essa expressão e retornar seu resultado.

2.3 Classes

O programa contém ao todo 4 Classes/Tipos Abstratos de Dados: *BinaryTreeNode*, *BinaryTree*, *Stack* e *Expression*, além de um *namespace string_utils*, que serão detalhados a seguir.

2.3.1 BinaryTreeNode

Contém a especificação de cada nó da árvore binária, e conta com 3 atributos:

- **value**: *string* que representa o valor do nó.
- **left**: ponteiro para outro *BinaryTreeNode*, representando o nó à esquerda na árvore.
- **right**: ponteiro para outro *BinaryTreeNode*, representando o nó à direita na árvore.

2.3.2 BinaryTree

Contém a especificação para a estrutura de dados do tipo Árvore Binária utilizada no algoritmo. Contém apenas 1 atributo:

- ***root***: ponteiro para *BinaryTreeNode* que representa a raiz da árvore.

Os métodos públicos da classe são os seguintes:

- **setRoot**: atribui o ponteiro para nó da árvore recebido ao atributo *root*.
- **isEmpty**: retorna um booleano indicando se a árvore está vazia ou não.
- **insert**: recebe um valor do tipo *string* e insere um nó com esse valor. Pode receber também ponteiros para o nó da esquerda e da direita do nó inserido.
- **preOrder**: retorna uma *string* de acordo com o caminhamento *preOrder* da árvore.
- **inOrderWithParenthesis**: retorna uma *string* de acordo com o caminhamento *inOrder* da árvore, já inserindo os parênteses para ficar como uma expressão infixa.
- **postOrder**: retorna uma *string* de acordo com o caminhamento *postOrder* da árvore.
- **getInOrderStack**: retorna uma pilha com os valores da árvore ordenados pelo encaminhamento *inOrder*.
- **getPostOrderStack**: retorna uma pilha com os valores da árvore ordenados pelo encaminhamento *postOrder*.

2.3.3 Stack

Contém a especificação para a estrutura de dados do tipo Pilha. Foi utilizado o recurso de *Class Templates* do C++ para que a pilha instanciada pudesse conter qualquer tipo de dados, ou seja, é possível criar pilhas de *double* ou *string* com essa mesma classe, por exemplo. Os atributos da classe são os seguintes:

- **top**: inteiro que armazena o índice do elemento do *array* de itens que está no topo da lista.
- **size**: inteiro que armazena o tamanho da lista.
- **MAXTAM**: contém o tamanho máximo do *array* de itens auxiliar.
- **items**: *array* do tipo do *Template* da classe, que armazena os elementos da lista.

É importante ressaltar que as pilhas são geralmente implementadas de duas formas: sequencialmente, através de um *array* auxiliar, e de forma encadeada, com ponteiros. A implementação escolhida foi a sequencial por ser mais intuitiva, fácil de desenvolver e apresentar um leve ganho de performance devido à menor complexidade assintótica de algumas funções.

	Pilha Arranjo	Pilha Encadeada
Construtor	**O(1)**	**O(1)**
Destrutor	**-**	O(n)
Empilha	**O(1)**	**O(1)**
Desempilha	**O(1)**	**O(1)**
Limpa	**O(1)**	O(n)
Tamanho	Fixo	**Dinâmico**
Memória Extra	**Não**	Sim
Implementação Simples	**Sim**	Não

Figura 1: Tabela com a complexidade assintótica para as duas implementações da Pilha

Observe que a única vantagem real em utilizar a implementação encadeada está no tamanho dinâmico da pilha, o que não é um problema de acordo com a especificação do trabalho, visto que o limite de 1000 espaços determinado para o vetor de itens da pilha é mais que necessário para que ela atue como estrutura auxiliar para armazenar expressões de até 1000 caracteres.

Os métodos da classe são os seguintes:

- **push**: empilha um elemento na pilha.
- **pop**: desempilha um elemento da pilha, retornando-o.
- **getTop**: retorna o elemento que está no topo da pilha. Repare que é quase equivalente ao método *pop*, com a diferença de que não remove o elemento do topo.
- **getSize**: retorna o tamanho da pilha.
- **isEmpty**: retorna se a pilha está vazia.

- **clean**: limpa a pilha.
- **print**: imprime o estado atual da pilha.

2.3.4 Expression

Contém a especificação para a estrutura de dados que tem a função de armazenar a expressão em uma árvore, contendo métodos úteis para a manipulação e resolução da mesma. O único atributo da classe é o seguinte:

- **tree**: objeto de *BinaryTree* que contém a expressão.

Os métodos da classe são os seguintes:

- **storeInfixExpression**: recebe uma *string* contendo uma expressão infixa, processa e armazena na forma de árvore binária.
- **storePostfixExpression**: recebe uma *string* contendo uma expressão posfixa, processa e armazena na forma de árvore binária.
- **solve**: resolve a expressão armazenada na árvore, retornando um *double* com o resultado.
- **returnAsInfix**: retorna a expressão armazenada na notação infixa.
- **returnAsPostfix**: retorna a expressão armazenada na notação posfixa.
- **isEmpty**: retorna um booleano indicando se a árvore da expressão está vazia ou não.

2.3.5 string_utils

Namespace contendo funções úteis para verificação de strings, sendo elas:

- **isInstruction**: verifica se a *string* passada é uma instrução (LER, INFIXA, POSFIXA, RESOLVE) e retorna qual delas é.
- **isDouble**: verifica se a *string* passada pode ser convertida para *double*.
- **isOperator**: verifica se a *string* passada é um operador (+, -, *, /).
- **isParenthesis**: verifica se a *string* passada é um parêntese.
- **detectStringType**: verifica se a *string* passada é um *double* ou um operador ou um parêntese, nesse último caso retornando se o parêntese é de abertura ou fechamento.

3 Análise de Complexidade

A seguir será apresentado a análise da complexidade de tempo e espaço das principais funções do programa.

3.1 Stack

Método	Tempo	Espaço
push	$O(1)$	-
pop	$O(1)$	$O(1)$
getTop	$O(1)$	-
getSize	$O(1)$	-
isEmpty	$O(1)$	-
clean	$O(1)$	-
print	$O(n)$	-

3.2 BinaryTree

Método	Tempo	Espaço
setRoot	$O(1)$	-
isEmpty	$O(1)$	-
insert	$O(1)$	$O(1)$
preOrderTraversal	$O(n)$	-
inOrderTraversalWithParenthesis	$O(n)$	-
postOrderTraversal	$O(n)$	-
inOrderStack	$O(n)$	-
postOrderStack	$O(n)$	-
recursiveDelete	$O(n)$	-

3.3 string_utils

Método	Tempo	Espaço
isInstruction	$O(1)$	-
isDouble	$O(1)$	-
isOperator	$O(1)$	-
isParenthesis	$O(1)$	-
detectStringType	$O(1)$	-

3.4 Expression

Método	Tempo	Espaço
storeInfixExpression	$O(n)$	$O(n)$
storePostfixExpression	$O(n)$	$O(n)$
solve	$O(n)$	$O(n)$
returnAsInfix	$O(n)$	-
returnAsPostfix	$O(n)$	-
isEmpty	$O(1)$	-

4 Estratégias de Robustez

Para melhor legibilidade e padronização do código, os nomes de funções e variáveis foram criados em inglês e no formato *camelCase*, havendo o cuidado de deixar os nomes breves

mas ao mesmo tempo descritivos, de forma que se possa entender o que cada função/-variável faz. A formatação do código foi feita através do formatador padrão da extensão *C/C++* do *VSCode*.

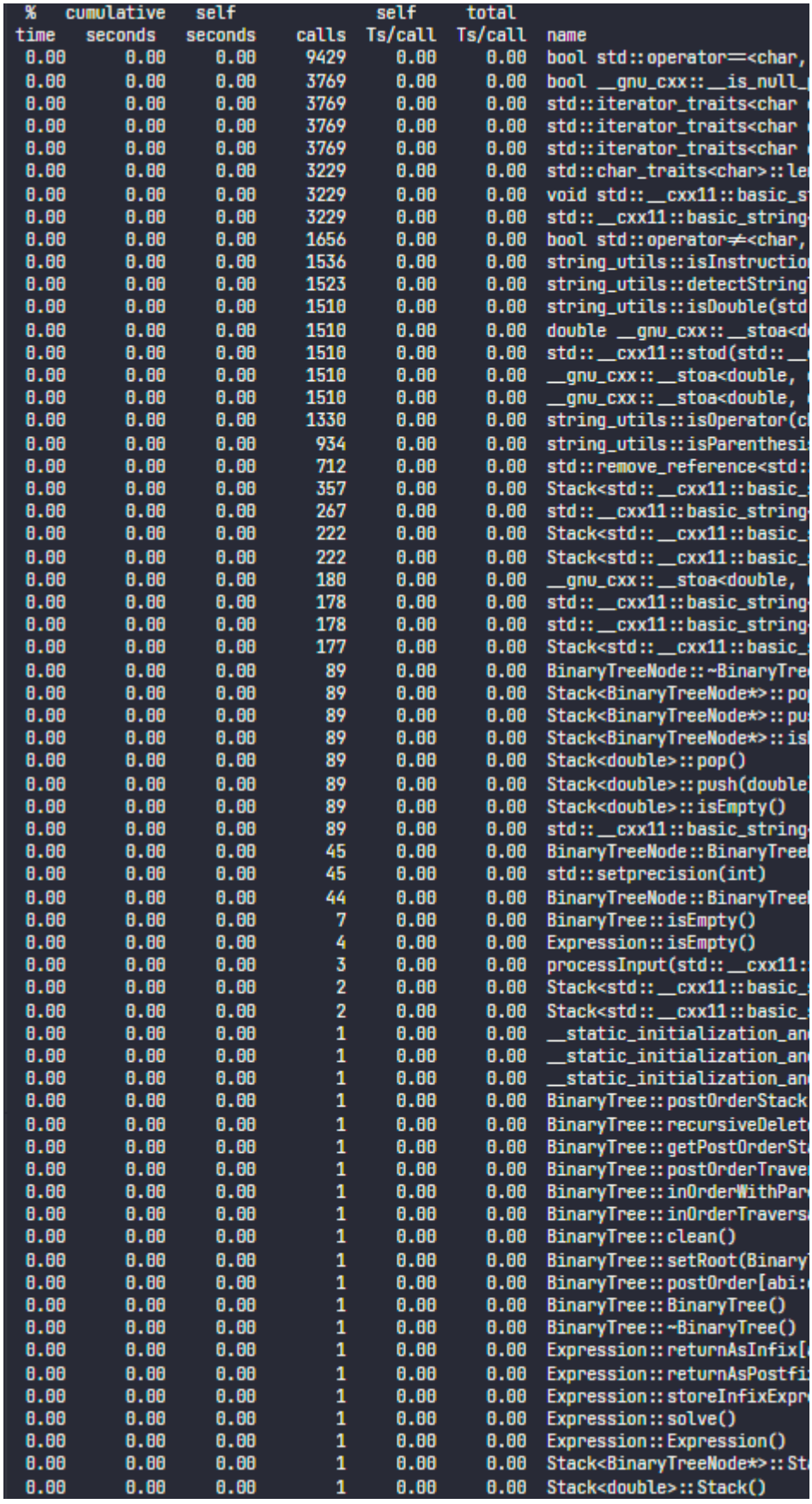
O tratamento de erros foi feito através do lançamento e captura de exceções, em geral do tipo *runtime_error*. Os principais erros tratados consistem na verificação da validade da expressão passada e, na hora da sua resolução, se há alguma divisão por 0. O primeiro foi feito nos métodos *storeInfixExpression* e *storePostfixExpression* da classe *Expression*, onde foi utilizado *try...catch* para capturar algum erro na hora do armazenamento, lançando um *runtime_error* com a descrição de que a expressão é inválida caso esse erro ocorra. Neste caso, os erros de expressão inválida geralmente ocorrem devido à tentativa de desempilhar um item das pilhas auxiliares quando já estão vazias.

Já no método *solve*, há a verificação da divisão por 0 com lançamento de exceção ao se detectar que o segundo operando em uma operação de divisão (ou seja, o divisor) é 0. A captura das exceções e feedback ao usuário é feito na função *processInput* do *main*.

5 Análise Experimental

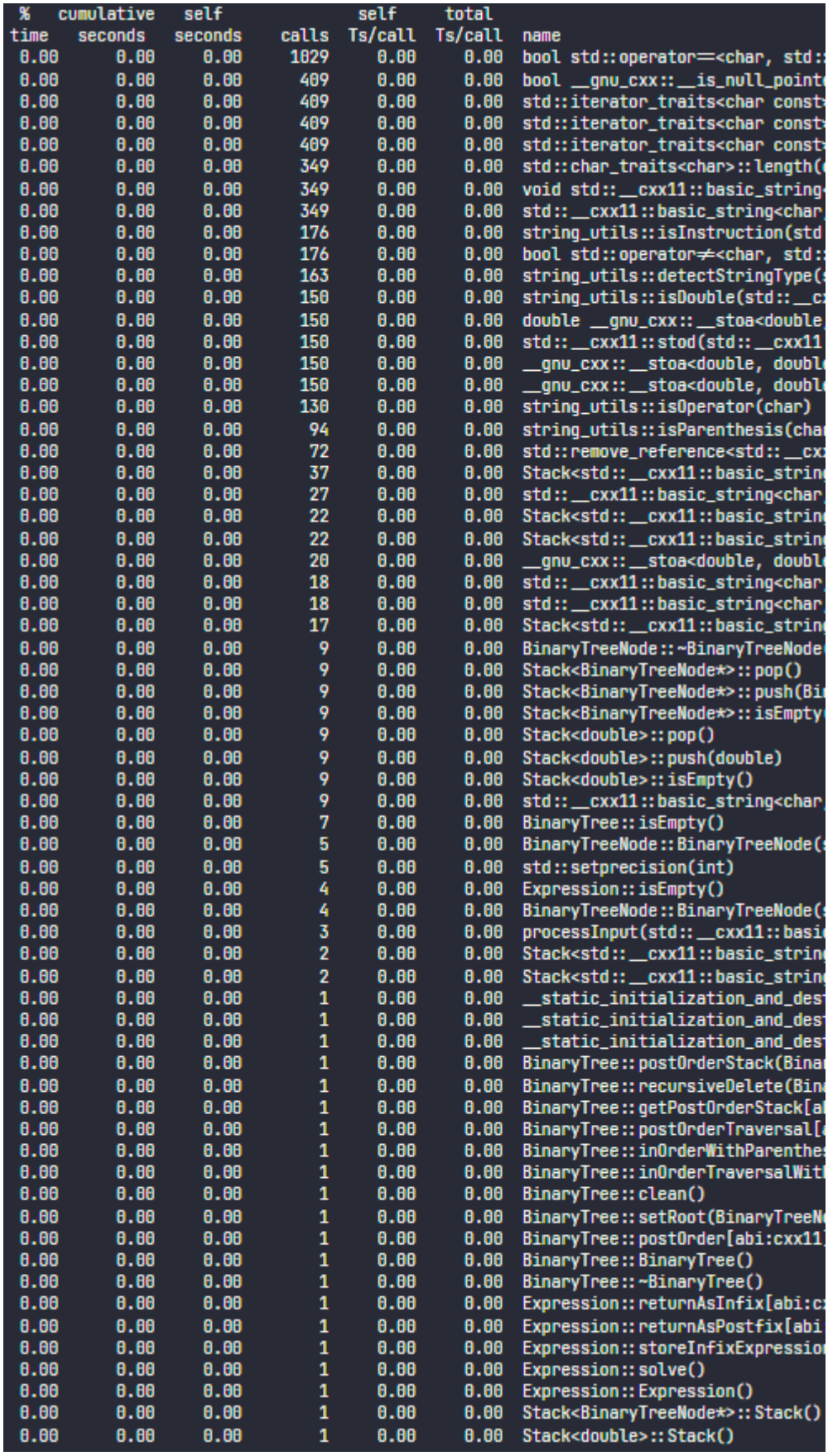
Para os casos de teste disponibilizados pelo professor, o processamento de todas as expressões foi praticamente instantâneo, mesmo com as maiores, de quase 1000 caracteres. Isso ocorre pois a carga máxima de 1000 caracteres é bem pequena sob um ponto de vista computacional, não ocasionando problemas de performance em um programa onde as funções mais custosas tem complexidade linear $O(n)$.

A seguir, estão os testes efetuados com uma expressão grande (aproximadamente 850 caracteres) e uma pequena (aproximadamente 90 caracteres). Observe que, apesar de o tempo de execução ser extremamente pequeno para os dois casos, o de maior expressão apresenta um número consideravelmente maior de chamadas de funções em relação ao teste com expressão menor.



% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	9429	0.00	0.00	bool std::operator==<char,
0.00	0.00	0.00	3769	0.00	0.00	bool __gnu_cxx::__is_null_
0.00	0.00	0.00	3769	0.00	0.00	std::iterator_traits<char
0.00	0.00	0.00	3769	0.00	0.00	std::iterator_traits<char
0.00	0.00	0.00	3769	0.00	0.00	std::iterator_traits<char
0.00	0.00	0.00	3229	0.00	0.00	std::char_traits<char>::le
0.00	0.00	0.00	3229	0.00	0.00	void std::__cxx11::basic_s
0.00	0.00	0.00	3229	0.00	0.00	std::__cxx11::basic_string
0.00	0.00	0.00	1656	0.00	0.00	bool std::operator≠<char,
0.00	0.00	0.00	1536	0.00	0.00	string_utils::isInstructio
0.00	0.00	0.00	1523	0.00	0.00	string_utils::detectString
0.00	0.00	0.00	1510	0.00	0.00	string_utils::isDouble(std
0.00	0.00	0.00	1510	0.00	0.00	double __gnu_cxx::__stoa<d
0.00	0.00	0.00	1510	0.00	0.00	std::__cxx11::stod(std::__
0.00	0.00	0.00	1510	0.00	0.00	__gnu_cxx::__stoa<double,
0.00	0.00	0.00	1510	0.00	0.00	__gnu_cxx::__stoa<double,
0.00	0.00	0.00	1330	0.00	0.00	string_utils::isOperator(c
0.00	0.00	0.00	934	0.00	0.00	string_utils::isParenthesi
0.00	0.00	0.00	712	0.00	0.00	std::remove_reference<std:
0.00	0.00	0.00	357	0.00	0.00	Stack<std::__cxx11::basic_
0.00	0.00	0.00	267	0.00	0.00	std::__cxx11::basic_string
0.00	0.00	0.00	222	0.00	0.00	Stack<std::__cxx11::basic_
0.00	0.00	0.00	222	0.00	0.00	Stack<std::__cxx11::basic_
0.00	0.00	0.00	180	0.00	0.00	__gnu_cxx::__stoa<double,
0.00	0.00	0.00	178	0.00	0.00	std::__cxx11::basic_string
0.00	0.00	0.00	178	0.00	0.00	std::__cxx11::basic_string
0.00	0.00	0.00	177	0.00	0.00	Stack<std::__cxx11::basic_
0.00	0.00	0.00	89	0.00	0.00	BinaryTreeNode::~BinaryTre
0.00	0.00	0.00	89	0.00	0.00	Stack<BinaryTreeNode*>::po
0.00	0.00	0.00	89	0.00	0.00	Stack<BinaryTreeNode*>::pu
0.00	0.00	0.00	89	0.00	0.00	Stack<BinaryTreeNode*>::is
0.00	0.00	0.00	89	0.00	0.00	Stack<double>::pop()
0.00	0.00	0.00	89	0.00	0.00	Stack<double>::push(double
0.00	0.00	0.00	89	0.00	0.00	Stack<double>::isEmpty()
0.00	0.00	0.00	89	0.00	0.00	std::__cxx11::basic_string
0.00	0.00	0.00	45	0.00	0.00	BinaryTreeNode::BinaryTree
0.00	0.00	0.00	45	0.00	0.00	std::setprecision(int)
0.00	0.00	0.00	44	0.00	0.00	BinaryTreeNode::BinaryTree
0.00	0.00	0.00	7	0.00	0.00	BinaryTree::isEmpty()
0.00	0.00	0.00	4	0.00	0.00	Expression::isEmpty()
0.00	0.00	0.00	3	0.00	0.00	processInput(std::__cxx11:
0.00	0.00	0.00	2	0.00	0.00	Stack<std::__cxx11::basic_
0.00	0.00	0.00	2	0.00	0.00	Stack<std::__cxx11::basic_
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_an
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_an
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_an
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::postOrderStack
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::recursiveDelet
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::getPostOrderSt
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::postOrderTrave
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::inOrderWithPar
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::inOrderTravers
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::clean()
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::setRoot(Binary
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::postOrder[abi:
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::BinaryTree()
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::~BinaryTree()
0.00	0.00	0.00	1	0.00	0.00	Expression::returnAsInfix[
0.00	0.00	0.00	1	0.00	0.00	Expression::returnAsPostfi
0.00	0.00	0.00	1	0.00	0.00	Expression::storeInfixExpr
0.00	0.00	0.00	1	0.00	0.00	Expression::solve()
0.00	0.00	0.00	1	0.00	0.00	Expression::Expression()
0.00	0.00	0.00	1	0.00	0.00	Stack<BinaryTreeNode*>::St
0.00	0.00	0.00	1	0.00	0.00	Stack<double>::Stack()

Figura 2: Tempo gasto e chamadas das funções para o teste com expressão grande (850 caracteres)



% time	cumulative seconds	self seconds	calls	self Ts/call	total Ts/call	name
0.00	0.00	0.00	1029	0.00	0.00	bool std::operator==<char, std::
0.00	0.00	0.00	409	0.00	0.00	bool __gnu_cxx::__is_null_point
0.00	0.00	0.00	409	0.00	0.00	std::iterator_traits<char const
0.00	0.00	0.00	409	0.00	0.00	std::iterator_traits<char const
0.00	0.00	0.00	409	0.00	0.00	std::iterator_traits<char const
0.00	0.00	0.00	349	0.00	0.00	std::char_traits<char>::length(
0.00	0.00	0.00	349	0.00	0.00	void std::__cxx11::basic_string
0.00	0.00	0.00	349	0.00	0.00	std::__cxx11::basic_string<char
0.00	0.00	0.00	176	0.00	0.00	string_utils::isInstruction(std
0.00	0.00	0.00	176	0.00	0.00	bool std::operator≠<char, std::
0.00	0.00	0.00	163	0.00	0.00	string_utils::detectStringType(
0.00	0.00	0.00	150	0.00	0.00	string_utils::isDouble(std::__c
0.00	0.00	0.00	150	0.00	0.00	double __gnu_cxx::__stoa<double
0.00	0.00	0.00	150	0.00	0.00	std::__cxx11::stod(std::__cxx11
0.00	0.00	0.00	150	0.00	0.00	__gnu_cxx::__stoa<double, doubl
0.00	0.00	0.00	150	0.00	0.00	__gnu_cxx::__stoa<double, doubl
0.00	0.00	0.00	130	0.00	0.00	string_utils::isOperator(char)
0.00	0.00	0.00	94	0.00	0.00	string_utils::isParenthesis(cha
0.00	0.00	0.00	72	0.00	0.00	std::remove_reference<std::__cx
0.00	0.00	0.00	37	0.00	0.00	Stack<std::__cxx11::basic_strin
0.00	0.00	0.00	27	0.00	0.00	std::__cxx11::basic_string<char
0.00	0.00	0.00	22	0.00	0.00	Stack<std::__cxx11::basic_strin
0.00	0.00	0.00	22	0.00	0.00	Stack<std::__cxx11::basic_strin
0.00	0.00	0.00	20	0.00	0.00	__gnu_cxx::__stoa<double, doubl
0.00	0.00	0.00	18	0.00	0.00	std::__cxx11::basic_string<char
0.00	0.00	0.00	18	0.00	0.00	std::__cxx11::basic_string<char
0.00	0.00	0.00	17	0.00	0.00	Stack<std::__cxx11::basic_strin
0.00	0.00	0.00	9	0.00	0.00	BinaryTreeNode::~BinaryTreeNode
0.00	0.00	0.00	9	0.00	0.00	Stack<BinaryTreeNode*>::pop()
0.00	0.00	0.00	9	0.00	0.00	Stack<BinaryTreeNode*>::push(Bi
0.00	0.00	0.00	9	0.00	0.00	Stack<BinaryTreeNode*>::isEmpty
0.00	0.00	0.00	9	0.00	0.00	Stack<double>::pop()
0.00	0.00	0.00	9	0.00	0.00	Stack<double>::push(double)
0.00	0.00	0.00	9	0.00	0.00	Stack<double>::isEmpty()
0.00	0.00	0.00	9	0.00	0.00	std::__cxx11::basic_string<char
0.00	0.00	0.00	7	0.00	0.00	BinaryTree::isEmpty()
0.00	0.00	0.00	5	0.00	0.00	BinaryTreeNode::BinaryTreeNode(
0.00	0.00	0.00	5	0.00	0.00	std::setprecision(int)
0.00	0.00	0.00	4	0.00	0.00	Expression::isEmpty()
0.00	0.00	0.00	4	0.00	0.00	BinaryTreeNode::BinaryTreeNode(
0.00	0.00	0.00	3	0.00	0.00	processInput(std::__cxx11::basi
0.00	0.00	0.00	2	0.00	0.00	Stack<std::__cxx11::basic_strin
0.00	0.00	0.00	2	0.00	0.00	Stack<std::__cxx11::basic_strin
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_and_des
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_and_des
0.00	0.00	0.00	1	0.00	0.00	__static_initialization_and_des
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::postOrderStack(Bina
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::recursiveDelete(Bin
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::getPostOrderStack[a
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::postOrderTraversal[
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::inOrderWithParenthe
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::inOrderTraversalWit
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::clean()
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::setRoot(BinaryTreeN
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::postOrder[abi:cxx11
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::BinaryTree()
0.00	0.00	0.00	1	0.00	0.00	BinaryTree::~BinaryTree()
0.00	0.00	0.00	1	0.00	0.00	Expression::returnAsInfix[abi:c
0.00	0.00	0.00	1	0.00	0.00	Expression::returnAsPostfix[abi
0.00	0.00	0.00	1	0.00	0.00	Expression::storeInfixExpressio
0.00	0.00	0.00	1	0.00	0.00	Expression::solve()
0.00	0.00	0.00	1	0.00	0.00	Expression::Expression()
0.00	0.00	0.00	1	0.00	0.00	Stack<BinaryTreeNode*>::Stack()
0.00	0.00	0.00	1	0.00	0.00	Stack<double>::Stack()

Figura 3: Tempo gasto e chamadas das funções para o teste com expressão pequena (90 caracteres)

6 Conclusão

Após o desenvolvimento do programa para a resolução do problema proposto, bem como o desenvolvimento dos algoritmos necessários para a manipulação, armazenamento e resolução de expressões numéricas na notação infixa e posfixa, conclui-se que o presente trabalho fora de suma importância para o aprimoramento do conhecimento a respeito de estruturas de dados, levantando a discussão de como elas podem ser usadas para representar sistematicamente certos aspectos da vida cotidiana.

Ainda que a análise e a execução das funções não tenha gerado casos onde uma grande quantidade de recursos computacionais é usada, já que o escopo do problema não contempla um número de operações computacionalmente grande, tal observação se mostrou significante para aprimorar a visão sistemática das funções, através do exercício da visualização de sua complexidade temporal e espacial.

Portanto, o resultado deste trabalho se mostrou como o aperfeiçoamento das habilidades em programação e desenvolvimento de soluções, da capacidade de visualizar problemas de maneira sistemática, abstraindo sua essência para estruturas algorítmicas, e da aptidão em levar em consideração o custo computacional que certas funções trazem ao programa.

7 Referências

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.

8 Instruções para Execução

Para utilizar o programa, descompacte o arquivo .zip e acesse o diretório raiz com seu terminal de preferência. Logo em seguida, utilize o comando `make run` para compilar e executar o programa.