RELATÓRIO DO TRABALHO PRÁTICO 1: RESOLVEDOR DE EXPRESSÃO NUMÉRICA

Diogo Tuler Chaves - 2022043663

1. Introdução

Essa documentação lida com o problema de implementar um resolvedor de expressão númerica. Esse resolvedor lida com dois tipos de representações, a usual(também chamada de infixa) e a notação polonesa inversa (também chamada de pós-fixa), tendo que ser capaz de ler qualquer um dos dois tipos de entradas, realizar sua validação e armazená-los. Após o armazenamento, o programa tem que ser capaz de retornar, quando pedido, ambas as expressões e o resultado.

Para a implementação de um programa com essas funcionalidades foram usados dois tipos abstratos de dados diferentes, a "Pilha" e a "Árvore". A Pilha foi implementada como um "template" de classe para que pudesse ser capaz de lidar com diferentes tipos de dados, sendo muito útil para a validação e para o armazenamento de ambos os tipos de expressão em Árvore. Já a Árvore foi implementada para armazenar as expressões, uma de cada vez, sendo seus "Nós"(elementos da Árvore) formados de "strings" de C++, equivalente a "const char*",contendo um número ou um operador.Todos os detalhes sobre a manipulação desses TAD para a resolução do problema e sua implementação estão na seção 2 deste relatório.

Vale ressaltar que esse trabalho foi realizado em C++ e compilado com g++ em um "Lenovo Lenovo IdeaPad S145-15 IWL" com o processador "Intel® Core™ i5-8265 U CPU @ 1.60GHz × 8" com "NV 118 / Mesa Intel® UHD Graphics 620 (WHL GT2)" e com um sistema operacional "Ubuntu 22.04.2 LTS 64-bit"

2. Métodos

2.1. Organização

- **bin:** Contém o arquivo executável "main", resultado da compilação dos arquivos .o da pasta obj pelo G++.
- **include:** Contém os arquivos ".hpp" com as declarações de funções modularizadas ou de Tipos Abstratos de Dados. Além disso, alguns arquivos acabam por conter implementações, fato que será detalhado abaixo:
    - **No.hpp:** É o arquivo onde é definida apenas a classe No.
        - **Class No:** Classe usada para representar um Nó da Árvore, seus atributos são:
            - **No()**: Construtor da classe "No".
            - **string Elemento:** Cada "No" contém uma string que representa um número ou um operador.
            - **No* Direita,Esquerda:** Sendo uma árvore binária balanceada para cada "No" (menos as folhas) vão se conectar com dois outros, representados por esses ponteiros Direita e Esquerda.
    - **Pilha.hpp:** É o arquivo onde é definida e implementada a classe Elemento_da_Pilha, a classe Pilha(ambas usando template) e o struct Pilha_vazia, além de ser onde é definida a função Montar_Pilha_de_String.

        - **(template<class T>) Class Elemento_da_Pilha:** Classe que representa um elemento do tipo T da lista encadeada na memória.Foi usado um "template" pois a classe “Pilha” também é um.Pelo fato de ser um template toda a classe teve que ser implementada nesse arquivo “.hpp”. Os atributos dessa classe são:
            - **Elemento_da_Pilha( ):**Construtor da classe Elemento_da_Pilha.
            - **T Elemento:** Os elementos do tipo T que vão ser enfileirados.
            - **Elemento_da_Pilha<T>* Proximo:** Um ponteiro para o próximo elemento da pilha.
        - **(template<class T>) Pilha:**Classe que representa uma pilha de elementos do tipo T que, nesse código podem ser "No* e "string". Foi usado um "template" para que não fosse necessário implementar duas pilhas de elementos diferentes que possuem as mesmas funções. Pelo fato de ser um template toda a classe teve que ser implementada nesse arquivo “.hpp”. Os atributos dessa classe são:
            - **Pilha( ):** Construtor da classe Pilha.
            - **bool Pilha_Vazia( ):** Função que informa se a Pilha está vazia.
            - **void Empilha(T valor):** Função que adiciona elementos do tipo T no topo da pilha.
            - **T Desempilha( ):** Função que retorna o elemento do tipo T que está no topo da pilha e o apaga da pilha
            - **~Pilha( ):** Destrutor da classe Pilha recursivo.
            - **Elemento_da_Pilha<T> * Topo:** É o topo da pilha, sendo um elemento do tipo Elemento_da_Pilha<T>.
        - **struct Pilha_vazia:** Será abordado na parte 4 deste relatório sendo uma exceção usada nos mecanismos de programação defensiva implementados.
        - **void Montar_Pilha_De_String(string Entrada, Pilha<string>* Retorno):**Função para montar uma pilha a partir de uma string
    - **Arvore.hpp:** É o arquivo onde é definida a classe Arvore.
        - **Class Arvore**: Uma class que representa uma árvore no qual cada nó é do tipo No e representa ou um número da expressão(se for uma folha) ou um operador no formato de uma string. Os atributos dessa classe são:
            - **Arvore ( )**: Construtor da classe Arvore.
            - **bool Arvore_Vazia()**: Informa se a árvore está vazia.
            - **bool Montar_Arvore(string Entrada,bool Infixa)**:Função que monta a árvore recebendo como entrada o tipo de expressão e uma string com a expressão. Retorna se foi um sucesso ou não a montagem. Essa função chama ou a função Arvore::Montar_Arvore_Pela_Entrada_Infixa ou a função Arvore::Montar_Arvore_Pela_Entrada_Posfixa.
            - **string Retorna_Infixa():** Função para retornar a forma infixa.
            - **string Retorna_Posfixa()**: Função para retornar a forma pós-fixa.
            - **double Retorna_Resultado():** Função para retornar o resultado.
            - **bool Limpar_Arvore():** Função para limpar a árvore que chama a função Limpa Recursivo.
            - **~Arvore():** Destrutor da classe Árvore.

- **void Limpa_Recursivo(No** p):** Função Recursiva para fazer a limpeza da árvore.
- **string Caminha_In_Ordem_Adaptada(No* p):** Função que caminha na árvore da forma "in ordem" para retornar a forma infixa.
- **string Caminha_Pos_Ordem_Adaptada(No* p):** Função que caminha na árvore da forma "pós ordem" para retornar a forma pós-fixa.
- **No* Montar_Arvore_Pela_Entrada_Infixa(char** Entrada):** Monta a árvore, recebendo um char** como entrada.Ela faz essa montagem a partir de uma manipulação de Pilhas de No* e recursão.
- **void Montar_Arvore_Pela_Entrada_Posfixa (Pilha<string> * Entrada,No** p):** Monta a árvore, recebendo uma Pilha* de string e um ponteiro para o ponteiro que representa a raiz como entrada. Ela faz essa montagem desempilhando a pilha recebida como argumento de forma estratégica e recursiva.
- **double Resultado_Recursivo(No* p):** Usa uma função recursiva simples para resolver a expressão através das strings armazenadas na árvore.
- **No* Raiz:** É o Nó inicial, a raiz da árvore binária.

○ **Leitura_Entrada.hpp:** Arquivo com funções auxiliares para a leitura e análise da entrada.

■ **structs Muitos_Argumentos & Poucos_Argumentos & Arquivo_Invalido :** Serão abordados na parte 4 deste relatório sendo exceções usadas nos mecanismos de programação defensiva implementados.
■ **FILE* Abre_Arquivo(int argc,char ** argv):** Função para abrir o arquivo lido na linha de comando enviando exceções quando necessário.
■ **string Retorna_Funcao(string* Entrada, int *i):** Função que retorna o que deve ser feito("LER" ou "POSFIXA" ou "INFIXA" ou "RESOLVA").
■ **int Conta_Tamanho_Entrada(string Expressao):** Função que retorna o tamanho da entrada pois se for maior que 1000 é inválido.
■ **bool Expressao_Valida(string Tipo_da_Expressao,string Expressao):** Função que retorna se a expressão é válida ou não.
■ **bool Armazenar(Arvore* Armazenamento,string Expressao,string Tipo_da_Expressao):** Função que armazena a expressão em uma árvore chamando a função Arvore::Montar_Arvore.
■ **bool Ler_Expressao(int i,string Entrada,Arvore* Armazenamento):** Função para a leitura de uma expressão chamando a função "Armazenar" para armazená-la.
■ **void Retornar_Posfixa(bool Expressao_Valida,Arvore* Armazenamento):** Função para retornar a forma "POSFIXA" chamando a função Arvore::Retorna_Posfixa.
■ **void Retornar_Infixa(bool Expressao_Valida,Arvore* Armazenamento):** Função para retornar a forma "INFIXA" chamando a função Arvore::Retorna_Infixa.
■ **void Retornar_Resultado(bool Expressao_Valida,Arvore* Armazenamento):** Função para retornar o resultado chamando a função Arvore::Retorna_Resultado.

- **obj**: Contém os arquivos .cpp da pasta src traduzidos para linguagem de máquina pelo compilador GCC e com a extensão .o (object).
- **src**: Contém as implementações dos arquivos .hpp da pasta “include” em arquivos .cpp, além disso contém a função main que utiliza as funções de Leitura_Entrada.hpp para organizar a entrada e a saída de dados. Nestes arquivos estão programadas toda a lógica e a robustez contra erros do programa.

2.2. Funcionamento

Após apresentar todas as funções e TADs implementados é válido ressaltar como que o código funciona e cumpre seu propósito. Primeiramente, é passado como argumento na entrada um arquivo para a leitura(make run <nome_do_arquivo>) e a função “main” chama a função “Abre_Arquivo” para abri-lo. Em caso de sucesso a função “main” vai começar a ler linha por linha e fazer o que está sendo pedido em cada uma. Para isso utiliza a função
função Retorna_Funcao para verificar o que tem que ser feito, tendo 4 opções possíveis:

- “LER <TIPO DE EXPRESSÃO> <EXPRESSÃO>” : Ao optar pela operação "ler", a "main" aciona a função "Ler_Expressao". Essa função se inicia com a limpeza da árvore, caso já haja uma expressão armazenada. Em seguida, ela analisa o tipo da expressão especificado na entrada e invoca a função "Expressao_Valida" para verificar a validade da mesma. Se a expressão for válida, a função "Armazenar" é utilizada para chamar a função "Arvore::Montar_Arvore". Esta, por sua vez, invoca a função "Arvore::Montar_Arvore_Pela_Entrada_Posfixa" ou a função "Arvore::Montar_Arvore_Pela_Entrada_Infixa", dependendo do tipo de entrada fornecida. Ambas as funções utilizam pilhas, de formas diferentes, para armazenar a expressão em uma árvore. Essa árvore vai conter em cada nó um operador ou, caso esse nó seja uma folha, um número, ambos em formato de string, sendo utilizado uma conversão para double na hora de resolver a expressão. A árvore com a expressão é construída na variável "Armazenamento" na “main” e pode ser utilizada para retornar a expressão na forma pós-fixa e infixa e o resultado quantas vezes forem necessárias.

- “POSFIXA”: Ao ser escolhido a opção pós-fixa, a “main” chama a função “Retornar_Posfixa” que analisa se existe uma expressão armazenada na árvore e, caso tiver, chama a função “Arvore::Retorna_Posfixa” que,por sua vez, chama a função “Arvore::Caminha_Pos_Ordem_Adaptada” para, através de um caminhamento pós-ordem, retornar a expressão pós-fixa.

- “INFIXA”:Ao ser escolhido a opção infixa, a “main” chama a função “Retornar_Infixa” que analisa se existe uma expressão armazenada na árvore e, caso tiver, chama a função “Arvore::Retorna_Infixa” que,por sua vez, chama a função “Arvore::Caminha_In_Ordem_Adaptada” para, através de um caminhamento in-ordem, retornar a expressão infixa.

- “RESOLVE”: o ser escolhido a opção resolve, a “main” chama a função “Retornar_Resultado” que analisa se existe uma expressão armazenada na árvore e, caso tiver, chama a função “Arvore::Retorna_Resultado” que,por sua vez, chama a função “Arvore::Resultado_Recursivo()” para, através da recursividade, retornar a expressão infixa.

3. Análise de Complexidade

Muitas funções servem apenas para uma maior organização do código e sua complexidade depende totalmente das funções chamadas internamente por elas. Tendo isso em vista, aqui está uma análise de complexidade das funções principais do programa:

- **Pilha .hpp/.cpp**
  - **void Empilha(T valor) & T Desempilha( ):** Ambas não possuem laços nem recursividade, possuindo o custo de tempo igual a O(1). Em relação ao custo de espaço ambas utilizam apenas um Elemento_da_Pilha<T>* Auxiliar, fazendo sua função theta da complexidade de espaço utilizado ser constante (Θ(1)) independente do tamanho da entrada.
  - **~Pilha( ):** Sendo a variável 'k' o tamanho da fila no final do escopo, essa função chama a função "T Desempilha( )" k vezes, logo sua complexidade de tempo é Θ(k) e a função theta da sua complexidade de espaço é dada pelo tamanho da "Elemento_da_Pilha<T>* Auxiliar" multiplicado por k.
  - **void Montar_Pilha_De_String(string Entrada, Pilha<string>* Retorno):**Sendo a variável 'c' a quantidade de char fornecidos na entrada, essa função simplesmente percorre uma vez todos eles, separando por espaço e armazenado na pilha possuindo o custo de tempo igual a Θ(c). A função de complexidade da memória é igual a Θ(c), pois são armazenadas strings em uma pilha na qual a soma dos tamanhos das strings é c.
- **Arvore .hpp/.cpp**
  - **void Limpa_Recursivo(No** p)**: Sendo a variável 'n' a quantidade de números somados a quantidade de operadores armazenados nos nós da árvore, essa função percorre de forma recursiva esses nós, visitando cada um uma vez, logo sua complexidade de tempo é dada por Θ(n). Essa função utiliza uma variável No* Auxiliar n vezes, logo seu tamanho multiplicado por n é a função theta da complexidade de espaço utilizado.
  - **string Caminha_In_Ordem_Adaptada(No* p) & string Caminha_Pos_Ordem_Adaptada(No* p) :** Ambas, apesar de não possuírem laços, utilizam da recursividade para caminhar sobre todos os operadores e números da árvore para retornarem à expressão. Como eles visitam todos os nós apenas uma vez sua complexidade, sendo a variável 'n' a quantidade de números somados a quantidade de operadores, pode ser dada por Θ(n). Em relação à complexidade de espaço, ambas as funções utilizam apenas duas variáveis do tipo "string" chamadas de auxiliar e de retorno. A Auxiliar nunca vai armazenar mais do que um número ou operador, enquanto a retorno vai armazenar todos os números e operadores, sendo uma espécie de concatenação desses n valores.Logo, a soma dos tamanhos dos n valores é a função theta da complexidade de espaço utilizado.
  - **No* Montar_Arvore_Pela_Entrada_Infixa(char** Entrada):** Essa função de forma recursiva desempilha uma pilha com o tamanho 'n', sendo a variável 'n' a quantidade de números somados a quantidade de operadores, logo teria uma complexidade de Θ(n). Todavia, essa função chama a função "void Montar_Pilha_De_String(string Entrada, Pilha<string>* Retorno)" que possui a

complexidade de tempo dada por Θ(c) e, já que c>=n, temos O(c) como o custo. Em relação ao custo de espaço também é idêntico a da função "void Montar_Pilha_De_String(string Entrada, Pilha<string>* Retorno)" pois a pilha que é enviada para essa função a única variável de tamanho variável, logo a função de complexidade da memória é igual a Θ(c).

- **void Montar_Arvore_Pela_Entrada_Posfixa (Pilha<string> * Entrada,No** p):** Essa função de forma recursiva desempilha uma pilha com o tamanho 'n', sendo a variável 'n' a quantidade de números somados a quantidade de operadores somados a quantidade de operadores em formatos de nó, e vai armazenando subárvores em outra pilha. Depois disso, essa pilha é desempilhada e seus nós são armazenados na pilha inicial(que a princípio está vazia) e, por fim, a pilha inicial é desempilhada novamente retornando apenas um Nó, que no caso é a raiz. Logo, são feitas no máximo 3n operações que daria uma complexidade de O(3n). Todavia, essa função chama a função "void Montar_Pilha_De_String(string Entrada, Pilha<string>* Retorno)" que possui a complexidade de tempo dada por Θ(c), não sabemos se c>3n logo temos max(O(3n),O(c)) como o custo. Em relação ao custo de espaço é idêntico a da função "void Montar_Pilha_De_String(string Entrada, Pilha<string>* Retorno)" pois a pilha que é enviada para essa função e a outra pilha derivada dessa são as únicas variáveis de tamanho variável, logo a função de complexidade da memória é igual a Θ(c).

- **Leitura_Entrada .hpp/.cpp**
  - **int Conta_Tamanho_Entrada(string Expressao):** Sendo a variável 'c' a quantidade de char fornecidos na entrada, essa função simplesmente percorre uma vez todos eles, possuindo o custo de tempo igual a Θ(c). Em relação ao custo de espaço é constante(Θ(1)), pois independente do tamanho utiliza apenas uma variável que é de fato o contador que vai ser retornado.
  - **bool Expressao_Valida(string Tipo_da_Expressao,string Expressao):** Sendo a variável 'c' a quantidade de char na expressão de entrada essa função percorre, no máximo todos eles, possuindo o custo de tempo igual a O(c). Nesse caso é no máximo pois caso seja encontrado uma invalidez ao longo da análise a função retorna falso antes de acabar o loop.Em relação a memória armazenada a única variável não constante é a Pilha<string> Auxiliar que possui como tamanho a quantidade de números somados a quantidade de operadores (sendo eles válidos ou não) somado a quantidade de parênteses (no caso das infixas) que vamos chamar de 'n'. O custo de espaço pode ser dado por Θ(n).

4. Estratégias de Robustez

4.1. Decisões de projeto

Para o funcionamento efetivo do código foram desenvolvidos diversos mecanismos de defesa que visam a programação defensiva, deixando o código mais robusto e impedindo sua falha em diversas situações. Antes de falar sobre esses mecanismos, é importante salientar sobre algumas decisões de projeto que impactaram diretamente a forma com que esses mecanismos foram implementados:

- O código recebe um arquivo passado pelo terminal com os comandos a serem feitos, sendo finalizado quando chega ao final do arquivo.
- Ao final de cada linha do arquivo é necessário um '\n' (enter), caso contrário a expressão será dada como inválida.
- Todos os elementos diferentes da expressão tem que estar separados por espaços, exemplos:
    - ((3.703479) / (6.146766)) -> essa expressão vai ser considerada válida pois atende os requisitos de uma expressão infixa válida e tem seus elementos divididos por espaços.
    - ((3.703479) / (6.146766)) -> essa expressão vai ser considerada inválida pois apesar de atender os requisitos de uma expressão infixa válida e não tem todos seus elementos divididos por espaços.
- O programa foi feito para conseguir aceitar números negativos sendo necessário apenas colocar o sinal '-' junto ao número sem espaços, exemplo:
    - -10 5 + -> essa expressão posfixa vai ser válida contendo os elementos '-10','5' e '+'
- O programa foi implementado para conseguir funcionar com ou sem parênteses para expressões infixas. Foi implementado toda uma lógica de análise para tais expressões e, contanto que sejam válidas matematicamente, vão ser armazenadas na árvore de forma correta respeitando a precedência de operadores.
- Não são aceitas expressões com mais de 1020 caracteres. Na documentação do trabalho a limitação é 1000, mas foi optado por colocar 20 a mais para evitar falhas.
- Foi adotado que ter uma divisão por '0' não torna a expressão inválida, todavia o programa possui uma programação defensiva que impede que a divisão seja feita durante o processo, retornando erro apenas para o resultado.
- Foi adotado que entradas contendo somente um número ou somente um operador (entre parênteses ou não) não são válidas.
- Foi adotado que entradas contendo '()' não são válidas.

4.2. Mecanismos defensivos e de tolerância a falhas

Tendo em vista as decisões de projeto e as possíveis falhas, foram usados bloco try/catch nas partes sensíveis do código e diferentes estruturas de dados que representavam os diferentes erros. Elas são:

- **Struct Pilha_vazia =** Struct para enviar um erro no try catch quando temos uma pilha vazia e queremos desempilhar. Usado todas as vezes que alguma pilha foi desempilhada sem conferir se esta está vazia.
- **struct Muitos_Argumentos =** Struct para enviar um erro no try catch quando temos um argv com mais argumentos que o necessário. Usado na função Abre_Arquivo.
- **struct Poucos_Argumentos =** Struct para enviar um erro no try catch quando temos um argv sem os argumentos necessários. Usado na função Abre_Arquivo.
- **struct Arquivo_Invalido =** Struct para enviar um erro no try catch quando temos como entrada um nome de arquivo inválido. Usado na função Abre_Arquivo
- **struct Erro_de_Montagem =** Struct para enviar um erro no try catch quando temos algum problema e não conseguimos criar alguma pilha no meio das funções da árvore. Usado em funções dentro da árvore que precisam de pilhas.
- **struct Divisão_Por_Zero =** Struct para enviar um erro no try catch quando temos divisão por 0. Usado na função de resolver a expressão.

Sobre a análise da entrada é válido salientar sobre o comportamento do código. Se uma linha for vazia ou conter uma operação diferente de “LER <TIPO>”,“POSFIXA”, “INFIXA” ou “RESOLVE”, foi implementado que o retorno vai ser uma mensagem dizendo que essa operação é inválida, sendo impresso a operação escrita. No caso da linha vazia vai ser impresso que a operação “ ” é inválida. Além disso, se o <TIPO> do ler não for “POSFIXA” ou “INFIXA” o código vai retornar que a expressão não é válida.

Além dessas estratégias foram implementados pequenos ‘if’ dentro de algumas funções para evitar possíveis erros.

5. Análise Experimental

5.1. Análise de Tempo

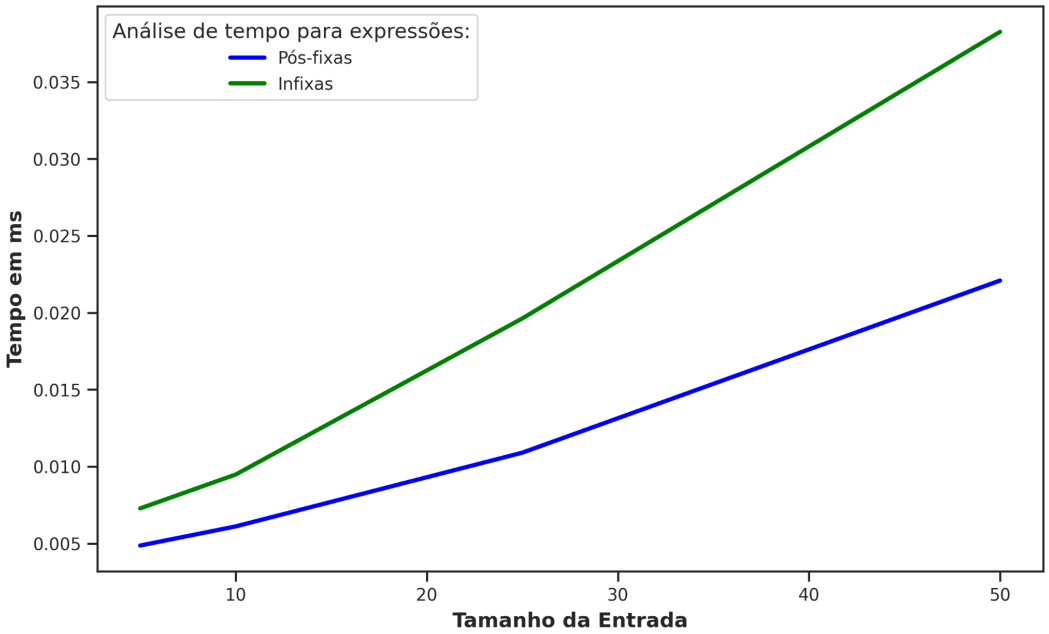


Gráfico do tempo de execução do programa para expressões pós-fixas e infixas.

Para obtermos uma análise mais precisa do tempo gasto pelo algoritmo, utilizamos o tempo de relógio para criar um arquivo CSV que contém informações sobre o tamanho da entrada de teste (fornecidos no Moodle), o tipo de expressão e uma média do tempo gasto para a leitura da expressão, sua conversão para o outro tipo e para a sua resolução em cada tamanho de entrada. Dessa forma, conseguimos uma medição mais acurada do desempenho do algoritmo em diferentes cenários de entrada. Após a criação do CSV foi implementado um código em Python para a criação do gráfico acima.

O gráfico apresentado é linear tanto para as entradas pós-fixas quanto para as entradas infixas. Inegavelmente, isso está de acordo com as análises de complexidade feitas para a função, tendo em vista que nenhuma delas ultrapassa O(n), sendo n o tamanho da entrada. Tendo isso em vista, podemos afirmar que, comprovadamente, o programa possui uma função de complexidade de tempo igual a O(n) independente da expressão.

5.2. Análise de Resultado

Para conseguir analisar a acurácia do programa em ler, armazenar, converter e resolver as expressões foram feitos testes ao longo de todo o desenvolvimento. Além disso, todas as medidas de programação defensiva foram testadas para uma maior certeza sobre a resistência a falhas do programa. Ao longo dos testes várias medidas de proteção tiveram que ser alteradas, seja por apresentar falhas em, ou seja por não indicar a falha e permitir um erro na execução.

Por fim, todos os exemplos postados foram testados e validados.Além disso foram testados, no caso da expressão infixa,expressões encontradas em livros de matemática básica, tendo em vista que nos exemplos não existem expressões sem os parênteses e com número negativos sendo que o código foi adaptado para funcionar nesses casos. Como o código será testado não será incluído capturas de telas contendo esses resultados, tendo em vista que os testes irão demonstrar isso.

6. Conclusões

Em conclusão, no trabalho foi desenvolvido um programa de armazenamento, conversão e resolução de expressões do tipo pós-fixa e infixa, tendo sido utilizados TADs que foram de extrema importância para a resolução dessa tarefa.

No dia a dia do programador TADs são utilizados a todos momentos, seja para códigos simples ou até mesmo grandes sistemas. A maioria das linguagens, inclusive C++, já possui a maioria deles implementados, logo cabe ao programador apenas utilizá-los da maneira mais adequada para o momento. Esse trabalho foi muito importante, pois fez necessário a implementação desses TADs por conta própria que, de fato, foi uma experiência muito positiva, pois permitiu uma implementação personalizada. Em outras palavras, seguindo os moldes originais da “Pilha” e da “Árvore”, foi usado a criatividade para modificações na implementação que melhor se adaptassem para a resolução efetiva do problema.

Além disso, toda implementação agrega algo na vida do programador. Ao longo do desenvolvimento do código foram enfrentadas várias falhas e defeitos dos quais foi possível extrair muito conhecimento. Ademais, a implementação descrita nesse relatório utilizou bastante manipulação com “ponteiros” que, de fato, agregou bastante no entendimento de como a memória é alocada, acessada e afins.

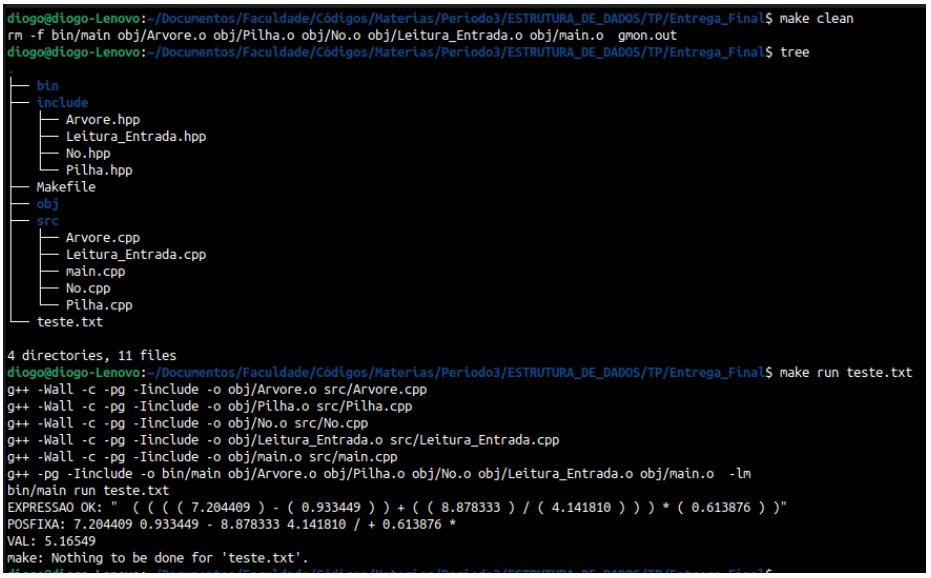
Por fim, o relatório teve extrema importância na absorção de conhecimento, pois tornou obrigatório o bom planejamento das ideias e uma implementação organizada, limpa e coerente. Tais fatores são vitais no desenvolvimento de qualquer programa e treinar isso é sempre muito agregador.

7. Bibliografia

- **CPlusPlus.com, Acesso em: 26 de abril de 2023.**
- **Notação infixa – Wikipédia, a enciclopédia livre, Acesso em: 26 de abril de 2023.**
- **Expressões matemáticas, Acesso em: 28 de abril de 2023.**
- **terminate called after throwing an instance of 'std::length_error' what(): basic_string::_M_create - Stack Overflow, Acesso em: 30 de abril de 2023.**
- **https://virtual.ufmg.br/20231/pluginfile.php/334774/mod_resource/content/1/08%20-%20Arvores.pdf , Acesso em: 30 de abril de 2023.**

8. Instruções para compilação e execução

- Acessar a pasta do TP pelo terminal
- Adicionar o arquivo a ser lido na pasta raiz do projeto, isto é, na pasta que contém o Makefile.
- Para rodar é necessário dar o seguinte comando:
    - make run *nome_do_arquivo*
        - Exemplo:



Nesse exemplo foi dado o comando ‘tree’ do terminal apenas para uma demonstração visual de onde o arquivo a ser lido deve se localizar.

- O make clean é importante na limpeza dos objetos e o executável gerado na compilação do programa, sendo recomendado sua utilização entre testes.