Trabalho Prático 1

Victor Augusto Hon Fonseca- número de matrícula: 2022035814

1.0 Introdução

Sob o contexto passado pelo enunciado do trabalho que estou ensinando matemática em uma escola de ensino fundamental e que cheguei no tópico expressões numéricas, foi proposto que fosse implementado um sistema resolvedor dessas expressões, capaz de armazená-las, de determinar se foram escritas corretamente, de acordo com seu tipo (ou infixa, ou posfixa), de resolvê-las e, além disso, capaz de mostrar como seria a expressão inserida convertida para a outra forma de notação. Para resolver esse problema, o programa receberá comandos de um arquivo “txt”, junto com as expressões numéricas de cada comando e, a partir disso, usando estruturas de dados, classes e funções determinadas realizará cada uma das funcionalidades requisitadas.

2.0 Método

O programa foi desenvolvido utilizando a linguagem de programação C++, as bibliotecas “iostream” e “string” dessa linguagem para os arquivos, de modo geral, e, no arquivo “main.cpp”, foram utilizadas as bibliotecas “fstream” e “sstream”, para a leitura das linhas do arquivo “txt” utilizado e interpretação dos termos de cada linha como “comando”, “tipo da expressão numérica” e “expressão numérica”. Além disso, vale relatar que o programa foi compilado utilizando o compilador G++, da GNU Compiler Collection, na IDE Visual Studio Code, utilizando o recurso Subsistema do Windows para Linux (WSL), do sistema operacional Windows.

2.1 Estruturas de Dados

É importante destacar que, durante a implementação do programa, além de classes, também foi necessário utilizar estruturas de dados auxiliares, para armazenar a expressão e seus termos. No meu programa, após a expressão ser lida, ela é armazenada como uma pilha de elementos string (Pilha<string>), mas durante a implementação de funções são utilizadas outras estruturas de dados como, por exemplo, filas circulares, também compostas de elementos string (FilaCircular<string>) e ponteiros do tipo Pilha<string>* e arrays de forma direta. Ambas as classes FilaCircular e Pilha foram implementadas manualmente sequencialmente, utilizando de templates, sendo a primeira composta de atributos como “capacidade, inicio, fim” e um ponteiro para um array do tipo T (template), já a segunda composta pelos atributos “capacidade, topo, tamanho” e um um array do tipo T (template). Com ambos os atributos capacidade representando o tamanho máximo da estrutura de dados.

Ambas possuem métodos já esperados para essas estruturas de dados, entre eles: construtor, destrutor, método que checa se a fila circular ou a pilha estão vazias, método que checa se a fila circular ou a pilha estão cheias, método para acrescentar elementos no final (aumentando o topo, o tamanho e o elemento com índice do novo topo, sendo igual ao novo elemento adicionado) e um método que, no caso da pilha, retira o último elemento e, no da fila circular, retira o primeiro elemento, ambos atribuindo seu valor a uma variável e o retornando.

2.2 Classes

O programa utiliza apenas uma classe principal, a classe Expressão, a qual possui como atributos: a string expression (que corresponde à expressão numérica, propriamente dita, adquirida do usuário, neste caso, lida de um arquivo “txt”), a string tipo (lida também do arquivo “txt”) e um atributo do tipo “Pilha<string>” que armazena cada termo da expressão. Além disso, essa classe possui os métodos principais “infixToPostfix” (transforma uma expressão em notação infixa em posfixa), “posfixaInfixa” (transforma uma expressão em notação posfixa em infixa), “resolverInfixa”, “resolverPosfixa”, que resolvem as expressões de cada tipo, e um método “valida” que, por sua vez, depende de “ler_infixa” retornar verdadeiro, no caso de uma expressão infixa, ou “ler_posfixa” retornar verdadeiro, no caso de uma expressão posfixa cada tipo de expressão. Ademais, busca-se também modularizar o programa dividindo nos arquivos “expressão.hpp”, que declara a classe, seus atributos e métodos, “expressão.cpp” que faz a implementação dos métodos, realizando essa mesma divisão para as classes fila circular e pilha também.

2.3 Funções

O programa conta com 7 funções principais, todas sendo métodos da classe “Expressão”, as quais são as responsáveis, em conjunto com outras funções auxiliares, para atender diretamente aos requisitos do trabalho.

-**ler_infixa**: Função que recebe como parâmetro a expressão em forma de string, checa termo a termo se a expressão atende vários requisitos de uma expressão infixa válida, como por exemplo, pra cada parênteses aberto ter um fechado, contribuindo para a robustez do código, antes de realizar operações na expressão. Se a expressão for válida é retornado um bool com valor true, se não for é retornado um false.

-**ler_posfixa:** Função que realiza os mesmos procedimentos que “ler_infixa” e tem as mesmas condições e retornos, porém avaliando a expressão de acordo com as regras de expressões posfixas.

-**valida:** Função que dependendo do retorno ou da função “ler_infixa” se a expressão for infixa, ou da função “ler_posfixa” se a expressão for posfixa, retorna uma Pilha de strings com os termos da expressão.

-**resolver_posfixa2:** Função que recebe como parâmetro o atributo pilha, da classe Expressão (Pilha<string>, com cada termo da expressão), cria um novo ponteiro do mesmo tipo. A partir disso, vai se lendo cada termo da pilha a partir do atributo array dela, se for um número ele é adicionado à nova pilha, se for um operador, desempilha-se os dois últimos números, realiza-se a operação entre eles, (pela função auxiliar operação2, e pela função auxiliar “str_to_ld”, a qual transforma o número em formato string para o número em formato long double, ambas utilizando de “stringstream”, para manter um grau maior de precisão), empilha-se o resultado e assim, por diante, até acabar de ler os termos da pilha parâmetro. Ao acabar imprime-se o resultado na tela.

-**posfixaInfixa**: Função semelhante à “resolver_posfixa”, com um algoritmo parecido, no entanto, quando um termo do array da pilha é um operador, desempilha-se os dois últimos termos, cria-se uma string igual ao primeiro elemento desempilhado, o operador e o segundo elemento desempilhado, com um parêntesis aberto no início da string e um fechado no final. Depois, empilha-se essa string na pilha e repete-se esse processo até acabar de se ler os elementos do array da pilha parâmetro.

-infixaPosfixa: Função que cria uma Pilha para os operadores e uma fila chamada output, caminha-se pelos termos da pilha com os termos da string se for número, ele é adicionado à fila output, se não for, utiliza regras de precedência entre operadores. A partir dela é determinado se o operador é adicionado diretamente ao output ou à pilha de operadores, ao final os elementos da pilha de operadores são invertidos e adicionados à fila output, cujos elementos são impressos no fim do código. Essa função foi inspirada no conteúdo do “Shunting Yard algorithm”, visto em um pequeno vídeo e em uma breve explicação de um site (ambos serão mostrados na secção de bibliografia)

-**resolver_infixa**: Nessa função também recebe-se como parâmetro a pilha com os termos da string, mas dentro dela cria-se um ponteiro para uma Pilha<string>, equivalente ao retorno da função infixaPostfixSemImprimir(função auxiliar que realiza exatamente o mesmo procedimento que “infixaPosfixa”, só que não imprime nada na tela) e chama-se a função auxiliar “resolver_posfixa”(recebe ponteiros do tipo Pilha<string>, ao invés de variáveis normais desse tipo), que recebe como parâmetro o ponteiro anterior. Dessa forma, imprime-se o resultado da expressão infixa do parâmetro, caso ela fosse posfixa, que é o mesmo de caso ela seja infixa.

Uma outra função auxiliar utilizada é a “get_precedence”, utilizada nas funções “infixToPostfix” e “infixToPostfixSemImprimir”, para determinar a precedência entre dois operadores, conforme o conteúdo do “Shunting Yard algorithm”, visto nas fontes já citadas.

3.0 Análise de Complexidade

3.1 Complexidade quanto ao tempo

-**resolver_posfixa2**: Como essa função possui dois loops, um pra cada elemento da pilha parâmetro e outro para cada caractere de cada elemento, e a função auxiliar operação chamada tem complexidade O(1), pois simplesmente resolve uma operação de matemática básica, a complexidade dessa função é O(n^2).

-posfixaInfixa: Essa função também possui dois loops, ambos com o propósito, semelhante ao de “resolver_posfixa2”, logo sua complexidade é O(n^2).

-ler_infixa; Apesar de haver mais de um loop, todos são com complexidade O(n), então a complexidade geral da função é O(n).

-**ler_posfixa**: Complexidade O(n), igual à da função “ler_infixa”.

-infixaPosfixa: Como essa função possui dois loops aninhados, um que percorre os elementos de uma pilha de strings, e outro que percorre cada caractere de cada elemento da pilha, então a complexidade dessa função é O(n^2).

-resolver_infixa: A complexidade dessa função é O(n^2), pois, embora, chame-se duas funções de complexidade O(n^2) dentro dela, ambas são sempre chamadas só uma vez, para qualquer caso, então a complexidade é O(n^2).

Para as funções auxiliares como a que gera a precedência, é apenas feito uma atribuição dependendo do operador (não há loop na função), logo sua complexidade é O(1) e, obviamente, a função infixaPosfixaSemImprimir, possui a mesma complexidade que infixaPosfixa, que é O(n^2) e a função “resolver_posfixa” possui a mesma complexidade que “resolver_posfixa2”

3.2 Complexidade quanto ao espaço

Considerando como forma de medida para a complexidade do espaço, o número de alocações dinâmicas realizadas, no meu programa, ao todo, há 8 alocações dinâmicas, presentes, por exemplo, na criação de filas (possuem como atributo um ponteiro para array) e na criação de ponteiros do tipo Pilha, alguns sendo equivalentes ao retorno de funções. Entretanto, essas alocações não mudam de tamanho, dependendo da entrada, sendo independentes dela. Logo, a complexidade espacial é O(1).

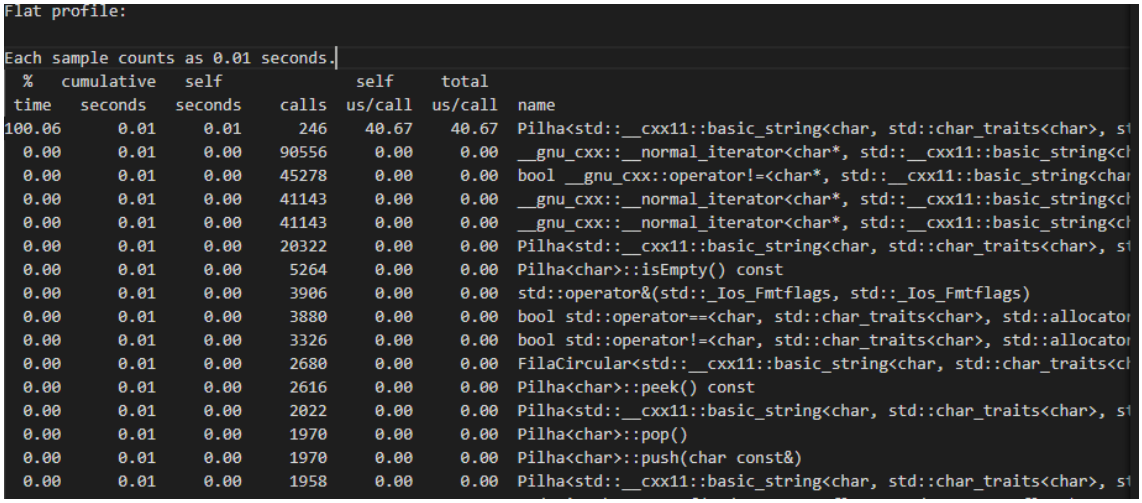
4.0 Estratégias de Robustez

Infelizmente, não foi possível, implementar exceções no programa principal (apenas nas filas e pilhas, caso alguma pilha ou fila esteja cheia ou vazia), pois, nas orientações do trabalho, devem ser lidos comandos de um “arquivo.txt”, mas se uma das expressões comandadas a ser armazenada for inválida, a execução do programa não pode ser paralisada, deve apenas ser impresso na tela uma mensagem semelhante a “Expressão Inválida”, para esse determinado comando e qualquer outro, até que outra linha indique o comando de armazenar uma nova expressão. Desse modo, não foi possível implementar exceções.

Além disso, caso o tipo esteja correto, a expressão é passada por parâmetro da função “ler_infixa”, ou “ler_posfixa”, e, dependendo das regras para cada tipo de expressão, como no caso da infixa, para cada parênteses aberto, deve haver um fechado, não pode haver dois números lado a lado, sem um operador entre eles, ou, novamente, o tipo da expressão vai ser “”, assim como sua string expression., e, devido a implementação do “main.cpp”, até haver um novo comando “LER”, o output para cada linha será semelhante à “Expressão Inválida”, ou a expressão será armazenada como uma Pilha<string>, com cada termo dela. Ambas as funções, avaliam a string expression., de acordo, com várias regras, como para infixas, além das já ditas, se a quantidade números é sempre uma unidade maior que a de operadores, o que também é avaliado na posfixa.

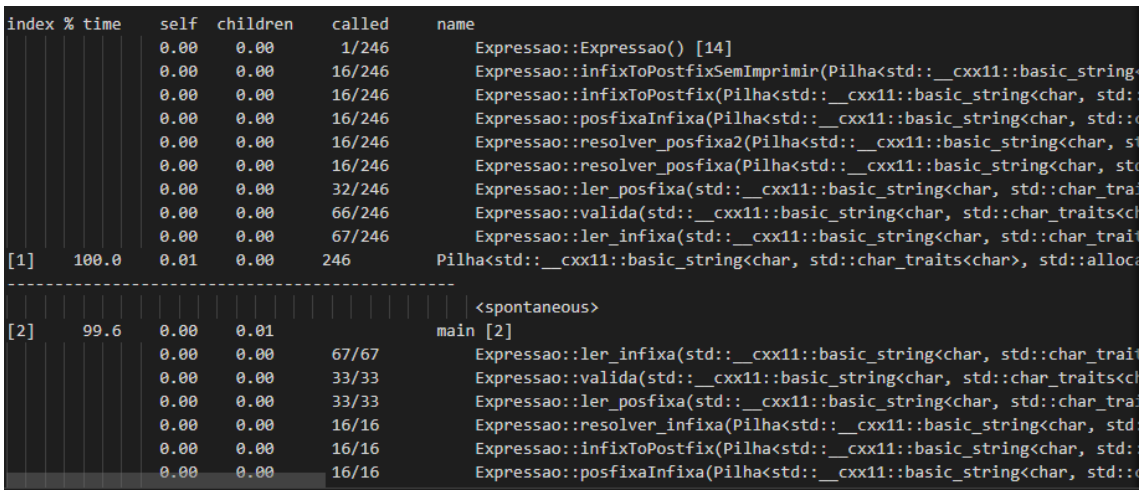
5.0 Análise Experimental

Para a minha análise experimental, uma das ferramentas utilizadas, foi o “gprof”, e o relatório “flat profile”, gerado a partir dessa ferramenta.



```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.06      0.01     0.01      246    40.67    40.67  Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, st
  0.00      0.01     0.00    90556     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<ch
  0.00      0.01     0.00    45278     0.00     0.00  bool __gnu_cxx::operator!=<char*, std::__cxx11::basic_string<char
  0.00      0.01     0.00    41143     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<ch
  0.00      0.01     0.00    41143     0.00     0.00  __gnu_cxx::__normal_iterator<char*, std::__cxx11::basic_string<ch
  0.00      0.01     0.00    20322     0.00     0.00  Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, st
  0.00      0.01     0.00     5264     0.00     0.00  Pilha<char>::isEmpty() const
  0.00      0.01     0.00     3906     0.00     0.00  std::operator&(std::_Ios_Fmtflags, std::_Ios_Fmtflags)
  0.00      0.01     0.00     3880     0.00     0.00  bool std::operator==<char, std::char_traits<char>, std::allocator
  0.00      0.01     0.00     3326     0.00     0.00  bool std::operator!=<char, std::char_traits<char>, std::allocator
  0.00      0.01     0.00     2680     0.00     0.00  FilaCircular<std::__cxx11::basic_string<char, std::char_traits<ch
  0.00      0.01     0.00     2616     0.00     0.00  Pilha<char>::peek() const
  0.00      0.01     0.00     2022     0.00     0.00  Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, st
  0.00      0.01     0.00     1970     0.00     0.00  Pilha<char>::pop()
  0.00      0.01     0.00     1970     0.00     0.00  Pilha<char>::push(char const&)
  0.00      0.01     0.00     1958     0.00     0.00  Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, st
```



```
index % time    self  children    called     name
                0.00    0.00       1/246         Expressao::Expressao() [14]
                0.00    0.00      16/246         Expressao::infixToPostfixSemImprimir(Pilha<std::__cxx11::basic_string
                0.00    0.00      16/246         Expressao::infixToPostfix(Pilha<std::__cxx11::basic_string<char, std::
                0.00    0.00      16/246         Expressao::posfixaInfixa(Pilha<std::__cxx11::basic_string<char, std::<
                0.00    0.00      16/246         Expressao::resolver_posfixa2(Pilha<std::__cxx11::basic_string<char, st
                0.00    0.00      16/246         Expressao::resolver_posfixa(Pilha<std::__cxx11::basic_string<char, st
                0.00    0.00      32/246         Expressao::ler_posfixa(std::__cxx11::basic_string<char, std::char_trai
                0.00    0.00      66/246         Expressao::valida(std::__cxx11::basic_string<char, std::char_traits<ch
                0.00    0.00      67/246         Expressao::ler_infixa(std::__cxx11::basic_string<char, std::char_trait
[1]    100.0    0.01    0.00     246         Pilha<std::__cxx11::basic_string<char, std::char_traits<char>, std::alloca
-----------------------------------------------
                                                <spontaneous>
[2]     99.6    0.00    0.01                 main [2]
                0.00    0.00      67/67          Expressao::ler_infixa(std::__cxx11::basic_string<char, std::char_trait
                0.00    0.00      33/33          Expressao::valida(std::__cxx11::basic_string<char, std::char_traits<ch
                0.00    0.00      33/33          Expressao::ler_posfixa(std::__cxx11::basic_string<char, std::char_trai
                0.00    0.00      16/16          Expressao::resolver_infixa(Pilha<std::__cxx11::basic_string<char, std:
                0.00    0.00      16/16          Expressao::infixToPostfix(Pilha<std::__cxx11::basic_string<char, std::
                0.00    0.00      16/16          Expressao::posfixaInfixa(Pilha<std::__cxx11::basic_string<char, std::<
```

Essas imagens mostram apenas trechos do relatório “flat profile” gerado pelo gprof (o relatório era muito grande e todos os tempos variavam eram ou 0.0 ou 0.1, mas o relatório completo vai estar na pasta bin, do programa). Esse resultado curiosamente mostra todos os tempos como 0.0 ou 0.1, como a ferramenta foi utilizada corretamente, compilando-se o código utilizando a flag “-pg” e a geração do relatório também, uma das possíveis causas que explica esse curioso comportamento, que contrasta com o elevado número de chamadas de cada operação, presente na primeira foto é que as operações realizadas são mais simples, não demandando um tempo de Cpu gasto significativo, com exceção apenas nas operações relacionadas à pilhas, cujo array ligado a elas possui um elevado número de elementos. Isso pode ser explicado pelo fato de que as alocações dinâmicas realizadas são constantes independentemente da entrada, o que gera uma complexidade O(1), as estruturas de dados usadas são relativamente simples (pilhas e filas circulares) quando comparadas com mais complexas como árvores e que embora haja loops em algumas funções, a complexidade delas não ultrapassa O(n^2).

Além disso, com o intuito de checar a eficiência do código, foi utilizada a ferramenta “Valgrind”, buscando checar, por exemplo, se havia problemas ligados ao gerenciamento de memória.

```
==2337== HEAP SUMMARY:
==2337==     in use at exit: 0 bytes in 0 blocks
==2337==   total heap usage: 4,132 allocs, 4,132 frees, 6,573,052 bytes allocated
==2337==
==2337== All heap blocks were freed -- no leaks are possible
==2337==
==2337== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Entretanto, como é possível perceber na imagem não houve erros ligados ao gerenciamento de memória.

```
[Inferior 1 (process 2503) exited normally]
(gdb)
```

Além disso, também foi utilizada a ferramenta Gdb, com o intuito de fazer uma verificação ainda maior por eventuais erros, mas como, pode ser visto na imagem, não houve nenhum erro detectado por mais essa ferramenta, podendo concluir que, o desempenho computacional na execução do programa não foi afetado por nenhum erro, por exemplo, ligado ao gerenciamento de memória

```
--------------------------------------------------------------------------------
  n        time(i)         total(B)   useful-heap(B) extra-heap(B)    stacks(B)
--------------------------------------------------------------------------------
 24     33,756,954           147,744          147,649            95            0
 25     34,677,916           115,736          115,650            86            0
 26     35,926,566           211,768          211,663           105            0
 27     37,548,826           147,744          147,646            98            0
 28     38,843,595            84,704           84,600           104            0
 29     39,898,739           115,776          115,670           106            0
 30     41,721,097           147,704          147,626            78            0
 31     43,254,126            86,560           86,462            98            0
 32     44,421,126            84,936           84,848            88            0
 33     45,285,489           213,056          212,937           119            0
99.94% (212,937B) (heap allocation functions) malloc/new/new[], --alloc-fns, etc.
->34.12% (72,704B) 0x48F0C19: ??? (in /usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.28)
| ->34.12% (72,704B) 0x4011B99: call_init.part.0 (dl-init.c:72)
|   ->34.12% (72,704B) 0x4011CA0: call_init (dl-init.c:30)
|     ->34.12% (72,704B) 0x4011CA0: _dl_init (dl-init.c:119)
|       ->34.12% (72,704B) 0x4001139: ??? (in /usr/lib/x86_64-linux-gnu/ld-2.31.so)
|
->30.05% (64,016B) 0x10BFBC: Expressao::infixToPostfixSemImprimir(Pilha<std::__cxx11::basic_string<char, std::char_tra
| ->30.05% (64,016B) 0x10D97F: Expressao::resolver_infixa(Pilha<std::__cxx11::basic_string<char, std::char_traits<char
|   ->30.05% (64,016B) 0x11068F: main (in /home/victorufmg/EstDados 2023/TP1DeuCertoDefinitivo/bin/app)
|
->30.05% (64,016B) 0x10D58C: Expressao::resolver_posfixa(Pilha<std::__cxx11::basic_string<char, std::char_traits<char>,
| ->30.05% (64,016B) 0x10D9AE: Expressao::resolver_infixa(Pilha<std::__cxx11::basic_string<char, std::char_traits<char
|   ->30.05% (64,016B) 0x11068F: main (in /home/victorufmg/EstDados 2023/TP1DeuCertoDefinitivo/bin/app)
```

.

Ademais, utilizou-se ferramentas de profiling, como o Valgrind Massif, para visualizar o uso da memória ao longo do tempo, permitindo, por exemplo a visualização de número de acessos e acesso a endereços. A partir do arquivo gerado, pode-se perceber, por exemplo, nesse trecho, que a maior parte da memória alocada foi na alocação dinâmica, na criação de ponteiros, por exemplo, para o tipo “Pilha<string>”, o que ocorre com destaque nas funções “infixToPostfixSemImprimir”, o que é compreensível, considerando que dentro dessa função, cria-se um ponteiro do tipo mencionado e ele é o retorno da função e cria-se uma fila circular, a qual possui como atributo um ponteiro para array (além disso, essa função é chamada dentro de “resolver_infixa”), “resolver_posfixa” (recebe como parâmetro um ponteiro, cria-se um ponteiro do tipo mencionado dentro dele, ela interage com funções auxiliares como “operação2” e ela é chamada dentro de “resolver_infixa”) e, por fim, “resolver_infixa”, a qual cria um ponteiro para o retorno da função “infixToPostfixSemImprimir” e o usa como parâmetro

para chamar “resolver_posfixa”. Logo essa função não só utiliza da alocação dinâmica, mas se relaciona com duas funções que são as que mais necessitam de alocação de memória. Assim, nota-se que essas funções, juntamente com a classe Pilha, que apresenta um array com considerável número de elementos, como atributo, e, mais especificamente, os ponteiros criados usando “new”, foram os principais fatores que afetaram o desempenho computacional do programa atual.

6.0 Conclusões

Nesse trabalho, foi requisitado que fosse implementado um sistema resolvedor de expressões numéricas que estivessem ou em notação infixa ou em notação posfixa. Essa proposta foi um excelente exercício de revisão e consolidação de conteúdos ensinados em Estruturas de Dados, como as estruturas Pilha e Fila Circular, e a análise de complexidade de funções. Além disso, foi muito útil para a revisão de conteúdos mais antigos do curso como alocação dinâmica de memória e elaboração de classes.

Entretanto, é importante frisar que um dos aprendizados mais importantes deste trabalho, foi sobre a importância do raciocínio lógico e abstrato para a solução de problemas, tendo sido necessário pensar em soluções criativas para realizar certas operações matemáticas. Um dos exemplos mais perceptíveis foi quando estava buscando implementar a função de resolver expressões posfixas. Em resumo, inicialmente achava que expressões posfixas não podiam intercalar números e operadores, sendo basicamente uma sequência de números e depois uma sequência de operadores, tendo elaborado um algoritmo inicial com uma pilha de números e uma fila de operadores. Porém, quando descobri que poderia haver números e operadores intercalados tive que pensar em uma solução para esse problema, que foi quando pensei que se eu resolve-se as sub-expresões que possuíam números e operadores intercalados e os acrescentasse como resultado na pilha de operadores, meu algoritmo passaria a cobrir também esse tipo de expressões posfixas, mas, então, percebi que poderia resolver a situação simplesmente usando uma única pilha e que a cada operador lido da expressão numérica, bastava desempilhar os dois últimos elementos, realizar a operação entre eles e empilhar o resultado até a expressão ser completamente lida.

Bibliografia

https://youtu.be/Jd71l0cHZL0 (vídeo de explicação breve do Shunting Yard Algorithm)

https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm
(explicação breve sobre precedência de operadores em uma expressão Posfixa)

Instruções para Compilação e Execução

1. Deve-se abrir o terminal.
2. Entrar na pasta TP, usando o comando “cd TP”.
3. Utilizar o comando “make run” para compilar e executar o código.
4. Após avaliação e uso, caso desejado, utilizar o comando “make clean” para deletar os arquivos desnecessários, como o executável e arquivos da pasta “obj”(“.o”).