Trabalho Prático

Resolvedor de Expressão Numérica

Thales Augusto Rocha Fernandes - thalesarf@ufmg.br

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais

1. Introdução

As expressões numéricas estão presentes no dia a dia de todos os cidadãos, seja no trabalho, estudo ou situações cotidianas. Essas expressões podem ser representadas por diferentes notações como a infixa, que é a representação com operadores entre os operandos e é usualmente utilizada em calculadoras, como no caso: "(5 + (3 * 2))". Uma outra notação que pode ser usada é a posfixa que é a representação com operadores após os operandos, como no caso: "5 3 2 * +". O objetivo desse trabalho é criar um código em C/C++ que recebe, imprime e resolve expressões numéricas representadas pela notação posfixa ou pela notação infixa, além de poder converter uma expressão em uma notação para a outra. Para concretizar a funcionalidade, o programa foi dividido entre 6 funções em que: duas delas servem para conferir se a expressão recebida é válida ou não; duas funções realizam a conversão de uma expressão infixa para posfixa, ou vice-versa; uma função realiza o cálculo da expressão após todas as operações serem realizadas; uma função agrupa todas as anteriores para realizar o resolvedor de expressões numéricas. Essa implementação não fará uso de estruturas de dados provenientes de bibliotecas específicas da linguagem C++. Assim, a representação da estrutura de dados usadas estão disponíveis no código e possuem funções e características específicas para este projeto.

2. Método

2.1. Configurações da máquina

Sistema operacional: WSL2 - Ubuntu 20.04 LTS

2.2. Estruturas de dados

Na implementação desse trabalho, as estruturas de dados utilizadas foram a pilha e a fila. O uso da pilha foi em vista da sua característica de que o último elemento a ser inserido nessa estrutura é o primeiro a ser retirado, o que é essencial para a implementação das funções que serão explicadas no tópico 2.4. Essa pilha é do tipo sequencial com o uso de arranjos e alocação estática e a pilha usa um template para que ela possa armazenar itens no formato char, float ou string, de acordo com a função. As operações que essa pilha realiza são criar uma nova pilha, testar se a pilha está vazia, empilhar um item, desempilhar um item, limpar a pilha e retornar o item do topo da pilha. O uso da fila foi em vista da facilidade de conversão que essa estrutura possibilita, já que sempre que formos desenfileirar os caracteres dessa estrutura eles sairão na ordem em que foram enfileirados. Essa fila é do tipo sequencial com o uso de arranjos e alocação estática e a fila usa um template para que ela possa armazenar itens em diversos formatos. As operações que essa fila realiza são criar uma nova fila, testar se a fila está vazia, enfileirar um item, desenfileirar um item, limpar a fila e retornar o item da frente da fila.

2.3. Classes

As duas classes utilizadas nesse programa implementam as estruturas de dados pilha e fila, com as características descritas no tópico anterior. As classes foram implementadas usando template e, dessa forma, estão contidas inteiramente em um arquivo '.h'.

2.4. Funções

O código possui 6 principais funções que estão separadas em diferentes arquivos que possuem o mesmo nome de cada função principal, com exceção da terminação ".cpp" e ".h" contida nesses arquivos.

Main: Essa função lê o arquivo de entrada do usuário e ativa as outras funções do programa para realizar o resolvedor de expressão numérica. A entrada do programa pode variar entre cinco comandos. Os comandos "LER POSFIXA" ou "LER INFIXA" que indica o formato da expressão numérica e que realiza, em seguida, a leitura da expressão que, se for válida, será armazenada. O comando "POSFIXA" imprime uma expressão armazenada no formato pósfixo. O comando "INFIXA" imprime uma expressão numérica armazenada no formato infixo. E o comando "RESOLVE" imprime o resultado da expressão numérica armazenada no formato float com precisão de 6 casas decimais.

Para analisar esse formato de entrada o código declara três variáveis do tipo string chamadas s, z e c, declara duas variáveis do tipo bool chamadas infixa e posfixa e também declara uma fila chamada exp com itens do tipo char. A string s e c serão usadas para ler os 5 comandos citados anteriormente. A string z será usada para ler a expressão numérica que é escrita na entrada logo após o comando "LER". A fila exp é usada para armazenar a expressão numérica lida pela string z caso essa expressão for válida, o que será verificado através das funções ValidaPos e ValidaIn que serão explicadas posteriormente. As variáveis bool infixa e posfixa são inicializadas como falsas, sendo que a infixa se torna verdadeira quando uma expressão numérica no formato infixo é armazenado na fila e a posfixa se torna verdadeira quando essa expressão armazenada estiver no formato pós-fixo, de forma a ativar condicionais que permitirão a realização do resolvedor de expressões numéricas.

ValidaIn: Essa função recebe uma string contendo uma expressão numérica como parâmetro e retorna um bool com valor true se for válida e false se for inválida. Após ser ativada, a função verifica se uma expressão na notação infixa é válida ou não, por meio da checagem de 5 condições.

A primeira condição é saber se todos os caracteres da expressão são válidos e, para isso, o código remove todos os espaços da expressão e verifica se todos os caracteres são dígitos, pontos, parênteses ou operadores.

A segunda condição é verificar se os números float da expressão são válidos e se não há dois números em sequência, algo incorreto na notação infixa. Para checar essa condição, o programa verifica se os números estão no formato certo por meio da divisão da expressão em substrings e na conversão dessas substrings em números float. Sempre que um número é convertido em float uma variável auxiliar bool fica com sinal lógico true e sempre que um operador é lido esse auxiliar fica com sinal lógico false, de forma a checar que sempre tem um operador entre os números. Se um número for lido enquanto o bool auxiliar for true, a expressão é inválida.

A terceira condição é verificar se não há dois operadores em sequência e requer um código similar ao da segunda condição. Para realizar essa verificação também usaremos um bool auxiliar que fica com sinal lógico true quando é lido um operador da expressão e sinal false quando outro caracter for lido. Se um operador for lido e o bool auxiliar for true, a expressão é inválida.

A quarta condição é checar se todos os parêntes da expressão foram fechados, com o uso de uma pilha de Tipo char. Para isso, o código percorre toda a expressão e empilha todos os caracteres '(' encontrados. Além disso, se o programa encontrar um caractere ')' iremos desempilhar um item da pilha. No entanto, se tentarmos desempilhar a pilha e ela estiver

vazia, a expressão é inválida. Depois de percorrer a expressão a pilha deve estar vazia e, se não estiver, a expressão não é válida.

A quinta condição é verificar se o número de operadores é igual ao número de operandos decrescido de 1 unidade. Para isso, o código percorre a expressão contando os números e os operadores e, após checar todos os caracteres da string, verifica a condição.

ValidaPos: Essa função recebe uma string contendo uma expressão como parâmetro e retorna um bool com valor true se for válida e false se for inválida. Após ser ativada, a função verifica se uma expressão numérica na notação posfixa é válida ou não, por meio da checagem de 2 condições.

A primeira condição é saber se todos os caracteres da expressão são válidos e, para isso, o código percorre toda a string e checa se todos os caracteres são dígitos, pontos ou operadores. Caso houver algum caracter inválido, a expressão é inválida.

A segunda condição é verificar que sempre há dois operandos antes de um operador na expressão, de forma que se um operador for lido um operando é decrescido. Essa condição é checada com o uso de uma pilha de tipo float, de forma que o código percorre toda a string contendo a expressão e, se um número for encontrado, ele é empilhado. Quando o programa encontrar um operador na string, ele verifica se há pelo menos dois itens na pilha e desempilha um item caso for verdade. Se o código encontrar um operador e não houver pelo menos dois itens na pilha, a expressão é inválida.

InParaPos: Essa função recebe uma fila de itens de tipo char contendo uma expressão numérica na notação infixa como referência e retorna uma string contendo uma expressão na notação posfixa equivalente à de entrada. Nesse contexto, o código faz o uso de uma função auxiliar chamada prioridade que irá atribuir valores maiores para operadores de multiplicação e divisão e valores menores para operadores de soma e subtração. O código também faz uso de uma string auxiliar que será manipulada e retornada e de uma pilha com itens do tipo char que usaremos para empilhar os parênteses e os operadores da expressão infixa. No início da função, o programa percorre todos os caracteres da expressão por meio de um loop, sendo que se o caracter atual for de um operando iremos adicionar os caracteres na string auxiliar até aparecer um espaço. No caso de o caracter atual ser igual a um parênteses de abertura, o código empilha esse elemento. No caso de o caracter atual ser igual a um parênteses de fechamento, o programa desempilha os caracteres da pilha e os adicionam na string auxiliar até que a pilha esteja vazia ou o elemento do topo da pilha seja um parênteses de abertura. Nesse processo, também adicionamos os devidos espaços na string auxiliar após cada desempilhamento e descartamos o parênteses de abertura contido na pilha. No caso de o caracter atual ser um operador, o código verifica se ele possui prioridade maior do que os outros elementos contidos na pilha. Se o caracter atual possui uma maior prioridade ele é empilhado, se ele não possuir maior prioridade que os outros elementos o programa desempilha os itens e os adiciona na string até que a prioridade do caracter atual seja maior do que a do item no topo da pilha. Nesse processo, também adicionamos os devidos espaços na string auxiliar. Após percorrer toda a expressão de entrada, ainda restará operadores na pilha e, dessa maneira, o programa irá desempilhar esses itens restantes e adicionar na string auxiliar. Finalmente, a string auxiliar é retornada contendo a expressão convertida.

PosParaIn: Essa função recebe uma fila de itens de tipo char contendo uma expressão numérica na notação posfixa como referência e retorna uma string contendo uma expressão na notação infixa equivalente à de entrada. Nesse contexto, o código faz uso de uma string auxiliar que será manipulada e retornada e de uma pilha com itens do tipo string que usaremos para empilhar os operandos e operadores. Além disso, o programa também faz uso da função istringstream da biblioteca sstream, sendo que o objetivo dessa função é

dividir a expressão de entrada em tokens, de forma que sempre que um espaço for encontrado ela é dividida em dois pedaços chamados tokens. Em seguida, o código percorre um token por vez e realiza diferentes ações dependendo do seu valor. Se o token for um número float, o programa empilha esse operando na pilha. Se o token não for um float ele é necessariamente um operador, já que estamos tratando da notação posfixa, assim, o código desempilha dois operandos da pilha e forma uma pequena expressão numérica com eles juntamente com o operador que é o token atual. Após formar essa subexpressão por meio da manipulação de strings, juntamente com a adição de parênteses, o programa empilha essa string resultante e repete o processo até o fim dos tokens. Finalmente, a expressão resultante convertida estará contida como o único elemento da pilha que será retornado.

Calculo: Essa função recebe uma fila de itens de tipo char contendo uma expressão numérica na notação posfixa como referência e retorna um número float contendo o resultado da expressão. Nesse contexto, o código faz uso de uma pilha com itens do tipo float que usaremos para empilhar os operandos e também faz uso da função istringstream da biblioteca sstream com o mesmo objetivo da função anterior. Em seguida, o programa percorre todos os tokens da expressão de entrada, de forma que, se o token for um número, esse número é convertido para float e empilhado. No caso do token ser um operando, o código desempilha dois itens em duas variáveis float auxiliares e realiza a operação entre esses números, de forma que o segundo item a ser desempilhado fica à esquerda do operador e o resultado dessa operação é novamente empilhado. Após percorrer todos os tokens, só sobra um número na pilha que é o resultado que será retornado.

3. Análise de Complexidade

Nessa análise, vamos considerar o valor n como o número de caracteres da expressão passada como parâmetro para as funções. Nenhuma função do código usa recursividade nem estruturas de dados ou classes que possuem propriedades recursivas, o que é importante para a análise de complexidade de espaço, já que alguns tipos de recursividade podem sobrecarregar a pilha de execução do programa. A função main não é analisada, já que ela apenas agrupa as outras funções.

ValidaIn – Complexidade de tempo: Essa função é composta por quatro loops independentes que percorrem todos os caracteres da string de entrada e de uma pilha de char com alocação sequencial. Cada loop tem um custo de O(n) e como a pilha usada tem alocação estática, o custo das suas operações é O(1) e, consequentemente, o custo da função é igual a uma contante multiplicado por uma função de valor O(n). Assim, o custo assintótico dessa função é igual a O(n).

ValidaIn – Complexidade de espaço: Além dos caracteres da string de entrada, essa função manuseia variáveis auxiliares do tipo bool, inteiro, float e também utiliza uma pilha de tipo char, de forma que todos esses elementos utilizam alocação estática. No pior caso possível, essa função recebe como parâmetro uma string com uma expressão inválida que contém apenas parênteses de abertura. Essa string anterior fará com que um dos loops da função empilhe todos os seus caracteres dentro da pilha, o que ocupará uma grande parte da memória. Dessa maneira, devido à possibilidade desse pior caso ocorrer, o custo de espaço da função é O(n).

ValidaPos – Complexidade de tempo: Essa função é composta por três loops, em que um é independente e os outros dois são aninhados e de uma pilha de float com alocação estática. O loop independente percorre toda a string e verifica todos os caracteres, resultando em um custo de O(n). O segundo loop também percorre toda a string e possui outro loop aninhado que percorrerá todos os caracteres da string caso essa expressão de entrada seja composta apenas por números e pontos. No entanto, as condições de parada desses dois loops usam a mesma variável e o loop aninhado acrescenta uma unidade do inteiro usado para parada a

cada ciclo. Dessa forma, mesmo estando aninhados, esses dois loops funcionam como um único loop com custo de O(n). A pilha usada tem alocação estática sendo que o custo das suas operações é O(1) e, consequentemente, o custo da função é igual a uma constante multiplicado por uma função de valor O(n). Assim, o custo assintótico dessa função é igual a O(n).

ValidaPos – Complexidade de espaço: Além dos caracteres da string de entrada, essa função manuseia variáveis auxiliares do tipo float e uma pilha de tipo float, de forma que todos esses elementos utilizam alocação estática. No pior caso possível, essa função recebe como parâmetro uma string com uma expressão inválida que contém apenas números inteiros. Essa string anterior fará com que um dos loops da função empilhe todos esses números dentro da pilha, o que ocupará uma grande parte da memória. Dessa maneira, devido à possibilidade desse pior caso ocorrer, o custo de espaço da função é O(n).

PosParaIn – Complexidade de tempo: Essa função é composta por um loop que percorre todos os tokens da expressão de entrada e de uma pilha de string com alocação estática. O pior caso de custo de tempo para essa função seria quando os números da expressão forem todos inteiros sem parte decimal, já que teriamos n/2 tokens. A pilha usada tem alocação estática sendo que os custos das suas operações é O(1). Dessa forma, o custo assintótico dessa função é igual a O(n).

PosParaIn – Complexidade de espaço: Além dos tokens formados pela expressão de entrada, essa função manuseia strings auxiliares e uma pilha de tipo string. A estrutura string utiliza memória dinâmica, de forma que as strings auxiliares dessa função podem ter apenas um caracter ou podem ser do tamanho da expressão de entrada. Como essa função converte a expressão da notação posfixa para infixa, a string resultante é maior do que a de entrada, mas a quantidade de elementos dentro da pilha de tipo string é sempre menor do que o número de tokens total da expressão de entrada, devido ao funcionamento da função descrito anteriormente. No pior caso possível, a expressão de entrada possui uma grande quantidade de operandos no início e uma grande quantidade de operadores no final, de forma que haja uma grande quantidade de números empilhados na pilha e essa quantidade se aproxime de n. Dessa maneira, o custo de espaço da função é O(n).

InParaPos – Complexidade de tempo: Essa função é composta por 4 loops, de forma que 2 desses loops estão dentro de outro loop e há um loop independente dos outros 3. Paralelamente, a função também possui uma pilha de char com alocação estática. O loop principal percorre todas as caracteres da expressão com um custo de O(n) e possui outros dois loops dentro dele. Um desses outros loops é ativado quando o caracter da expressão de entrada for um parênteses de fechamento e o outro loop é ativado quando o caracter for um operador, de forma que ambos os loops possuem custo de O(n). O último loop independente é ativado no final da função e percorre a pilha resultante das outras funções. Essa pilha resultante tem um valor fixo bem menor do que n devido ao funcionamento da função, fazendo com que esse loop independente possua um custo O(1). A pilha usada tem alocação estática e os custos de suas operações é O(1). Dessa maneira, o custo assintótico dessa função é igual a O(n^2).

InParaPos – Complexidade de espaço: Além dos caracteres da expressão de entrada, essa função manuseia outras strings auxiliares e uma pilha de tipo char que utiliza alocação estática. Apesar da complexidade de tempo dessa função ser O(n^2), a complexidade de espaço é menor, já que a expressão infixa de entrada sempre possui uma quantidade de caracteres limitados entre parênteses e os itens empilhados são desempilhados quando um parênteses de fechamento é encontrado. Assim, cada iteração do loop da função realizará um número limitado de operações e a pilha de char do programa sempre possuirá um tamanho limite e não infinito. Dessa maneira, o custo de espaço da função é O(1).

Calculo – Complexidade de tempo: Essa função é composta por um loop que percorre todos os tokens da expressão de entrada e de uma pilha de float com alocação estática. O pior caso de custo de tempo para essa função seria quando os números da expressão forem todos inteiros sem parte decimal, já que teríamos n/2 tokens. A pilha usada tem alocação estática sendo que os custos das suas operações é O(1). Dessa forma, o custo assintótico dessa função é igual a O(n).

Calculo – Complexidade de espaço: Além dos tokens formados pela expressão de entrada, essa função manuseia outras variáveis auxiliares e uma pilha de tipo float, de forma que todos esses elementos utilizam alocação estática. No pior caso possível, a expressão de entrada possui uma grande quantidade de operandos no início e uma grande quantidade de operadores no final, de forma que haja uma grande quantidade de números empilhados na pilha e essa quantidade se aproxime de n. Dessa maneira, considerando os piores casos, o custo de espaço da função é O(n).

4. Análise de Robustez

Analisando a parte do tratamento de erros, os únicos erros que podem ocorrer são erros de expressões inválidas ou erro de divisão por zero na expressão numérica. Se a expressão numérica do arquivo de entrada for inválida, é imprimido na tela uma mensagem que diz "ERRO: EXP NAO VALIDA" na tela. No caso do arquivo de entrada usar os comandos "INFIXA", "POSFIXA" ou "RESOLVE" quando não há uma expressão numérica armazenada, é imprimido na tela uma mensagem que diz "ERRO: EXP NAO EXISTE". Se ocorrer uma divisão por zero na expressão, o programa é finalizado por meio de um throw de uma estrutura chamada "DivisaoPorZero". Esse throw só irá ocorrer no momento que o usuário tentar resolver a expressão, já que apenas o processo de cálculo da expressão verifica se esse erro ocorre.

5. Análise Experimental

Iremos analisar o tempo gasto pelo programa com o auxílio da ferramenta gprof e time na execução de dois casos. No primeiro caso, vamos inserir uma expressão de entrada com 580 caracteres na notação posfixa, sendo que essa expressão será armazenada, convertida para infixa e imprimida e, ao final, será resolvida. Há dois arquivos de entrada para esse primeiro caso, um desses arquivos repete o processo anterior 1000 vezes e o outro arquivo repete o processo 50000 vezes. A cada vez que o processo é repetido, a função ValidaPos é ativada uma vez, a função PosParaIn é ativada uma vez e a função Calculo é ativada uma vez.

Arquivo com 1000 repetições:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.00      0.01     0.01     1000    10.00    10.00  ValidaPos(std::__cxx11::basic_
  0.00      0.01     0.00    94000     0.00     0.00  void std::__cxx11::basic_strin
  0.00      0.01     0.00    89000     0.00     0.00  std::__cxx11::basic_string<cha
  0.00      0.01     0.00    45000     0.00     0.00  float __gnu_cxx::__stoa<float,
  0.00      0.01     0.00    44000     0.00     0.00  operacao(float, float, char)
  0.00      0.01     0.00     4000     0.00     0.00  frame_dummy
  0.00      0.01     0.00     1000     0.00     0.00  Calculo(std::__cxx11::basic_st
  0.00      0.01     0.00     1000     0.00     0.00  PosParaIn(std::__cxx11::basic_
  0.00      0.01     0.00     1000     0.00     0.00  FilaArranjo<std::__cxx11::basi
  0.00      0.01     0.00        1     0.00     0.00  FilaArranjo<std::__cxx11::basi
```

```
~/tpgprof$ time ./a.out < entrada1.txt > saida1.txt

real    0m0.452s
user    0m0.071s
sys 0m0.075s
```

Arquivo com 50000 repetições:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 44.00      0.22     0.22    50000     4.40     5.72  PosParaIn(std::__cxx11::ba
 22.00      0.33     0.11   200000     0.55     0.55  frame_dummy
 14.00      0.40     0.07    50000     1.40     1.40  ValidaPos(std::__cxx11::ba
  8.00      0.44     0.04    50000     0.80     1.35  Calculo(std::__cxx11::bas
  6.00      0.47     0.03  4700000     0.01     0.01  void std::__cxx11::basic_
  4.00      0.49     0.02                             main
  2.00      0.50     0.01  4450000     0.00     0.00  std::__cxx11::basic_strin
  0.00      0.50     0.00  2250000     0.00     0.00  float __gnu_cxx::__stoa<f
  0.00      0.50     0.00  2200000     0.00     0.00  operacao(float, float, ch
  0.00      0.50     0.00    50000     0.00     0.00  FilaArranjo<std::__cxx11:
  0.00      0.50     0.00        1     0.00     0.00  FilaArranjo<std::__cxx11:
```

```
~/tpgprof$ time ./a.out < entrada.txt > saida.txt

real    0m19.965s
user    0m3.781s
sys 0m3.015s
```

No segundo caso, vamos inserir uma expressão de entrada com 848 caracteres na notação infixa sendo que essa expressão será armazenada, convertida para posfixa e imprimida e, ao final, será resolvida. Há dois arquivos de entrada para esse segundo caso de forma que, um desses arquivos repete o processo anterior 1000 vezes e o outro arquivo repete o processo 50000 vezes. A cada vez que o processo é repetido, a função ValidaIn é ativada uma vez, a função InParaPos é ativada duas vezes e a função Calculo é ativada uma vez.

Arquivo com 1000 repetições:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
100.00      0.02     0.02     2000    10.00    10.00  InParaPos(std::__cxx11::basic_stri
  0.00      0.02     0.00   176000     0.00     0.00  prioridade(char)
  0.00      0.02     0.00    90000     0.00     0.00  float __gnu_cxx::__stoa<float, flo
  0.00      0.02     0.00    51000     0.00     0.00  void std::__cxx11::basic_string<ch
  0.00      0.02     0.00    49000     0.00     0.00  frame_dummy
  0.00      0.02     0.00    45000     0.00     0.00  std::__cxx11::basic_string<char, s
  0.00      0.02     0.00    44000     0.00     0.00  operacao(float, float, char)
  0.00      0.02     0.00     1000     0.00     0.00  Calculo(std::__cxx11::basic_string
  0.00      0.02     0.00     1000     0.00     0.00  ValidaIn(std::__cxx11::basic_strin
  0.00      0.02     0.00     1000     0.00     0.00  FilaArranjo<std::__cxx11::basic_st
  0.00      0.02     0.00        1     0.00     0.00  FilaArranjo<std::__cxx11::basic_st
```

```
~/tpgprof$ time ./a.out < entrada1.txt > saida1.txt

real    0m0.392s
user    0m0.096s
sys 0m0.050s
```

Arquivo com 50000 repetições:

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 44.55      0.45     0.45   100000     4.50     4.71  InParaPos(std::__cxx11:
 35.64      0.81     0.36    50000     7.20     8.80  ValidaIn(std::__cxx11::
  2.97      0.84     0.03  4500000     0.01     0.01  float __gnu_cxx::__stoa
  2.97      0.87     0.03  2550000     0.01     0.01  void std::__cxx11::basi
  2.97      0.90     0.03    50000     0.60     1.31  Calculo(std::__cxx11::b
  1.98      0.92     0.02  8800000     0.00     0.00  prioridade(char)
  1.98      0.94     0.02  2450000     0.01     0.01  frame_dummy
  1.98      0.96     0.02  2250000     0.01     0.02  std::__cxx11::basic_str
  1.98      0.98     0.02  2200000     0.01     0.01  operacao(float, float,
  1.98      1.00     0.02                             main
  0.99      1.01     0.01                             _init
  0.00      1.01     0.00    50000     0.00     0.00  FilaArranjo<std::__cxx1
  0.00      1.01     0.00        1     0.00     0.00  FilaArranjo<std::__cxx1
```

```
~/tpgprof$ time ./a.out < entrada1.txt > saida1.txt

real    0m19.129s
user    0m4.223s
sys 0m2.840s
```

É possível notar com os testes que o programa exerce as funcionalidades do resolvedor de expressão em um tempo curto, mesmo com um grande arquivo de entrada contendo várias expressões. No primeiro caso, a conversão de expressão posfixa para infixa foi o que demandou mais recursos do sistema, seguido pela validação da expressão e, por fim, o cálculo do resultado. No segundo caso, a conversão de expressão infixa para posfixa foi o que demandou mais recursos, seguido pela validação da expressão e, por fim, o cálculo do resultado. Ambos os casos tiveram tempo de execução parecidos, mesmo com o uso de funções diferentes. Apesar da expressão do segundo caso ser maior do que a do primeiro e de demandar o uso de um maior número de funções, o programa gastou um maior tempo realizando o primeiro caso, o que demonstra que quando a string de entrada é posfixa há uma maior demanda sobre o sistema.

6. Conclusão

Nesse trabalho, foi realizado um resolvedor de expressões numéricas através de um código na linguagem C++ utilizando estruturas de dados, disponíveis no próprio programa, que possuem funções e características específicas para este projeto. Dessa maneira, o processo de formulação do código proporcionou um maior entendimento acerca dos tipos abstratos de dados pilha e fila. Além disso, esse projeto demandou o uso de uma quantidade considerável de funções com um certo nível de dificuldade, o que exigiu o uso de ferramentas de depuração como o GDB e o valgrind. Outro componente exigido por esse trabalho foi a análise de complexidade, por meio de notações assintóticas, e a análise experimental, por meio de ferramentas que testam os desempenhos computacionais. A utilização de conceitos e ferramentas usados para compreender as estruturas de dados, as estratégias de depuração e as estratégias de análise experimental e de análise de complexidade foram ensinados nas aulas de Estrutura de Dados e, assim, esse trabalho prático foi essencial para fixar o conteúdo do curso e para aumentar a familiaridade com projetos de programação.

7. Bibliografia:

Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição

Slides virtuais da disciplina de Estrutura de Dados disponibilizados por Wagner Meira Jr.

cplusplus.com/reference/sstream/

Instruções para Compilação e Execução:

A compilação e execução é feita através do makefile.

Quando o usuário executar o comando “make” no terminal todos os arquivos são compilados e um executável é gerado na pasta bin. O makefile foi feito com a intenção de que o usuário insira um arquivo de texto de entrada chamado “entrada.txt” na pasta raiz TP. Dessa forma, ao executar o comando make, o código vai ser executado logo após a compilação usando esse arquivo de entrada e a saída será imprimida em outro arquivo de texto chamado “saída.txt”.

Se o usuário desejar executar o código manualmente pelo terminal, após utilizar o comando “make” ele deve digitar “./bin/resolvedor < entrada.txt” sendo que o arquivo entrada.txt deve estar dentro da pasta TP.