Universidade Federal de Minas Gerais

Departamento de Ciência da Computação

Trabalho Prático 1 – **Estrutura de Dados**

Daniel Nogueira Junqueira – 2021072244

danijnog@ufmg.br

1. Introdução

O seguinte programa tem por objetivo representar um resolvedor de expressão numérica, realizando as operações de ler, armazenar, converter e resolver expressões tanto infixa quanto posfixa. Para isso, é passado um arquivo com as expressões pela linha de comando e realizada as devidas operações com o auxílio de uma Estrutura de Dados definida ao longo do projeto.

2. Método

O programa foi desenvolvido e feito no WSL 2, com a distribuição Linux Ubuntu, utilizando a linguagem C. As especificações do computador são:

- Sistema Operacional em que está ativo o WSL: Windows 11.
- Processador: Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz 2.59 GHz
- RAM instalada: 16,0 GB.
- Tipo de sistema: 64 bits.

2.1 Estrutura de Dados e implementação

No programa, foi escolhida a Estrutura de Dados pilha para realizar o armazenamento, a conversão e a resolução das expressões.

Pelo fato da pilha ter a característica LIFO (Last in First out), o último elemento empilhado é o primeiro a ser desempilhado, isso facilita na conversão das expressões (tanto infixa para posfixa quanto posfixa para infixa) e na resolução da expressão posfixa, encaixando bem no trabalho.

Os métodos relacionados a pilha estão no arquivo *armazenaExp.h.*

Nas funções de conversão, tanto de posfixa para infixa quanto de infixa para posfixa é retornado um ponteiro de char mas toda a manipulação é feita através de um pilha, e no final da função esse ponteiro de char é armazenado e empilhado na pilha. Pela análise que realizei, ficaria mais intuitivo e com um código mais enxuto e mais fácil de entender feito dessa forma do que a função retornar a pilha.

Além disso, o programa resolve uma expressão apenas de uma forma: em seu formato posfixa, não resolvendo-a da forma infixa, já que ficou subentendido no enunciado do trabalho que deveria apenas resolver a expressão, não importando a forma.

O enunciado do TP diz que números pontos flutuantes deveriam estar separados por virgula (,), porém nas entradas que foram passadas como testes os números pontos flutuantes estavam separados por pontos (.) indicando a casa decimal, então a lógica no programa foi implementada considerando os números pontos flutuantes separados por pontos (.) levando em consideração o modelo de entradas que foi passado aos alunos.

2.2 Arquivos e funções

Os arquivos trabalhados estão divididos nas pastas ‘src’ e ‘include’. Na pasta ‘include’, estão os arquivos de cabeçalho que serão implementados de fato no respectivo arquivo na pasta ‘src’.

Os arquivos na pasta ‘include’ são os seguintes:

- *armazenaExp.h*
- *converteExp.h*
- *identificaExp.h*
- *resolveExp.h*
- *msgAssert.h*

Descrição das funções de cada **arquivo**:

***armazenaExp.h*:**

- função **validaEntrada ->** valida a entrada que é digitada na linha de comando do programa.

- função **armazenaExp** -> armazena uma expressão, utilizando para isso a Estrutura de Dados implementada no programa, que no caso foi uma pilha.

Além disso, nesse arquivo também contém as funções relacionadas a manipulação da Estrutura de Dados pilha, que são:

- **criaPilha**, **empilha**, **desempilha**, **limpaPilha** e **imprimePilha**.

converteExp.h:

- função **infixaParaPosfixa** -> converte uma expressão infixa para posfixa recebendo como parâmetro a expressão a ser convertida e uma pilha, armazenando a expressão convertida na pilha.
- função **posfixaParaInfixa** -> converte uma expressão posfixa para infixa recebendo como parâmetro a expressão a ser convertida e uma pilha, armazenando a expressão convertida na pilha.
- função **ehOperando** -> recebe um operando como parâmetro retornando 1 caso a função seja verdadeira.

identificaExp.h:

- função **identificaExpInfixa** -> recebe uma expressão como parâmetro e identifica se é infixa.
- função **identificaExpPosfixa** -> recebe uma expressão como parâmetro e identifica se é posfixa.

resolveExp.h:

- função **operacaoPosfixa** -> recebe como parâmetro o primeiro operando, o segundo operando e um operador, e realiza as devidas operações de acordo com o operador.
- função **resolvePosfixa** -> recebe como parâmetro a expressão a ser resolvida e uma pilha, com o intuito de resolver a expressão com o auxílio da pilha manipulando a mesma.

msgAssert.h:

- Arquivo relacionado para o lançamento e tratamento de exceções, já que na linguagem C não é possível fazer o tratamento de exceções com “try, throw, catch” como em outras linguagens.

3. Análise de Complexidade

Função **validaEntrada:** recebe como parâmetros o argc e argv e analisa apenas a entrada passada como parâmetro no argv. Não realiza nenhum loop, possuindo complexidade de tempo e espaço constante O(1);

Funções **criaPilha, empilha, desempilha e limpaPilha:** todas possuem complexidade de tempo e espaço constante: O(1);

Função **imprimePilha**: função para imprimir todos os itens que estão na pilha. Pelo fato de ter que percorrer a pilha e todos os itens em certa posição, possui complexidade de tempo O(n) e complexidade de espaço também O(n);

Função **armazenaExp**: função que recebe uma pilha e uma expressão como parâmetro, criando a pilha e percorrendo cada posição da expressão e empilhando na pilha. Por isso, possui complexidade de tempo e espaço O(n), onde ‘n’ é o tamanho da expressão.

Função **infixaParaPosfixa**: recebe uma expressão e uma pilha como parâmetro, onde percorre a expressão enquanto ainda não chega ao caracter nulo (\0) fazendo as devidas manipulações com a pilha para realizar a transformação. No final da função, ainda armazena o ponteiro de char retornado na pilha utilizando a função “armazenaExp”, que tem complexidade O(n). Então, a complexidade de tempo e espaço dessa função é O(n), onde ‘n’ é o tamanho da expressão.

Função **ehOperando:** recebe um ponteiro de char como parâmetro e retorna 1 caso seja um operando. Percorre o array de char em cada posição e por isso possui complexidade de tempo e espaço O(n), onde ‘n’ é o tamanho do array.

Função **posfixaParaInfixa:** recebe uma expressão e uma pilha como parâmetro, onde realiza um loop percorrendo a expressão, além de utilizar uma pilha auxiliar declarada como ponteiro de ponteiro. Dentro desse loop, é feito um outro loop devido a função ‘strtok’ em c, onde o objetivo é conseguir separar os caracteres pelos espaços, que forma um token na função, e poder empilhar esse token caso seja um operando, realizando a devida manipulação para que seja feita corretamente a conversão da expressão. Por isso, a complexidade de tempo é O(n^2) e a complexidade de espaço é O(n), onde ‘n’ é o tamanho da expressão.

Função **identificaExpInfixa:** recebe uma expressão com parâmetro, percorrendo a mesma até seu final para identificar se é uma expressão infixa válida. Possui complexidade de tempo e espaço O(n), onde ‘n’ é o tamanho da expressão.

Função **identificaExpPosfixa:** recebe uma expressão como parâmetro, percorrendo a mesma até seu final para identificar se é uma expressão posfixa válida. Possui complexidade de tempo e espaço O(n), onde ‘n’ é o tamanho da expressão.

Função **operacaoPosfixa:** recebe dois floats representando os números e um char pra representar o operador, fazendo a operação entre os números de acordo com o operador. Possui complexidade de tempo e espaço constante O(1).

Função **resolvePosfixa:** recebe uma expressão e uma pilha como parâmetro, com o objetivo de resolver a expressão em seu formato posfixa. Separa a expressão em tokens usando a função ‘strtok’ com o delimitador de espaço e possui complexidade de tempo e espaço O(n), onde ‘n’ é o tamanho da expressão.

Função **lerEntrada**: recebe como parâmetros o argc e argv, com o objetivo de ler o arquivo que for passado como parâmetro na linha de comando, e executar todas as operações necessárias em cada linha. Nesse arquivo, enquanto percorre o número de linhas do arquivo, é feito a chamada de outras funções já criadas para identificar a expressão, armazena-lá, transformar e resolver a expressão de acordo com a necessidade. Por isso, possui complexidade de tempo O(n^2) e complexidade de espaço O(n), onde ‘n’ é o número de linhas do arquivo.

4. Estratégias de robustez

Tratamento de erros: no programa o tratamento de erros é feito utilizando macros definidas no arquivo *msgAssert.h*, já que em C não é possível tratar exceções e erros de forma mais adequada como em C++. Além dessas macros, é utilizado também a função ‘fprintf’ para auxiliar o tratamento de erros, especificando a saída de erros padrão do programa e a mensagem retornada. Com isso, o programa fica mais simples e robusto para a identificação e o tratamento de exceções caso alguma venha a acontecer do que se fosse utilizado apenas a função ‘printf’ como é feito muitas vezes.

Memória: Ao rodar o Valgrind no programa, verifica-se que ocorre um vazamento de memória (não significativo) na função que converte uma expressão posfixa para infixa (posfixaParaInfixa). O que acontece é que, dentro de um loop que é feito para percorrer a expressão nessa função, é necessário alocar memória para uma variável dinamicamente e o problema é que não é possível desalocar essa memória dentro do loop para o bom funcionamento do código, se não a conversão não seria realizada corretamente. No fim da função, eu realizo outro loop para desalocar grande parte da memória que foi alocada, porém uma outra parte acaba ainda sendo perdida devido ao que acabei de citar.
Por isso, nesse momento, foi prezado a legibilidade, o bom funcionamento e a simplicidade do código em detrimento do pouco vazamento de memória que acontece. Vale ressaltar também que em nenhuma outra função ocorre vazamentos de memória.

```
==18930==
==18930== HEAP SUMMARY:
==18930==     in use at exit: 1,740 bytes in 10 blocks
==18930==   total heap usage: 43 allocs, 33 frees, 453,494 bytes allocated
==18930==
==18930== 1,740 bytes in 10 blocks are definitely lost in loss record 1 of 1
==18930==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-li
nux.so)
==18930==    by 0x10A74B: posfixaParaInfixa (converteExp.c:132)
==18930==    by 0x10987B: lerEntrada (main.c:82)
==18930==    by 0x1099DC: main (main.c:111)
==18930==
==18930== LEAK SUMMARY:
==18930==    definitely lost: 1,740 bytes in 10 blocks
==18930==    indirectly lost: 0 bytes in 0 blocks
==18930==      possibly lost: 0 bytes in 0 blocks
==18930==    still reachable: 0 bytes in 0 blocks
==18930==         suppressed: 0 bytes in 0 blocks
==18930==
```

Print da memória citada.

5. Análise experimental

Rodando o gprof para um arquivo com 31 linhas para resolver 8 expressões, obtemos o seguinte resultado:

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00     1929     0.00     0.00  empilha
  0.00      0.00     0.00     1261     0.00     0.00  ehOperando
  0.00      0.00     0.00       43     0.00     0.00  operacaoPosfixa
  0.00      0.00     0.00       42     0.00     0.00  desempilha
  0.00      0.00     0.00       21     0.00     0.00  armazenaExp
  0.00      0.00     0.00       21     0.00     0.00  criaPilha
  0.00      0.00     0.00       13     0.00     0.00  identificaExpInfixa
  0.00      0.00     0.00        8     0.00     0.00  limpaPilha
  0.00      0.00     0.00        8     0.00     0.00  resolvePosfixa
  0.00      0.00     0.00        5     0.00     0.00  identificaExpPosfixa
  0.00      0.00     0.00        5     0.00     0.00  posfixaParaInfixa
  0.00      0.00     0.00        3     0.00     0.00  infixaParaPosfixa
  0.00      0.00     0.00        1     0.00     0.00  lerEntrada
  0.00      0.00     0.00        1     0.00     0.00  validaEntrada
```

Aumentando o arquivo para 63 linhas com 16 expressões:

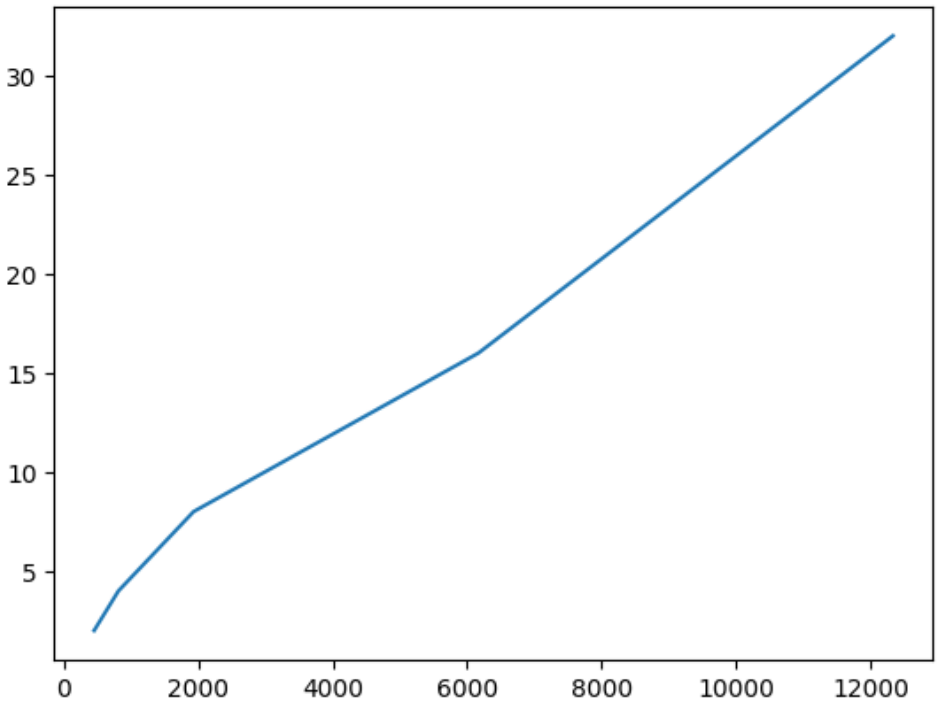
```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00     6169     0.00     0.00  empilha
  0.00      0.00     0.00     3913     0.00     0.00  ehOperando
  0.00      0.00     0.00      143     0.00     0.00  operacaoPosfixa
  0.00      0.00     0.00      104     0.00     0.00  desempilha
  0.00      0.00     0.00       43     0.00     0.00  armazenaExp
  0.00      0.00     0.00       43     0.00     0.00  criaPilha
  0.00      0.00     0.00       27     0.00     0.00  identificaExpInfixa
  0.00      0.00     0.00       16     0.00     0.00  limpaPilha
  0.00      0.00     0.00       16     0.00     0.00  resolvePosfixa
  0.00      0.00     0.00       11     0.00     0.00  identificaExpPosfixa
  0.00      0.00     0.00       11     0.00     0.00  posfixaParaInfixa
  0.00      0.00     0.00        5     0.00     0.00  infixaParaPosfixa
  0.00      0.00     0.00        1     0.00     0.00  lerEntrada
  0.00      0.00     0.00        1     0.00     0.00  validaEntrada
```

Com isso percebemos que a função ‘empilha’ e ‘ehOperando’ é chamada constantemente no programa, principalmente quando aumentamos o tamanho da entrada, porém apesar de serem chamadas bastante elas não utilizam uma porcentagem significativa do tempo do programa.

O gráfico abaixo é referente a chamada da função “empilha” a medida que o tamanho da entrada no arquivo cresce – no caso uma expressão conta como 1 entrada, 2 expressões como 2 entradas, etc.



Eixo x -> número de chamadas do método “empilha”

Eixo y -> tamanho da entrada do arquivo

O crescimento é praticamente linear.

A função empilha é chamada constantemente devido a um loop feito para empilhar uma expressão, sendo composta por caracteres, espaços, operandos e parênteses (neste caso se for infixa).

6. Conclusões

Portanto, basicamente, o trabalho se constitui de um resolvedor de expressão numérica, que lê, armazena, converte (expressão infixa para posfixa, e posfixa para infixa) e resolve a expressão em seu formato posfixa.

O trabalho foi interessante e proveitoso por que todas essas funcionalidades envolvem a manipulação de uma Estrutura de Dados de sua preferência (nesse programa foi implementada a pilha, porém tenho consciência que outras Estruturas de Dados também poderiam ser usadas, como árvores) para que sejam possíveis implementar os requisitos pedidos no trabalho. Acredito que o que mais foi trabalhoso foi implementar as conversões, pois, apesar de parecer ser uma lógica simples, temos que pensar que estamos trabalhando com pontos flutuantes (o que dificultou bem o projeto) e o algoritmo tem de ser, preferencialmente, escalável, já que podemos receber expressões bem grandes na entrada do programa para trabalhar em cima.

Com isso, considero que o conteúdo proposto do trabalho foi bom e estava alcançável, ao mesmo tempo em que desafia e tira os alunos da zona de conforto, o que também é bom.

7. Bibliografia

Slides e vídeos sobre Pilhas que está no moodle da disciplina.

Slides sobre Análise de complexidade

https://homepages.dcc.ufmg.br/~cunha/teaching/20121/aeds2/complexity.pdf

8. Instruções para compilação e execução

No diretório do programa, de o comando ‘make run’ para compilar todos os arquivos.
Com isso, o executável ‘main’ será gerado na pasta ‘bin’.

Para executar o ‘main’ junto com o arquivo com as expressões a ser testada, de o comando ‘bin/main nome_do_arquivo.in’, onde ‘nome_do_arquivo.in’ deve ser o nome de seu arquivo.

Coloque seu arquivo preferencialmente na mesma pasta do projeto para ser mais simples, se não é necessário especificar o caminho do arquivo.

Por exemplo se o arquivo estiver a um diretório antes do diretório atual da pasta do programa: ‘bin/main ../nome_do_arquivo.in’

Atualmente na pasta do programa, já existe um arquivo de entrada nomeado ‘entrada.in’ que pode ser utilizado também para testar o programa, caso deseje.