Trabalho Prático 1
Resolvedor de Expressão Numérica

Giovanna Naves Ribeiro
Matrícula: 2022043647
Data: 25/04/2023

1. Introdução:

O trabalho descrito a seguir tem como objetivo resolver expressões numéricas fornecidas em diferentes formatos, assim como transformar as expressões de um formato para o outro. A proposta da atividade é simular um programa para ser utilizado em uma sala de aula e abordar estruturas de dados na resolução das operações pedidas.
Para isso, resolvi organizar o código de maneira a empregar as seguintes estruturas de dados para facilitar a montagem das operações:

2. Método:

- Linguagem utilizada: c++
- Especificações do computador utilizado:
    - Sistema Operacional: Ubuntu 22.04.1 LTS
    - Processador: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz 2.30 GHz
    - RAM instalada: 8,00 GB (utilizável: 7,79 GB)

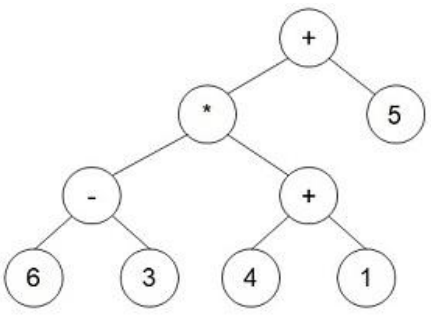
2.1. Estruturas de Dados

A estrutura de dados escolhida para armazenar as expressões numéricas foi a **árvore**. Utilizamos também a **pilha** para nos auxiliar em alguns algoritmos, que serão explicados no tópico 2.3.

2.2. Classes

O programa utiliza classes para a implementação das estruturas de dados, seguindo os princípios da programação orientada a objetos.

A classe **ArvoreExp** é utilizada para implementar uma árvore dinamicamente, cujos nós carregam strings, que serão posteriormente convertidas em double para a resolução das expressões. Segue um exemplo de expressão numérica implementada em uma árvore:



A classe **Funcoes** carrega apenas a função TransformaEmPosfixo, que utilizamos para transformar expressões infixas em pósfixas (utilizando uma pilha) para só então construirmos a árvore.

A classe **Pilha** implementa uma pilha que será usada tanto para a conversão de expressões infixas para pósfixas quanto para a resolução de uma expressão armazenada em árvore.

2.3. Funções e algoritmos

A resolução das expressões numéricas será feita apenas a partir da forma posfixa, portanto, as expressões lidas do arquivo de entrada na forma infixa serão transformadas em pósfixas por meio da utilização do Algoritmo de Shunting Yard (ou Pátio de Manobras), implementado na função **TransformaEmPosfixa**, que remove os parênteses e vai reorganizando os operadores pós fixados aos operandos a que eles se aplicam. Esse método utiliza uma pilha para transformar a expressão infixa em uma string com a expressão posfixa.

Observação: Para saber se uma expressão é infixa e, portanto, chamar a função TransformaEmPosfixa, identificamos se a expressão numérica se inicia com um parênteses de abertura.

Com a expressão pósfixa (seja ela lida diretamente do arquivo ou resultado da conversão de uma expressão infixa), chamamos a função **ConstruirArvore** da classe ArvoreExp e utilizaremos a estrutura de árvore para armazenar a expressão denominada “exp”.

Se o arquivo de entrada pede uma conversão de INFIXA para POSFIXA, como já o fizemos utilizando a pilha, apenas imprimiremos a expressão armazenada em “exp”. Caso a conversão pedida seja de POSFIXA para INFIXA, precisaremos ler a árvore utilizando o caminhamento que retorna uma expressão infixa. Para isso, chamamos a função recursiva **PrintarEmOrdem**, da classe ArvoreExp, que passa a raiz da árvore como parâmetro.

Enfim, devemos resolver a expressão e, para isso, chamaremos a função recursiva **Resolver**, também da classe ArvoreExp. Dentro dessa função, os operandos armazenados como strings na árvore são convertidos para double e sofrem as operações chamadas pelos operadores.

3. Análise de Complexidade:

Para analisar a complexidade assintótica do programa, devemos somar as complexidades das principais funções apresentadas:

- A função **TransformaEmPosfixa** tem complexidade **O(n)** onde n é o tamanho da string de entrada (a expressão INFIXA) passada para a função.
- A função **ConstruirArvore** tem complexidade **O(n)**, onde n é o tamanho da string de entrada (a expressão POSFIXA) passada para a função.

- A função **PrintarEmOrdem** tem complexidade **O(n)**, onde n é o número de nós da árvore cuja raiz foi passada para a função.
- A função **Resolver** tem complexidade **O(n)**, onde n é também o número de nós da árvore cuja raiz foi passada para a função.

Portanto, podemos concluir que o programa tem a complexidade na ordem de **O(n)**, sendo n o tamanho da string de entrada.

4. Estratégias de Robustez:

O primeiro cuidado que tomamos no código em relação a possíveis falhas é conferir se foi possível encontrar e abrir o arquivo de entrada. Caso haja algum erro, o informamos ao usuário.

```
int main(int argc, char *argv[])
{
    // Se não der nome do arquivo -> erro
    if (argc < 2)
    {
        std::cerr << "Erro: Nenhum arquivo encontrado" << std::endl;
        return 1;
    }
```

```
// Checa se abriu
if (!infile.is_open())
{
    std::cout << "Erro: não foi possível abrir o arquivo" << filename << std::endl;
    return 1;
}
```

Em seguida, conferimos se o tipo da expressão dito pela entrada realmente é o tipo correto. Caso a expressão seja dita INFIXA pela entrada mas não o seja realmente devido à ausência de parênteses (e o contrário para as posfixas), o programa retorna "Expressão não válida: tipo da expressão não correspondente".

Além disso, a função **TransformaEmPosfixo** verifica se cada parêntese de abertura na expressão infixa tem seu parêntese de fechamento correspondente. Caso contrário, retorna "Expressão não válida: parênteses não correspondentes".

5. Análise Experimental:

Utilizamos a ferramenta **gprof** para realizar a análise experimental do código. Podemos observar a seguir, o relatório gerado para a transformação de uma expressão INFIXA para POSFIXA e sua resolução. Nele, observamos que o tempo de execução do código é tão curto que mesmo quando chamamos, no arquivo de entrada, quatro expressões diferentes para serem convertidas e resolvidas, a duração não é grande o suficiente para ser mostrada pelo relatório do gprof.

```
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     6886     0.00     0.00  bool std::operator==<char, std::char_traits<char>,
__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, char const*)
  0.00      0.00     0.00     1401     0.00     0.00  Pilha<std::__cxx11::basic_string<char, std::char_t
  0.00      0.00     0.00     1121     0.00     0.00  Pilha<std::__cxx11::basic_string<char, std::char_t
  0.00      0.00     0.00      833     0.00     0.00  bool std::operator!=<char, std::char_traits<char>,
__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&, char const*)
  0.00      0.00     0.00      596     0.00     0.00  precedencia(std::__cxx11::basic_string<char, std::
  0.00      0.00     0.00      424     0.00     0.00  Pilha<std::__cxx11::basic_string<char, std::char_t
  0.00      0.00     0.00      424     0.00     0.00  Pilha<std::__cxx11::basic_string<char, std::char_t
__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >)
  0.00      0.00     0.00      314     0.00     0.00  isOperator(std::__cxx11::basic_string<char, std::c
  0.00      0.00     0.00      314     0.00     0.00  No::No(std::__cxx11::basic_string<char, std::char_
No*)
  0.00      0.00     0.00      314     0.00     0.00  Pilha<No*>::Desempilha()
  0.00      0.00     0.00      314     0.00     0.00  Pilha<No*>::Empilha(No*)
  0.00      0.00     0.00      159     0.00     0.00  double __gnu_cxx::__stoa<double, double, char>(dou
const*, unsigned long*)
  0.00      0.00     0.00      159     0.00     0.00  std::__cxx11::stod(std::__cxx11::basic_string<char
const&, unsigned long*)
  0.00      0.00     0.00      159     0.00     0.00  std::__cxx11::basic_string<char, std::char_traits<
std::char_traits<char>, std::allocator<char> >(std::__cxx11::basic_string<char, std::char_traits<char>,
  0.00      0.00     0.00      159     0.00     0.00  __gnu_cxx::__stoa<double, double, char>(double (*
unsigned long*)::_Range_chk::_S_chk(double, std::integral_constant<bool, false>)
  0.00      0.00     0.00      159     0.00     0.00  __gnu_cxx::__stoa<double, double, char>(double (*
unsigned long*)::_Save_errno::_Save_errno()
  0.00      0.00     0.00      159     0.00     0.00  __gnu_cxx::__stoa<double, double, char>(double (*
unsigned long*)::_Save_errno::~_Save_errno()
```

Como o gprof não nos mostra com clareza o tempo exato de execução, utilizamos também a função **clock_gettime()**, da biblioteca <sys/time.h>, para calcular o tempo de sistema, de usuário e de relógio. Calculamos esses tempos em milissegundos passando na entrada as mesmas 4 expressões dos testes com o gprof, e obtivemos 0.000000, 0.087000 e 0.116980 milissegundos respectivamente.

6. Conclusões:

Concluímos que o programa converte com sucesso expressões infixas para posfixas e vice e versa, assim como as resolve em um tempo extremamente curto, se mostrando eficiente. Além disso, ele trata exceções e as informa ao usuário auxiliando na utilização correta do programa.

Podemos também afirmar que a utilização da função de medição de tempo da biblioteca sys/time.h foi mais útil que o gprof para calcular o tempo de execução do programa pois nos forneceram dados mais tangíveis.

A construção do código contribuiu consideravelmente para o aprendizado da aplicação de classes, pilhas e árvores, e se mostrou desafiador principalmente nas partes envolvendo alocação de memória.

7. Bibliografia:

Gisele L Pappa e Wagner Meira Jr. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

“C++ Solution Using Shunting-Yard Algorithm. Expandable to Include Other Operators - Basic Calculator II.” *LeetCode*, https://leetcode.com/problems/basic-calculator-ii/solutions/168822/c++-solution-using-shunting-yard-algorithm.-Expandable-to-include-other-operators.

8. Instruções para compilação e execução:

Para compilar o programa, o primeiro a se fazer é abrir a pasta TP e copiar o conteúdo do arquivo de entrada para o arquivo "**entrada.txt**" que se encontra na raiz do projeto, pois é ele que é lido no Makefile do programa.

Enfim, deve-se executar o comando "**make run**" no terminal para que o programa funcione. Recomenda-se ainda que, antes de cada "make run", seja executado um "**make clean**".