TRABALHO PRÁTICO 1

Resolvedor de Expressão Numérica

Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Estrutura de Dados - 2023/01 - Turma TN
Belo Horizonte - MG - Brasil

Nome: José Eduardo Duarte Massucato **Matrícula:** 2022043620
josedmass@ufmg.br

1. Introdução

O trabalho proposto tem por objetivo lidar com expressões numéricas, denotadas em notação infixa ou pós-fixa, que utilizam das quatro operações básicas da matemática: soma (+), subtração (-), divisão (/) e multiplicação (*). O programa consegue ler expressões com o comando “LER”, converter de uma notação para a outra com o comando “INFIXA” - caso o usuário queira a expressão na notação infixa - ou “POSFIXA” - caso o usuário a queira na notação pós-fixa - e resolver a expressão com o comando “RESOLVE”. Para que as expressões sejam adequadamente armazenadas, o usuário **deve** separar cada termo com um espaço, conforme descrito no enunciado do Trabalho Prático. Diante dessa perspectiva, o algoritmo foi desenvolvido na linguagem C++, utilizando-se das estruturas de dados apropriadas para lidar com essas notações.

2. Método

2.1. Estrutura de Dados

Duas estruturas de dados foram utilizadas no programa: uma árvore binária e uma pilha encadeada. A **árvore binária** foi escolhida para armazenar a expressão em notação infixa, pois, por definição, a árvore consegue representar bem a prioridade dos seus elementos, através da ancestralidade entre os nós. Isso significa que, nessa estrutura, quanto mais profundo e mais à esquerda um elemento está na árvore, maior a sua prioridade na expressão.

Por outro lado, a **pilha encadeada** foi escolhida para armazenar a expressão em notação pós-fixa, uma vez que, nesta notação, a prioridade de uma operação se dá pela ordem de entrada dos elementos, fato que a pilha consegue lidar muito bem. Por decisão de projeto, optei por deixar a pilha “de cabeça para baixo” - isto é, os primeiros elementos da expressão estão mais ao topo da pilha -, pois, dessa forma, a manipulação da expressão fica mais fácil, como será explicado posteriormente. Isso significa que quanto mais próximo do topo da pilha um elemento está armazenado, maior a sua prioridade na expressão.

2.2. Classes

2.2.1. `class` **`Celula`**

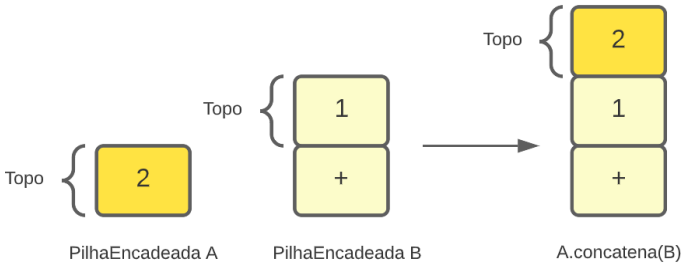
Possui como **variáveis privadas** `std::string item`, que guarda o valor armazenado nesta célula, e `Calula* prox`, que aponta para a próxima célula. Possui como **funções públicas**:

- `Celula()`: constrói uma célula vazia, isto é, `item` recebe uma *string* vazia e `prox` aponta para *NULL*.

2.2.2. `class` **`PilhaEncadeada`**

Possui como **variáveis privadas** `int tamanho`, que armazena o número atual de itens contidos na pilha, e `Celula* topo`, que aponta para o topo da pilha. Possui como **funções públicas:**

- `PilhaEncadeada()`: constrói uma pilha vazia, isto é, com tamanho igual a 0 e com topo apontando para *NULL*.
- `PilhaEncadeada& operator=(const PilhaEncadeada& other)`: descreve o comportamento desejado para atribuição de duas pilhas. Ao final, a pilha `*this` fica com o mesmo topo, mesmo tamanho e mesmos itens que a pilha `other`.
- `Celula* GetTopo()` é uma função auxiliar que retorna o topo da pilha, enquanto que `int GetTamanho()` é uma função auxiliar que retorna o tamanho atual da pilha. Ambas essas funções foram criadas com o intuito de garantir o encapsulamento do algoritmo.
- `void Empilha_Exp(std::string exp[], int end)`: insere na pilha uma expressão na notação pós-fixa, armazenada no vetor de *strings* `exp`, iterativamente. Esse vetor é percorrido de trás para frente, para que a pilha fique de "cabeça para baixo", como mencionado anteriormente.
- `void Empilha_Item(std::string s)`: insere no topo pilha um único elemento com valor igual a `s`.
- `void concatena(PilhaEncadeada& other)`: concatena duas pilhas, de forma que a pilha `*this` fique sobre a pilha `other`. Ao final, deletamos o topo da pilha `other`, uma vez que ele se torna desnecessário. A imagem abaixo ilustra essa operação.



- `std::string Desempilha()`: deleta o atual `topo` da pilha, retornando seu item, fazendo com que ele passe a ser igual à `topo->prox` e então diminuindo o tamanho da pilha em uma unidade.

- `std::string Percorre()`: percorre a pilha, a partir do topo, retornando todos os seus elementos em uma única *string*, sem alterar sua configuração. Nessa função, destaca-se a utilidade de eu ter optado por armazenar a expressão "de cabeça para baixo", pois, para imprimir, basta chamar a função `Percorre()`.
- `void Limpa()`: limpa a pilha iterativamente e, ao final, faz com que `topo` aponte para *NULL*.
- `~PilhaEncadeada()`: destrói a pilha, chamando a função `Limpa()`.

2.2.3. `class` **`No`**

Possui como **variáveis privadas** `std::string item`, que guarda o valor armazenado nesse nó, `No* esq` e `No* dir`, que apontam, respectivamente, para os filhos da esquerda e da direita do nó. Possui como **funções públicas**:

- `No()`: constrói um nó vazio, isto é, `item` recebe uma *string* vazia e tanto `esq` quanto `dir` apontam para *NULL*.
- `No(std::string valor)`: constrói um nó com `item` igual à `valor` e tanto `esq` quando `dir` apontando para *NULL*.

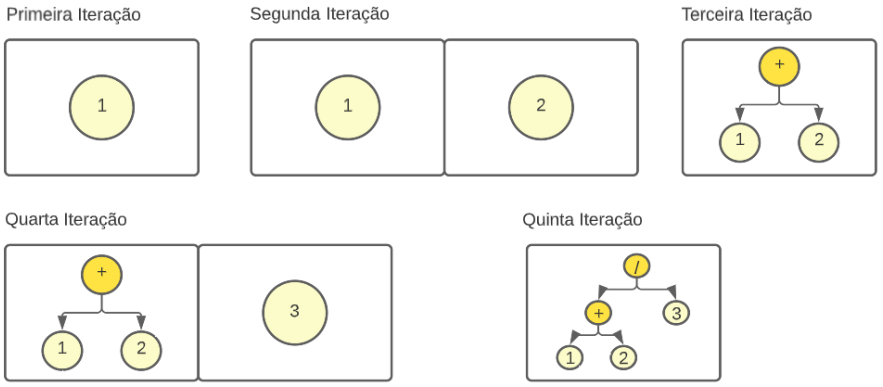
2.2.4. `class` **`ArvoreBinaria`**

Possui como **variável privada** `No* raiz`, que aponta para a raiz da árvore. Possui como **funções públicas**:

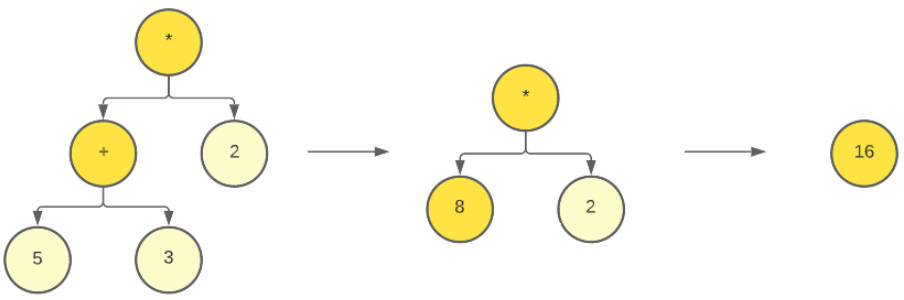
- `ArvoreBinaria()`: constrói uma árvore com `raiz` apontando para *NULL*.
- `ArvoreBinaria& operator=(const ArvoreBinaria& other)`: descreve o comportamento desejado para atribuição de duas árvores. Ao final, a árvore `*this` fica com a mesma raiz e os mesmos filhos que a pilha `other`. Isso é realizado a partir do comando `CopiaArvore(this->raiz, other.raiz)`, que será apresentado em sequência.
- `void CopiaArvore(No* &dessa, No* other)`: é uma função que utiliza da recursividade para fazer com que todos os elementos `dessa->esq` e `dessa->dir` sejam respectivamente iguais à `other->esq` e `other->dir`. Ao final, temos que a árvore `dessa` fica igual à `other`.
- `No* GetRaiz()`: é uma função auxiliar que retorna a raiz da árvore. Essa função foi criada para garantir o encapsulamento do algoritmo.
- `No* InsereRecursivo(std::string exp[], int start, int end)`: essa função utiliza da recursão para inserir na árvore uma expressão em notação infixa, armazenada no vetor de *strings* `exp`, cujo índice de início é `start` e o de fim é `end`. Algumas métricas, como `int min_precedencia` (que analisa a precedência dos operadores) e `int cont_parenteses` (que calcula a proporção de parênteses) foram criadas para descobrir onde está localizada a menor precedência da expressão. Uma vez localizado em que índice (`int indice_raiz`) de `exp` está essa menor precedência, fazemos com que `raiz = No* raiz_exp = new No(exp[indice_raiz])` e então quebramos a expressão para o que está à esquerda desse índice e para o que está a direita. Ou seja, chamamos,

respectivamente, `InsereRecursivo(exp, start, indice_raiz - 1)` e `InsereRecursivo(exp, indice_raiz + 1, end)`. Isso faz com que o que seja de menor prioridade fique “mais superficialmente" na árvore, conforme descrito na **seção 2.1**.

- `void Insere_Da_Pilha(std::string exp, int end)`: o vetor de *strings* `exp` contém uma expressão numérica na notação pós-fixa. Essa função, iterativamente, constrói uma árvore a partir desse vetor, cujo índice do último elemento é `end`. A imagem a seguir ilustra como a expressão “1 2 + 3 /” seria armazenada.



- `PilhaEncadeada PosOrdem(No* p)`: essa função faz um percorrimento pós-ordem de árvores binárias, com o intuito de transformar a árvore armazenada em uma pilha encadeada. Os comandos `concatena(dir)` e `concatena(esq)` são utilizados para que consigamos acumular uma pilha resultante. Note que o que está mais profundo e mais à esquerda na árvore é empilhado primeiro, ou seja, essa pilha não está “de cabeça pra baixo”, como acordado na **seção 2.1**. Logo, é função de quem a chama revertê-la.
- `std::string InOrdem(No* p, bool primeiroNo)`: utiliza do percorrimento in-ordem de árvores binárias, com o intuito de retornar uma *string* que representa a expressão que está armazenada na árvore, em notação infixa. `p` é o nó onde estamos na iteração, enquanto que `primeiroNo` checa se estamos aprofundando ou subindo na árvore; se aprofundando, adiciona “(“ na *string* resultante e, se subindo, adiciona “)”.
- `double resolve(No* p)`: retorna o resultado da expressão armazenada na árvore. Essa função utiliza da recursividade para que sejam resolvidos os termos mais profundos e mais à esquerda da árvore. A imagem a seguir ilustra como a expressão “(5 + 3) * 2” seria resolvida.



- `void LimpaRecursivo(No* p)`: deleta, recursivamente, todos os elementos da árvore.
- `void Limpa()`: chama `LimpaRecursivo(raiz)` e depois faz `raiz = NULL`. Essa função fez-se necessária à medida que, no meu algoritmo, uma árvore vazia precisa ter a raiz apontando para *NULL.*
- `~ArvoreBinaria()`: destrói a árvore, chamando a função `Limpa()`.

2.2.5. `class Expressao`

Possui como **variáveis:** `ArvoreBinaria A` e `PilhaEncadeada P`. Essa classe tem o objetivo de armazenar as expressões em `A`, caso esteja na notação infixa ou em `P`, caso esteja na pós-fixa. Possui como **funções:**

- `Expressao()`: constrói um armazenamento vazio, isto é, com uma árvore e uma pilha vazias.
- `Expressao& operator=(const Expressao& other)`: descreve o comportamento desejado para atribuição de duas expressões. Ao final, `this->A` fica igual à `other->A`, enquanto que `this->P` fica igual à `other->P`.
- `~Expressao()`: destrói o armazenamento, chamando `A.Limpa()` e `P.Limpa()`.

2.3. Funções

Em *TP/src/main.cpp* leio, linha a linha, por meio do comando `std::getline`, o conteúdo do arquivo, até o seu fim. Então, passo essa linha para uma variável do tipo *std::istringstream,* para que eu consiga depois separar facilmente os elementos pelos espaços.

2.3.1. `std::string imprime_infx(ArvoreBinaria& A)`

É uma função auxiliar cujo objetivo é retornar uma *string,* que representa a expressão numérica armazenada, em notação infixa. Ela coloca um "(" inicial, chama `A.InOrdem()` e finaliza com um ")" final. Essa função retorna uma *string* parecida com "(((5) + (3)) * (2))", ou seja, com o máximo de parênteses possível.

2.3.2. `std::string imprime_posfx(PilhaEncadeada& P)`

É uma função auxiliar cujo objetivo é retornar uma *string*, que representa a expressão numérica armazenada, em notação pós-fixa. Ela faz isso chamando `P.Percorre()`, função descrita na **seção 2.2.2**.

2.3.3. `bool valido_infx(std::string s[], int tam)`

Checa se uma expressão em notação infixa, armazenada no vetor de *strings* `s`, é válida. Entende-se que uma expressão na notação infixa é válida quando ela atende a todas as seguintes condições:

1. Todo parêntese que for aberto é fechado;
2. Não possui dois números seguidos;
3. Não possui dois operadores seguidos;
4. Só possui como elementos: parênteses, números e/ou operadores.

Observação: uma calculadora comum não considera parênteses que são abertos e não são fechados um problema. Entretanto, no enunciado do TP consta que devemos nos imaginar implementando um algoritmo para o ensino fundamental, que está aprendendo a resolver expressões algébricas. Diante dessa perspectiva, considerei necessário que todo parêntese aberto deve ser fechado, para incentivar a escrita correta das expressões pelos usuários.

2.3.4. `bool valido_posfx(std::string s[], int tam)`

Checa se uma expressão em notação pós-fixa, armazenada no vetor de *strings* `s`, é válida. Entende-se que uma expressão na notação pós-fixa é válida quando ela atende a todas as seguintes condições.

1. Se, lendo da esquerda para a direita, em nenhum momento o número de operadores se iguala ou fica maior que o de operandos;
2. Se, no final, a quantidade de números for uma unidade a mais que a de operadores;
3. Só possui como elementos: números e/ou operadores.

2.3.5. `void ler(std::string s[], int tam, Expressao& exp, std::string notacao)`

É chamada com o comando “LER”. Essa função verifica se a expressão, armazenada em `s`, em notação `notacao`, é válida e, em caso afirmativo, a armazena. Caso `notacao` seja diferente de “INFIXA” ou de “POSFIXA”, a exceção `NOTACAO_INVALIDA` é lançada. Por outro lado, caso a expressão não seja válida, a exceção `EXPRESSAO_INVALIDA` é lançada.

2.3.6. `std::string converte_infixa(Expressao& exp)`

É chamada com o comando “INFIXA”. Essa função converte a expressão armazenada para uma árvore binária e retorna a expressão em notação infixa. Ela faz isso por meio de `exp.A.Insere_Da_Pilha`, descrita na **seção 2.2.4**. Caso não haja nada armazenado em `exp`, então a exceção `ARMAZENAMENTO_VAZIO` é lançada.

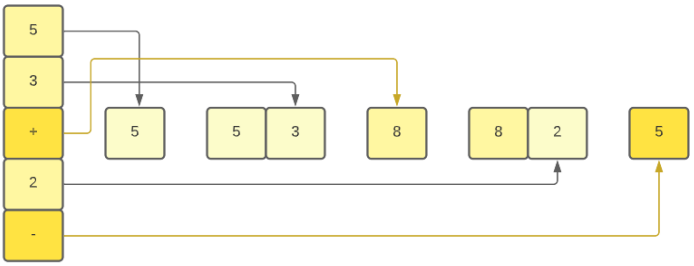
2.3.7. `std::string converte_posfixa(Expressao& exp)`

É chamada com o comando “POSFIXA”. Essa função converte a expressão armazenada para uma pilha encadeada e retorna a expressão em notação pós-fixa. Ela faz isso chamando `ArvoreBinaria::PosOrdem`, descrita na **seção 2.2.4**, colocando o resultado disso em uma pilha encadeada auxiliar e, então, invertendo essa pilha, chegando a uma pilha resultante, que está de acordo com a estrutura de

dados apresentada na **seção 2.1**. Caso não haja nada armazenado em `exp`, então a exceção `ARMAZENAMENTO_VAZIO` é lançada.

2.3.8. `double resolve (Expressao& exp)`

É chamada com o comando "RESOLVE". Caso haja uma pilha armazenada (notação pós-fixa), então ele vai desempilhando-a em uma pilha auxiliar e, ao encontrar um operador, realiza a operação com os dois termos mais ao topo da pilha auxiliar. A imagem abaixo ilustra essa iteração para a expressão "5 3 + 2 -".



Aqui fica evidente o porquê da decisão da pilha ficar de "cabeça para baixo", uma vez que fica mais fácil de resolver a expressão. Por outro lado, caso haja uma árvore binária armazenada (notação infixa), basta realizar `exp.A.resolve`, descrita na **seção 2.2.4**. Se não houver nada armazenado em `exp`, então a exceção `ARMAZENAMENTO_VAZIO` é lançada. Se, durante a realização da operação divisão (/), o denominador for igual a zero, então a exceção `DIVISAO_POR_ZERO` é lançada.

3. Análise de Complexidade

3.1. Comando "LER"

Chama a função `ler`, descrita na **seção 2.3.5**. Se a notação for "INFIXA", então a complexidade de tempo é limitada superiormente pela chamada da função `InsereRecursivo`, cuja complexidade de tempo é dada por:

$$T(n) = n + 2T(\tfrac{n}{2}) = \Theta(n \log n) \quad (Teorema\ Mestre)$$

Por outro lado, caso a notação seja "POSFIXA", então a complexidade de tempo é limitada superiormente pela chamada da função `EmpilhaExp`, cuja complexidade de tempo é:

$$O(n) \times O(1) = O(max(n, 1)) = O(n)$$

Ou seja, a chamada da função `ler` gera, no pior caso, uma complexidade de tempo igual a $O(n \log n)$. Já em relação à complexidade de espaço, temos que, para qualquer notação, ela é limitada por $O(n)$.

3.2. Comando "INFIXA"

Chama a função `converte_infixa`, descrita na **seção 2.3.6**. No pior caso, temos uma pilha encadeada armazenada e a convertemos para uma árvore binária. Nesse caso,

chamamos apenas funções iterativas, que realizam um número de operações relacionado ao tamanho da pilha armazenada. Logo, a complexidade de tempo é $O(n)$. Em relação a complexidade de espaço, ela é igual a $O(n)$, que é o tamanho da pilha que estamos transformando em árvore.

3.3. Comando “POSFIXA”

Chama a função `converte_posfixa`, descrita na **seção 2.3.7**. O pior caso é quando temos uma árvore binária armazenada e a convertemos para uma pilha encadeada. Nesse caso, chamamos a função recursiva `ArvoreBinaria::PosOrdem`, cuja complexidade de tempo é dada por:

$$T(n) = n + 2T(\frac{n}{2}) = \Theta(n \log n)$$

Logo, a complexidade de tempo final é limitada por $O(n \log n)$, enquanto que a complexidade de espaço é dada por $O(n)$, pois depende apenas do tamanho da árvore armazenada.

3.4. Comando “RESOLVE”

Se uma pilha encadeada estiver armazenada, então a complexidade de tempo é igual à de espaço, que é $O(n)$, pois trata-se de uma operação iterativa. Se for uma árvore, então temos complexidade de tempo dada por $T(n) = 1 + 2T(\frac{n}{2}) = \Theta(n) \therefore O(n)$, e de espaço igual a $O(h)$, onde `h` é a altura da árvore armazenada, devido à recursão de `ArvoreBinaria::resolve`.

4. Estratégias de Robustez

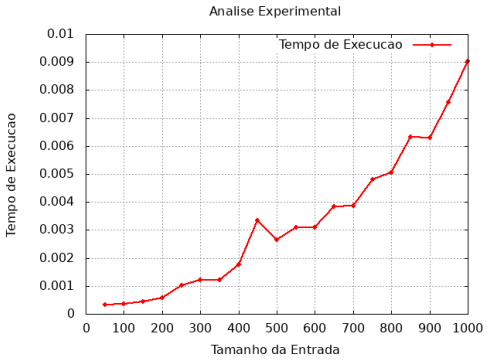
A fim de tornar o algoritmo mais robusto, tratamentos de exceções para além dos exigidos no enunciado do TP foram implementados, contidos em TP/include/Excecoes.h.

- `EXPRESSAO_INVALIDA`: É lançada caso o usuário escreva uma expressão que não seja válida. No caso das notações infixas, eu considerei irrelevante abrir e fechar parênteses vazios, isto é, “parênteses desnecessários”. Por exemplo, a expressão “( ) 5 + 3 ( )” é considerada válida.
- `COMANDO_INVALIDO`: É lançada caso o usuário escreva um comando diferente de “LER”, “INFIXA”, “POSFIXA” ou “RESOLVE”. No meu algoritmo, esses comandos devem ser escritos **exatamente** assim.
- `NOTACAO_INVALIDA`: É lançada caso o usuário chame o comando “LER”, e em seguida escreva algo diferente de “INFIXA” ou “POSFIXA”.
- `ARMAZENAMENTO_VAZIO`: É lançada caso o usuário esteja tentando realizar alguma operação sem antes ter armazenado uma expressão válida.
- `DIVISAO_POR_ZERO`: É lançada caso, ao chamar o comando “RESOLVE”, em algum momento da operação ocorrer uma divisão por zero, que não existe na matemática e poderia resultar em comportamentos indeterminados no programa.

Além disso, caso o usuário não coloque o nome do arquivo no momento da compilação, como será descrito na **seção 8**, um erro é imprimido na tela. Além disso, se o arquivo não abriu adequadamente, um erro também é imprimido na tela.

5. Análise Experimental

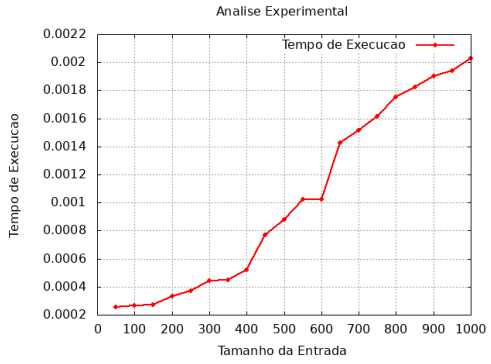
Com base nos exemplos de entradas fornecidos no Moodle da disciplina, considerei dois casos de arquivos para a análise experimental do algoritmo:



5.1. **`LER` `INFIXA EXP`, `POSFIXA`, `RESOLVE`:** com base na análise de complexidade descrita na **seção 3**, temos que essa sequência tem a seguinte complexidade de tempo:

$$O(n \log n) + O(n \log n) + O(n) = O(max(n \log n, n)) = O(n \log n).$$

O gráfico à esquerda, construído a partir da ferramenta *gnuplot*, representa essa sequência de comandos, para entradas de diferentes tamanhos. Note que, de fato, o gráfico se aproxima de uma curva linear-logarítmica.



5.2. **`LER` `POSFIXA EXP`, `INFIXA`, `RESOLVE`:** com base na análise de complexidade descrita na **seção 3**, temos que essa sequência tem a seguinte complexidade de tempo:

$$O(n) + O(n) + O(n) = O(max(n, n, n)) = O(n)$$

O gráfico à esquerda, construído a partir da ferramenta *gnuplot*, representa essa sequência de comandos, para entradas de diferentes tamanhos. Perceba que esse gráfico se aproxima de uma curva linear e que, o tempo máximo encontrado foi de 0.002s, valor menor que no caso anterior, de 0.009s, o que condiz com o fato de que $n < n \log n$, *para n grande*.

6. Conclusões

Neste trabalho, foram desenvolvidas diversas classes e funções para lidar com expressões algébricas escritas em notação infixa ou pós-fixa. Ao longo do desenvolvimento desse algoritmo, as principais estruturas de dados desenvolvidas foram pilhas encadeadas e árvores binárias, fato que levou ao aprendizado sobretudo de alocação dinâmica e recursão, imprescindíveis para o funcionamento delas. Além disso, elementos fundamentais da computação, como análise de complexidade de algoritmos e implementação de estratégias de robustez, foram desenvolvidas ao decorrer do trabalho. Em suma, foi um TP com uma boa curva de aprendizado.

7. Bibliografia

- Notação infixa. In: WIKIPÉDIA: A enciclopédia livre. [S.l.], 2018. Disponível em: <https://pt.wikipedia.org/wiki/Notação_infixa>. Acesso em: 11 abr. 2023.
- Notação polonesa inversa. In WIKIPÉDIA: A enciclopédia livre. [S.l.], 2022. Disponível em: <https://pt.wikipedia.org/wiki/Notação_polonesa_inversa>. Acesso em: 11 abr. 2023.
- cplusplus.com. Disponível em <https://cplusplus.com>. Acesso em: 21 abr. 2023

8. Instruções para compilação e execução

8.1. Abra o terminal.

8.2. Entre na pasta `TP`

- **Comando:** `cd / <caminho para a pasta > /TP`

8.3. Adicione o(s) arquivo(s) de entrada em `./TP`

8.4. Compile e execute o programa a partir do arquivo Makefile presente em `./TP`

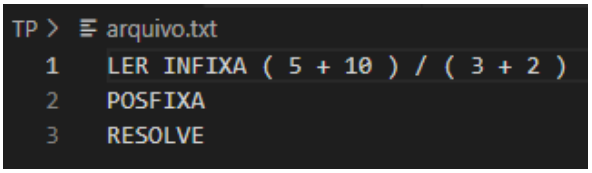
- **Comando:** `make run <file_name.txt>`
- *Observação 1:* este comando compila **e** executa o programa;
- *Observação 2:* o arquivo **deve** estar na pasta `TP`
- *Observação 3:* a cada nova execução, `<file_name.txt>` deve ser substituído pelo nome do arquivo que se deseja operar, com extensão `.txt`.

8.5. Como não foi especificado no enunciado do TP como deveria ser realizada a saída do TP, optei por deixar o resultado da execução impresso na **tela do terminal.**

8.6. Ao final de todas as execuções, utilizar o Makefile para limpar os arquivos gerados

- **Comando:** `make clean`

Exemplo para `./TP/arquivo.txt`




```
root@DESKTOP-O1SI1LT:~/home/josedmass/ED/TP1/TP# make run arquivo.txt
g++ -Wall -c -Iinclude -o obj/main.o src/main.cpp
g++ -Wall -c -Iinclude -o obj/ArvoreBinaria.o src/ArvoreBinaria.cpp
g++ -Wall -c -Iinclude -o obj/Expressao.o src/Expressao.cpp
g++ -Wall -c -Iinclude -o obj/operacoes.o src/operacoes.cpp
g++ -Wall -c -Iinclude -o obj/PilhaEncadeada.o src/PilhaEncadeada.cpp
g++ -pg -o bin/main obj/main.o obj/ArvoreBinaria.o obj/Expressao.o obj/operacoes.o obj/PilhaEncadeada.o  -lm
./bin/main arquivo.txt
EXPRESSAO OK: ( 5 + 10 ) / ( 3 + 2 )
POSFIXA: 5 10 + 3 2 + /
VAL: 3.000000
make: Nothing to be done for 'arquivo.txt'.
```

```
root@DESKTOP-O1SI1LT:~/home/josedmass/ED/TP1/TP# make clean
rm -f bin/main obj/main.o obj/ArvoreBinaria.o obj/Expressao.o obj/operacoes.o obj/PilhaEncadeada.o  gmon.out
```