# Trabalho Prático 1
# Resolvedor de Expressão Númerica


## Bernardo Dutra Lemos - 2022043949

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil
bernardolemos@dcc.ufmg.br


- ***Introdução:***

  A ideia principal do trabalho é ler uma linha contendo uma expressão matemática, seja ela no formato *Infixo* ou *Posfixo*, armazená-la em uma estrutura de dados e oferecer três opções, sendo elas imprimir a expressão nos dois formatos e uma opção para imprimir o resultado da expressão.

  A solução empregada para esse problema foi ler a expressão e com auxílio de estruturas de dados, como pilhas e filas, realizar a inserção da expressão em uma árvore binária, onde o caminhamento Pós-Ordem seria a forma Pósfixa, o caminhamento Em-Ordem seria a notação Infixa e o resultado da expressão seria uma forma adaptada do caminho Pré-Ordem.

  Exemplo de expressão em suas duas notações:

  - *Infixa*: 9 8 + 3 *
  - *Posfixa*: ( ( 9 + 8) * 3 )
  - *Resultado*: 51

- ***Método:***

  As estrutura de dados utilizadas para realizar esse trabalho foram pilhas, filas e árvore binária, as quais serão detalhadas a seguir:

  - **Pilha:**
    Estrutura escolhida para armazenar a expressão na forma Pósfixa, já que sua forma de funcionamento(LIFO - Last In Last Out), seria muito útil para a inserção na árvore.
  - **Fila Encadeada:**
    Estrutura escolhida para ajudar a realizar a conversão da forma infixa para a Pósfixa, já que o método usado para realizar essa conversão exigia que o primeiro elemento colocado fosse o primeiro sair, característica clássica das filas.
  - **Árvore binária:**
    Estrutura que mais se adequa para armazenar a expressão como um todo, pois seus caminhamentos realizam todas as operações requisitadas. Uma especificidade dessa árvore é que os nós internos sempre serão operadores matemáticos e as folhas sempre serão

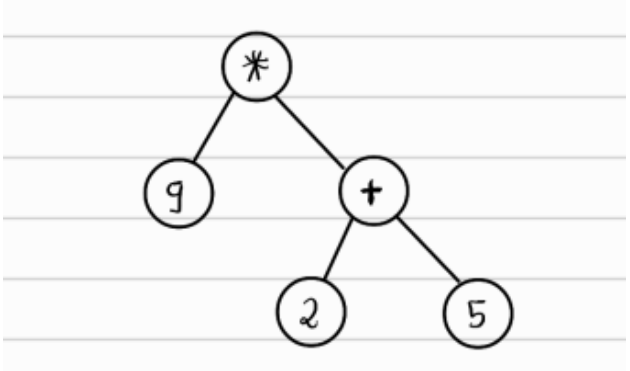
números. Ela apresenta todos os métodos convencionais de uma árvore binária, mas alguns adaptados e outros métodos extras para fazer validações de inserções, os quais serão mais comentados em sequência.

- **ArvoreExpressao:**
  Uma classe que tem apenas um atributo privado que é do tipo árvore, e possui diversos métodos, sendo a maioria deles baseado nos métodos do atributo privado, como o método “Vazio” (verifica se a expressão está vazia), “Resultado”, “*Infixa*” e “*Posfixa*” (chamam os métodos de caminhamento do atributo privado), também possui o método “InserirExp” (Recebe uma string e armazena na árvore). Além desses métodos, possui dois métodos privados que são: “IsInfix” (verifica se a string recebida pelo método de “InserirExp” é Infixa ou Posfixa) e “In2Pos” (realiza a conversão de Infixa para Posfixa).

- **Implementação:**
  A forma de implementação propriamente dita consiste em ler uma string com a função *getline*, presente na *standard library* do C++ e a partir dessa string, realizar séries de operações nela a fim de armazená-la na árvore.

  Primeiramente, é feito uma validação para definir em qual formato a expressão passada está, caso ela esteja no formato *Infixo* é feita a conversão para a forma *Pósfixa* através de um método privado da classe "ArvoreExpressao" e após isso a expressão, já na forma *Posfixa,* é colocada numa pilha de forma que cada elemento é retirado dela e inserido na árvore.

  Para fins de ilustração, a forma de inserção na árvore é realizada de forma que o formato dela para a expressão ( ( 5 + 2 ) * 9 ) seria:



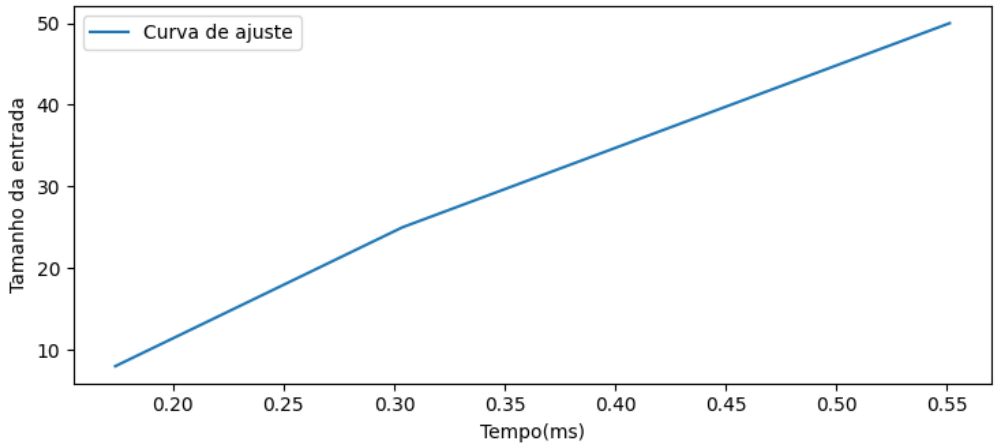

  Como dito anteriormente, a construção é feita de forma que os nós internos sempre são operadores binários e as folhas sempre são números.

Em resumo, o programa gira em torno da classe “ArvoreExpressao”, que recebe uma expressão em forma de string, realiza todas as verificações e manipulações necessárias para armazenar a expressão no atributo privado do tipo árvore e uma vez armazenado possui os métodos de impressão de resultados e conversão entre notações.

- ***Análise de Complexidade:***

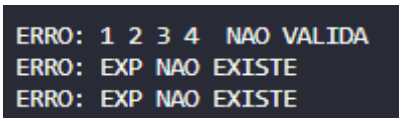
  A análise de complexidade será feita a partir de cada classe utilizada:

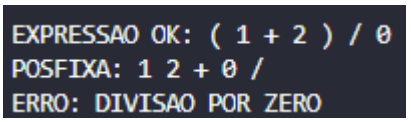
  - **Pilha:**
    Implementada de forma dinâmica, logo todos os métodos pertencentes a classe, com exceção do destrutor e do limpa que são O(n), são O(1).
  - **Fila Encadeada:**
    Assim como a Pilha, todos os métodos são O(1), com exceção do destrutor e do limpa.
  - **Árvore binária:**
    Método de inserção e o de verificação possuem em seu melhor caso uma complexidade de O(log n) e em seu pior caso O(n). Já todos os outros métodos possuem complexidade de O(n), pois visitam todos os nós e folhas da árvore.
  - **ArvoreExpressão:**
    Todos seus métodos estão diretamente ligados aos métodos da árvore binária, logo para os métodos de infixa, posfixa e resultado a complexidade é O(n) e para o de inserção possui os mesmos casos da árvore binária.
    A complexidade total do programa está ligada diretamente ao número total de caracteres válidos na expressão (números, operadores e espaços). Portanto, a complexidade do programa está na ordem de O(n), ou seja, linear.
    Aqui está um gráfico mostrando a forma como a curva de tempo do programa se comporta quando a entrada tem tamanho entre 5 e 50.



- ***Estratégias de robustez:***

As formas de programação defensiva implantadas foram através do uso de try catch, onde exceções eram lançadas e tratadas. As três exceções tratadas foram a de tentar armazenar uma expressão inválida, quando fosse tentar resolver a expressão armazenada houvesse uma divisão por zero e quando não houver nenhuma expressão armazenada. A forma como foi tratada será exemplificada nas imagens abaixo.




OBS: Quando uma expressão não é válida, ela não é armazenada na árvore, entretanto quando há uma divisão por zero, ela continua armazenada, apenas por uma decisão de projeto.

Devido a uma mudança na forma de entrada do trabalho, em que agora na entrada da expressão é também identificado em qual formato a expressão inserida está, foi feita uma adaptação de robustez na qual o programa identifica que se a expressão é realmente daquele formato especificado, caso não seja, é lançado um erro de expressão inválida. Por exemplo, se a entrada for “LER POSFIXA ( 3 + 4 )”, onde diz que a expressão é posfixa e na verdade ela é infixa, é lançado um erro. Mas é importante ressaltar que o programa não precisaria dessa pré-identificação para funcionar, pois ele já verifica automaticamente o tipo.

- ***Análises Experimentais:***

Para análises experimentais foi feita a medida de tempo de execução para diversas expressões de variados tamanhos, além de comparar seu resultado com o resultado esperado. As medidas de tempo foram feitas usando a função clock().

Para expressões de até 8 números a média de tempo foi de *0,173875* milissegundos. Quando a quantidade de números aumenta para 25 a média se torna aproximadamente *0,303714* milissegundos. E por último, quando são cerca de 50 números a média é *0,551286* milissegundos.

Uma coisa interessante de se observar é que quanto maior a expressão maior é o erro numérico que ela carrega, já que operações com floats tendem a gerar grandes inconsistências quando realizadas de forma

sequencial. Por exemplo, para algumas operações a partir de 20 números, começaram a aparecer divergências entre os resultados dados e os resultados esperados a partir da quarta casa decimal.

O erro mais expressivo para os casos testados, foi onde o resultado era um float próximo da oitava potência de dez e ocorreu um erro de -715,585214, onde a resposta esperada era -660219339.585214 e a obtida foi -660218624.000000, mas é o único caso onde a diferença foi muito expressiva e acredito que seja devido a aritmética de ponto flutuante ser inconsistente.

- ***Conclusões:***

O projeto inteiro consistiu em receber uma string, definir qual era o tipo de notação da entrada, converter para a notação *Pósfixa* e realizar a inserção dela na árvore onde todos os 3 métodos definidos estariam disponíveis.

Esse trabalho se mostrou bastante desafiador, exigindo tempo para pensar em como as estruturas de dados seriam utilizadas e quais seriam as melhores para cada situação e além disso em pensar como seria realizada as lógicas e conversões necessárias para o programa funcionar de acordo com o planejado.

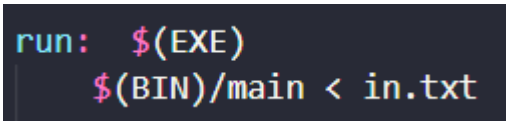
Durante a realização deste trabalho foi aprendido que é possível manipular as estruturas de dados de forma que elas se comportem da forma que é necessária para tal situação, por exemplo na forma de inserção, onde eu precisava garantir que as folhas sempre fossem números e os nós internos operadores, e na forma de percorrer a árvore para obter o resultado total da expressão. Também mostrou como estruturas de dados mais simples como pilhas e filas podem ser ferramentas poderosas de auxílio na manipulação dos dados, no caso ambas me ajudaram a pôr a expressão na árvore binária.

- ***Bibliografia:***

  - Disponível em:<https://www.andrew.cmu.edu/course/15-200/s06/applications/ln/junk.html#:~:text=To%20convert%20an%20infix%20expression,same%20expression%20in%20prefix%20notation.>. Acesso em 20 de abril de 2023
  - Disponível em:<https://pt.wikipedia.org/wiki/Nota%C3%A7%C3%A3o_polonesa_inversa#:~:text=Nota%C3%A7%C3%A3o%20Polonesa%20Inversa%20.> Acesso em 20 de abril de 2023
  - Disponível em:<https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html.> Acesso em 20 de abril de 2023

- ***Instruções para compilação e execução:***

1. Acessar a pasta extraída do zip utilizando “**cd TP**”.
2. Na seguinte linha (24) do Makefile, substitua o “in.txt” pelo nome do arquivo a ser lido como entrada




3. Digite no terminal “make run” para compilar e executar o programa.
4. Para limpar os objetos e o executável, utilize o comando “make clean”.