Trabalho Prático 1
Resolvedor de Expressão Numérica

Lorenzo Ventura Vagliano

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lorenzovagliano@dcc.ufmg.br

1 - Introdução

O problema proposto foi implementar um programa para a resolução de expressões tanto na notação "Infixa" quanto na notação "Pós Fixa". Além disso, o programa é capaz de converter expressões entre essas duas notações.

A notação Infixa é a notação que as pessoas geralmente usam para resolver expressões matemáticas à mão, já que ela é extremamente mais legível do que a Pós Fixa, que é usada por computadores.

Notação Infixa
Ex: (((a) + (b)) * ((c) + (d)))

Notação Pós Fixa
Ex: a b + c d + *

2 - Método

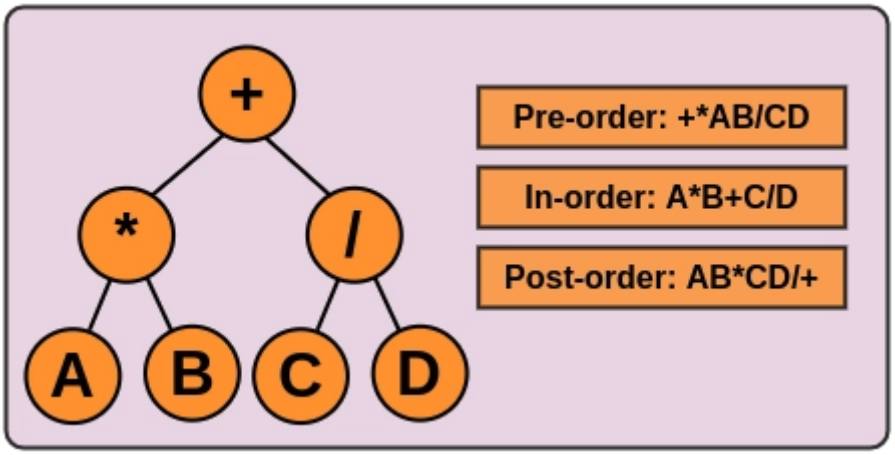
O programa foi desenvolvido em linguagem C++ e compilado pelo compilador g++ versão 11.3.0.

2.1 - Estrutura de Dados/Classes

A implementação do programa teve como base a estrutura de dados de uma Árvore Binária, mais especificamente, uma Árvore Binária de Expressão, na qual as suas folhas representam os números ou operandos e os seus nós internos, os operadores. Nesse caso, os nós da árvore armazenam strings.

Além disso, foram utilizados 2 tipos de Pilha, uma que comporta o tipo de nó da árvore e outra que comporta strings.

A chave para o problema foi o mapeamento de expressões utilizando caminhamentos na Árvore Binária, como pode ser observado na imagem:



2.2 - Funções

O programa tem 6 funções principais e algumas auxiliares.

TreeNode* buildPostfixExpressionTree(string postfix)

Essa é a função responsável por criar uma árvore de expressões a partir de uma expressão Pós Fixa.

Ela faz isso ao percorrer a string fornecida da esquerda para a direita, e para cada operando ou operador encontrado, cria um novo nó na árvore e o insere no local apropriado, conforme a ordem de precedência dos operadores.

Para construir a árvore, a função cria um nó para cada operando ou operador encontrado na string. Se o nó é um operando, ou seja, um número, o valor do nó é definido como o número correspondente. Se o nó é um operador, o valor do nó é definido como o caractere correspondente ao operador.

A função utiliza uma pilha para armazenar os nós que já foram criados e ainda não foram colocados na árvore. Quando um operando é encontrado na string, um novo nó é criado para ele e colocado na pilha. Quando um operador é encontrado, dois nós são desempilhados da pilha, um para a subárvore esquerda e outro para a subárvore direita. O operador é definido como o valor do nó raiz, e os dois nós desempilhados são definidos como os filhos esquerdo e direito desse nó raiz. O novo nó é então empilhado na pilha.

Ao final do percurso da string, a pilha contém apenas um nó, que é a raiz da árvore de expressão. Esse nó é desempilhado da pilha e retornado como resultado da função.

TreeNode* buildInfixExpressionTree(string infix)

Essa é a função responsável por criar uma árvore de expressões a partir de uma expressão Infixa. Que, por não ser uma derivação tão direta de uma Árvore de Expressão, é consideravelmente mais complexa.

A função itera sobre a expressão infixa e extrai cada um dos elementos presentes nela, que podem ser operadores, parênteses ou operandos. A cada iteração, a função verifica se o caractere atual é um espaço em branco. Caso seja, a função simplesmente prossegue para a próxima iteração. Caso contrário, a função verifica se o caractere atual é um dígito. Se for, a função continua iterando até encontrar o fim do número, e então adiciona o número à pilha. Se o caractere atual não for um dígito, a função trata ele como um operador ou parêntese. Se o caractere atual for um operador, a função empilha ele na pilha. No entanto, antes de empilhar o operador, a função verifica se a precedência do operador é menor ou igual à precedência do operador no topo da pilha. Se for, a função desempilha o operador do topo da pilha e o adiciona à árvore de expressão. Em seguida, a função repete o processo até que o operador no topo da pilha tenha precedência menor que o operador atual, ou até que a pilha esteja vazia.

Se o caractere atual for um parêntese de abertura, a função simplesmente o empilha na pilha.

Se o caractere atual for um parêntese de fechamento, a função desempilha todos os elementos da pilha até encontrar o parêntese de abertura correspondente. Em seguida, a função adiciona os elementos desempilhados à árvore de expressão.

Ao final da iteração, a pilha contém os operadores e parênteses restantes na expressão infixa. A função então desempilha esses elementos e adiciona eles à árvore de expressão.

void postfixToInfix(TreeNode* root)

Essa função faz a conversão de uma expressão posfixa para uma infixa. Ela faz isso ao navegar a árvore gerada pela expressão infixa utilizando o caminhamento In ordem.

O primeiro passo da função é verificar se o nó passado como parâmetro é nulo. Se for, a função retorna sem fazer nada. Caso contrário, a função verifica se o nó é uma folha (ou seja, não possui filhos). Se for, a função retorna a string que representa o valor da folha. Caso contrário, a função chama recursivamente a si mesma para converter a subárvore esquerda e direita e, em seguida, concatena as strings resultantes com o operador do nó atual. Ela também envolve cada passo da expressão com parênteses.

Esse processo continua recursivamente até que todos os nós da árvore tenham sido visitados e a expressão infixa completa tenha sido construída.

void infixToPostfix(TreeNode* root)

Essa função faz a conversão de uma expressão Infixa para uma Pós fixa. Ao invés de utilizar o caminhamento In ordem, ela utiliza o caminhamento Pós ordem.

Da mesma maneira, ela verifica a validade do nó passado como parâmetro e visita os nós da árvore pelo caminhamento, imprimindo seus conteúdos nessa ordem.

float evaluatePostfixExpressionTree(TreeNode* root) e

float evaluateInfixExpressionTree(TreeNode* root)

Ambas funções possuem o mesmo mecanismo para avaliar uma expressão, a única diferença está na árvore que cada uma recebe, Ou seja, ao implementar funções diferentes para transformar expressões Pós Fixas e In Fixas em Árvores de Expressão, poupamos um algoritmo complexo para resolvê-las, já que é necessário apenas resolver a árvore.

Para fazer isso, ela caminha recursivamente até as folhas da árvore, utilizando o caminhamento Pós Order, transformando os operandos de string para float e usando a função auxiliar evaluate(), que traduz o operador em string para os valores que acabaram de ser transformados em floats.

Ademais, ambas as funções verificam se em caso de divisão o denominador será 0.

3 - Análise de Complexidade

TreeNode* buildPostfixExpressionTree(string postfix)

Essa função percorre a string postfix uma vez, criando um nó para cada operando ou operador encontrado e empilhando-os em uma pilha. Quando um operador é encontrado, os dois últimos nós da pilha são removidos e são adicionados como filhos do nó criado para o operador. Assim, essa função realiza uma única passagem na string e em cada nó da árvore, tendo uma *complexidade de tempo* de **O(n)**, em que n é o tamanho da string postfix. Já a *complexidade de espaço* é **O(n)**, pois a pilha criada pode chegar a ter n/2 elementos, caso a string postfix seja composta somente de operadores.

TreeNode* buildInfixExpressionTree(string infix)

Assim como a função anterior, essa função percorre a string uma vez, criando um nó para cada operando ou operador encontrado e empilhando-os em uma pilha. No entanto, a ordem em que os nós são adicionados à árvore é diferente, uma vez que a árvore precisa ser construída seguindo as regras de precedência dos operadores. No pior caso, cada nó pode ser empilhado e desempilhado da pilha duas vezes (quando um operador de precedência maior é encontrado), resultando em uma *complexidade de tempo* de **O(n)**, em que n é o tamanho da string. Já a *complexidade de espaço* é **O(n)**, pela mesma razão que a função anterior.

void postfixToInfix(TreeNode* root)

Essa função percorre a árvore de expressão em uma ordem pré-fixada, ou seja, primeiro visita a raiz, depois a subárvore da esquerda e, por fim, a subárvore da direita. Durante a visita,

cada nó é empilhado em uma pilha de nós. Quando um operador é encontrado, os dois últimos nós são desempilhados e é criado um novo nó com o operador em questão como raiz e os nós desempilhados como seus filhos. Esse novo nó é empilhado novamente na pilha de nós. No pior caso, cada nó é empilhado e desempilhado da pilha duas vezes, resultando em uma *complexidade de tempo* de **O(n)**, em que n é o número de nós na árvore. Já a *complexidade de espaço* é **O(n)**, pois a pilha criada pode chegar a ter n elementos, caso a árvore seja linear.

void infixToPostfix(TreeNode* root)

Também possui *complexidade de tempo* O(n), onde n é o número de nós na árvore. Isso ocorre porque a função visita cada nó exatamente uma vez, realizando operações constantes em cada um deles.

Já a *complexidade de espaço* é **O(n)**, pois a função utiliza uma fila para armazenar os nós visitados durante a travessia da árvore. Como todos os nós são visitados uma vez, a fila terá tamanho O(n).

float evaluatePostfixExpressionTree(TreeNode* root) e

float evaluateInfixExpressionTree(TreeNode* root)

As funções têm *complexidade de tempo* **O(n)**, onde n é o número de nós na árvore. Isso ocorre porque a função visita cada nó exatamente uma vez, realizando operações constantes em cada um deles.

Já a *complexidade de espaço* é **O(h)**, onde h é a altura da árvore. Isso ocorre porque a função utiliza uma pilha para armazenar os operandos enquanto percorre a árvore. Como a pilha terá tamanho igual à altura da árvore, a complexidade de espaço é O(h).

Complexidade geral:

Tempo: O(n).

Espaço: O(n).

4 - Estratégias de Robustez

A fim de prover mais robustez ao sistema. Além de seguir boas práticas de desenvolvimento como a modularização e o encapsulamento de um ponto de vista de Orientação a Objetos, expressões inválidas como a divisão por 0 foram tratadas.

5 - Análise Experimental

O gráfico acima plota o tempo de execução das 6 funções do programa chamadas consecutivamente, evidenciando seu comportamento linear e ordem O(n).

Dessa maneira, analisando o gráfico podemos determinar se houve alguma anomalia ou erro. Mas, como o gráfico segue o comportamento esperado, podemos confirmar o que foi postulado em teoria.

6 - Conclusões

Com esse projeto, foi possível implementar um algoritmo de forma elegante por intermédio do uso da Árvore de Expressão. O problema poderia ter sido solucionado com força bruta, utilizando apenas pilhas e vetores, como alguns colegas observaram. Porém, o uso inteligente das estruturas de dados permitiu uma solução mais robusta, eficiente e intuitiva.

Projetando esse sistema, foram desenvolvidos métodos de caminhamento de árvores aplicados para a construção, conversão e solução de expressões. Ademais, o uso de pilhas se provou extremamente útil devido à sua natureza FIFO, em comparação com vetores, por exemplo.

7 - Bibliografia

Data Structures and Algorithms(Jenny's Lectures, 2020).
Case Study: Expression Tree Evaluator Vanderbilt University(Douglas C. Schmidt).
Slides Estruturas de Dados(Chaimowicz & Prates, 2019).
Árvores Binárias IME-USP(Paulo Feofiloff, 2017).

8 - Instruções para Compilação e Execução

Para compilar o programa, basta extrair o zip, abrir o terminal na pasta onde foi extraído e utilizar o comando “make run”.

Observações:
As entradas devem seguir o formato disponibilizado nos exemplos de entradas. Todos os números, operadores e parênteses separados por espaços. Na infixa, todo número cercado por parenteses.
Ex: Infixa: ((1) + (2)) Pós Fixa: 1 2 +

Se estiver utilizando o terminal integrado de uma IDE, insira os inputs linha por linha. No terminal do Linux, entretanto, é possível colar várias linhas ao mesmo tempo.