Trabalho Prático 1

Resolvedor de Expressão Numérica

João Victor Evangelista Cruz

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

joaovecruz@ufmg.br

1. Introdução

O problema proposto foi implementar um algoritmo capaz de resolver expressões numéricas. O algoritmo deve conseguir ler e armazenar expressões numéricas, estas podendo estar na forma de representação usual (também chamada de infixa) ou na notação polonesa inversa (também chamada de posfixa), assim como a possibilidade de obter o resultado da expressão automaticamente. Para a solução do problema, foi necessário utilizar estruturas de dados para armazenar a expressão matemática de forma que fosse possível fazer a transformação dela assim como se obter o resultado de uma forma mais simples.

2. Método

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

2.1 Estrutura de Dados

A implementação do programa teve como base as estruturas de dados Árvore e Pilha.

A árvore tem o papel de armazenar os termos da expressão matemática, sendo eles números ou símbolos das operações básicas: multiplicação (*); divisão (/); adição (+) e subtração (-). A árvore é organizada de modo que cada nó pai possui armazenado nele um símbolo, enquanto que cada nó folha possui um número. Ou seja, um nó que possui um número armazenado não pode ser um nó pai, enquanto que um nó que possui um símbolo armazenado deve ter pelo menos dois nós filhos, sejam eles números ou símbolos.

Já no caso da pilha, são utilizadas duas pilhas diferentes. A primeira armazena nós da árvore e serve para auxiliar na inserção da expressão na árvore. Já a segunda armazena números do tipo double e auxiliam na obtenção do resultado da expressão.

2.2 Classes

Para a modularização da implementação, foram criadas seis classes. Há a classe Arvore, responsável pela estrutura já descrita acima. Há também as classes PilhaNo e PilhaNum, a primeira é responsável pela pilha de nós e a segunda pela pilha de números, também descrita acima. Além disso, há a classe No, que é a classe auxiliar da classe Arvore e serve para armazenar os elementos da árvore e ligar esses elementos entre si. Por fim, há as classes CelulaNo e CelulaNum que são classes auxiliares das pilhas e também funcionam armazenando os elementos das pilhas e ligando esses elementos entre si.

2.3 Funções

Arvore:

void Arvore::InsereInfixa(std::string exp): Função que recebe como parâmetro um termo exp da expressão matemática na forma Infixa como um string e realiza a inserção dele na Arvore através da chamada da função auxiliar InsereRecursivo.

void Arvore::InsereRecursivo(No* &p, std::string item): Função recursiva que recebe como parâmetro um ponteiro de um No p da árvore e um termo item da expressão matemática como um string. A função tem o seguinte funcionamento: Como todas as expressões válidas têm parênteses abertos antes de cada número ou símbolo, é inserido primeiro um parêntese na árvore e o nó desse parêntese é empilhado na pilha auxiliar de nós da árvore. Quando item é um símbolo ou um número, um nó é desempilhado da pilha e define-se o conteúdo desse nó como o item.

void Arvore::InserePosfixa(std::string exp): Função que recebe como parâmetro a expressão matemática na forma Posfixa como um string, essa expressão tem seus termos separados por espaço e na função, esses termos são separados e utilizados para chamar a função auxiliar InserePosfixaTermo.

void Arvore::InserePosfixaTermo(std::string termo): Função que utiliza uma pilha auxiliar de nós. Nessa função, são criadas subarvores que vão sendo empilhadas e juntadas, formando a árvore de baixo para cima.

void Arvore::PosOrdem(No *p): Função recursiva que recebe como parâmetro um nó e realiza a o caminhamento na ordem Pós-Fixada e imprime cada elemento da árvore, dessa forma auxiliando na conversão da expressão da forma Infixa para a forma Posfixa.

void Arvore::InOrdem(No *p, int a): Função recursiva que recebe como parâmetro um nó e realiza a o caminhamento na ordem Central e imprime cada elemento da árvore, além de também imprimir parênteses dependendo do item que o nó possui, auxiliando assim na conversão da expressão da forma Posfixa para a forma Infixa.

void Arvore::Caminha(int tipo): Função que realiza a impressão da árvore através da chamada das funções PosOrdem e InOrdem. Quando tipo é igual a 2, chama-se InOrdem, já quando tipo é igual a 3, chama-se Pos-Ordem. Quando tipo tem qualquer outro valor, nada acontece.

void Arvore::ExpPos(No *p): Função que faz o caminhamento PosOrdem na árvore e assim, salva a expressão Posfixa na árvore, o que auxilia na obtenção do resultado da expressão.

void Arvore::Resultado(): Função utiliza uma pilha auxiliar de números e que faz a chamada da função ExpPos, separa a expressão em termos e chama a função AuxResult em cada termo, obtendo assim o resultado da expressão. Após isso, imprime na tela o resultado.

void Arvore::ApagaRecursivo(No *p): Função que percorre a árvore a partir do nó p, deletando todos os nós abaixo de p.

void Arvore::Limpa(): Função que chama a função ApagaRecursivo com o parâmetro raiz, assim, apagando todos os nós da árvore.

No * Arvore::GetRaiz(): Função que retorna a raiz da árvore.

void Arvore::LimpaPilha(): Função que deleta todos os elementos da Pilha de nós auxiliar da árvore.

Expressão.cpp:

bool VerificaNum(std::string s): Função auxiliar na leitura da expressão que recebe o string s e verifica se este é um número e se é válido, se está correto para ser inserido posteriormente na árvore. Retorna true se for um número válido, e retorna false caso a string não representar um número ou se o número não estiver na forma correta.

int VerificaSymbol(std::string s): Função auxiliar na leitura da expressão que recebe o string s e verifica se s representa algum dos sinais presentes nas expressões matemáticas. Se for algum operador das quatro operações básicas da matemática, a função retorna 1. Se for um sinal de abre parêntese (“(“), retorna 2. Se for um sinal de fecha parêntese (“)“), retorna 3. Caso não for nenhum desses símbolos, retorna 0.

bool VerificaExpInfixa(std::string s): Função que utiliza as funções VerificaNum e VerificaSymbol e recebe como parâmetro o string s que representa a expressão que se quer verificar se é válida ou não na forma Infixa. Para ser válida, a expressão deve ter apenas números, os quatro operadores válidos, além de parênteses. A expressão também deve ter números e sinais das operações intercalados. Além disso, o primeiro e o ultimo termo da expressão, sem contar os parênteses, devem ser números. Ademais, o número de abre parêntese deve ser o mesmo de fecha parêntese, e esse número deve ser também o mesmo número de termos da expressão.

bool VerificaExpPosfixa(std::string s): Função que utiliza as funções VerificaNum e VerificaSymbol e recebe como parâmetro o string s que representa a expressão que se quer verificar se é válida ou não na forma Posfixa. Para ser válida, a expressão deve ter apenas números e os quatro operadores válidos. O primeiro termo da expressão também deve ser um número e o último termo deve ser um operador. Além disso a quantidade de números deve ser sempre um a mais que a quantidade dos operadores. Ademais, a quantidade de símbolos já contados tem que ser menor do que a quantidade de números já contados até o momento.

bool AuxResult(std::string s, PilhaNum* &P): Função que recebe como parâmetro o endereço de uma pilha de números e um string s que representa um termo da expressão. Essa função auxilia a função

Resultado, da árvore. Se esse termo for um número, esse número é empilhado. Se for algum dos sinais das quatro operações básicas, são desempilhados dois números da pilha, é realizada a operação, que o sinal representa, com esses dois números e o resultado é novamente empilhado. Retorna true se ocorrer tudo certo nas operações realizadas e retorna false se houver alguma divisão por 0.

No:

void No::SetItem(std::string i): Função que define i como o item do respectivo nó.

void No::SetEsq(No *p): Função que define p como o nó esq do respectivo nó.

void No::SetDir(No *p): Função que define p como o nó dir do respectivo nó.

std::string No::GetItem(): Função que retorna o item do respectivo nó.

void No::Imprime(): Função que imprime o item do respectivo nó.

PilhaNo:

bool PilhaNo::Vazia(): Função que retorna true se a pilha estiver vazia e false se não estiver.

void PilhaNo::Empilha(No *no): Função que recebe como parâmetro um nó e empilha esse nó, colocando-o no topo da pilha.

No* PilhaNo::Desempilha(): Função que deleta o nó do topo da pilha e retorna esse nó.

void PilhaNo::Limpa(): Função que utiliza a função Desempilha e deleta todos os elementos da pilha.

PilhaNum:

bool PilhaNum::Vazia(): Função que retorna true se a pilha estiver vazia e false se não estiver.

void PilhaNum::Empilha(double numero): Função que recebe como parâmetro um número e empilha esse número, colocando-o no topo da pilha.

double PilhaNum::Desempilha(): Função que deleta o número do topo da pilha e retorna esse número.

void PilhaNum::Limpa(): Função que utiliza a função Desempilha e deleta todos os elementos da pilha.

3. Análise de Complexidade

3.1 Tempo

Os laços utilizados para separar os termos da expressão tem complexidade assintótica da ordem de f(n) = O(xn) + O(yn) = O(n), em que x é a quantidade média de caractere de cada termo da expressão, y é a quantidade média de espaços entre cada termo e n é o número de termos da expressão.

A função de inserção Infixa é g(n) = O(log n), já que ela vai fazendo chamadas recursivas e assim, reduzindo o problema em problemas menores.

A função de inserção Posfixa é h(n) = O(n), já que a árvore é criada de baixo para cima através da criação de várias subarvores.

A conversão de infixa para posfixa é de ordem i(n) = O(n), já que a função passa pelos n nós da árvore imprimindo.

Da mesma forma, a conversão de posfixa para infixa é de ordem j(n) = O(n) pelo mesmo motivo.

A função de imprimir o resultado também utiliza os laços de separação dos termos, por isso, também é k(n) = O(n)

Portanto, o programa tem complexidade assintótica de

f(n) + g(n) + h(n) + i(n) + j(n) + k(n) = O(n)

3.2 Espaço

Para a execução do programa, é necessário armazenar os n elementos da expressão na árvore, ou seja O(n), além de utilizar duas pilhas, cujo custo é O(p), em que p é sempre menor que n, já que nunca os n elementos são empilhados ao mesmo tempo.

Por isso, o custo assintótico de espaço para o algoritmo é

$O(p) + O(p) + O(n) = O(n)$

4. Estratégias de Robustez

No arquivo principal, há a abertura do arquivo que será lido para a execução. Se o arquivo não conseguir ser aberto, é impressa na tela um erro comunicando a falha.

Além disso, se a operação escolhida não for “LER”, “INFIXA”, “POSFIXA” ou “RESOLVE”, é impressa na tela uma mensagem de erro, já que não há operações diferentes dessas.

Se for selecionada a operação “LER”, porém após isso não tiver as palavras “INFIXA” ou “POSFIXA”, também é impressa na tela uma mensagem de erro.

Se na operação de leitura, a expressão for inválida, é impressa na tela uma mensagem de erro avisando da falha.

Caso forem selecionadas as operações “INFIXA”, “POSFIXA” ou “RESOLVE”, porém não houver nenhuma expressão armazenada, também é impressa na tela uma mensagem de erro.

Na função de inserção posfixa, como só devem ser inseridos sinais de operação ou números, se algo diferente disso tentar ser inserido, é gerado uma mensagem de erro na tela. Além disso, como é utilizada uma pilha nesse processo, se essa pilha estiver vazia quando houver a tentativa de desempilhar um elemento, também é impresso um erro na tela.

Por fim, na função que produz a resolução da expressão, caso haja alguma divisão por 0, não tem jeito de se obter o resultado e, portanto, é impresso uma mensagem de erro na tela.

5. Análise Experimental

N° de termos	Tempo gasto em segundos
10	0.003555
50	0.004274
100	0.004552
200	0.004980
500	0.006080

A partir dos experimentos para observar a variação do tempo gasto pelo algoritmo para realizar as operações de leitura, conversão e resolução da expressão em determinadas quantidades de termos em cada uma, podemos perceber que o tempo cresce linearmente conforme a quantidade de termos sobe, o que vai de encontro com o que já foi mostrado anteriormente na análise de complexidade do algoritmo, que foi dito que é de O(n).

6. Conclusão

Para o trabalho, foi necessário tanto a escolha e adaptação da estrutura de dados correta, para que a expressão pudesse ser armazenada de duas formas diferentes, além de ser possível converter de uma forma para outra. Além disso, houve a necessidade de utilizar de outra estrutura de duas formas diferentes para que fosse possível todo funcionamento do algoritmo.

Ademais, houve a necessidade de aprender mais sobre as formas infixa e posfixa das expressões matemáticas, as regras que cada notação deve respeitar para que desse para passar essas especificações para o algoritmo e o mesmo pudesse funcionar de maneira correta e realista.

Com esse trabalho, fica nítido a importância da organização do código através da modularização, principalmente quando se tem códigos grandes e com várias classes, como esse e também a utilização do makefile, para poupar tempo e esforço na hora de compilar o programa e fazer testes. Além disso, percebe-se a importância de saber utilizar as estruturas de dados, principalmente a Arvore, nesse trabalho, pois sem isso, seria quase impossível desenvolver uma solução para o problema apresentado. O conhecimento das diferentes estruturas de dados é vital para programas de qualidade.

7. Bibliografia

- Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011.

- Cormen, T., Leiserson, C, Rivest R., Stein, C. Algoritmos – Teoria e Prática, 3a. Edição, Elsevier, 2012.

8. Instruções para compilação e execução

Para a compilação do programa, deve-se utilizar o comando “make all”, no terminal, estando na raiz da pasta TP.

Para a execução, deve se colocar o arquivo de teste na pasta TP, na raiz do projeto e digitar o comando estando na raiz do projeto, no terminal:

“./bin/main nome_arquivo.txt”, onde nome_arquivo.txt representa o nome do arquivo de testes.

Por exemplo, se o nome do arquivo for “testes.txt”, o comando deverá ser

```
./bin/main testes.txt
```