Trabalho Pratico 0

Solucionador e conversor de expressões numéricas

Davi Sakamoto Lamounier: 2022035873

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte – MG – Brasil

1. Introdução

O problema proposto foi implementar um programa que recebe como argumento um arquivo de texto com uma ou mais expressões numéricas, que podem estar na notação infixa ou posfixa. O programa então deve ser capaz de validar a expressão recebida e armazená-la numa estrutura de dados apropriada, capaz de realizar conversões de uma notação para outra e solucionar a equação.

2. Implementação

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

A implementação do programa teve como base o uso de loops que percorrem as linhas do arquivo e identificam os comandos recebidos e a expressão numérica. Isso é feito a partir da biblioteca “fstream” O arquivo por sua vez tem seu nome passado como argumento durante a compilação. O programa então avalia os comandos, um por um, e realiza as operações necessárias atribuídas a cada um, caso alguma expressão tenha sido fornecida. Caso contrário, o programa apenas responderá aos comandos com erros de expressão inexistente. Uma vez recebida uma expressão, há uma análise da mesma para conferir se ela é válida, para então tentar armazená-la. Caso ela não seja aceita, um erro de expressão inválida será exibido e o programa continuará agindo como se nenhuma expressão tivesse sido fornecida até que uma nova expressão válida apareça.

2.1. Estrutura de Dados e Classes

Para modularizar a implementação do programa, foram criadas quatro classes. O armazenamento da expressão válida é feito utilizando uma árvore binária, que é a nossa primeira e mais importante classe. Ela representa a expressão numérica em sua forma sintática, uma vez que pode ser interpretada como um diagrama relacionando operadores e operandos, diferente das notações posfixa e infixa, que são uma representação escrita linearmente da expressão. Essa estrutura foi escolhida pois, além dessa representação mais fiel da expressão, ela pode facilmente se transformar em alguma das duas notações através dos caminhamentos.

Em nossa árvore os nós intermediários representam os operadores, e as folhas representam os números. Ela é montada de forma que toda operação realizada na expressão seja representada por um nó contendo o operador, e seus dois filhos são os operandos, podendo eles serem um número ou outro operador com mais dois filhos, indicando que a operação deve ser feita com o resultado desta outra operação, que possui prioridade maior. Dessa forma também é perceptível que todas as folhas serão números, uma vez que se tivessem filhos precisariam ser um operador para relacioná-los.

Além disso ela possui como um membro privado um ponteiro que aponta para um Nó, que por sua vez é uma outra classe implementada para complementar a árvore. O nó é bem simples e possui um item armazenado, e dois ponteiros para outros nós, sendo eles seus “filhos” direito e esquerdo.

Este item armazenado é nossa terceira classe, que é armazena uma string e um tipo, e é utilizada para representar os termos da expressão, podendo ser um número, um operador, um parêntese ou uma palavra (como o comando ou alguma outra palavra indesejada na expressão). O item determina seu tipo automaticamente em sua construção, e tem alguns operadores sobrecarregados para facilitar atribuições e comparações.

Além disso alguns métodos da árvore utilizam de uma pilha de ponteiros para nós, que é nossa quarta e última classe, ajudando na implementação da construção da árvore de maneira adequada e mais eficiente. A pilha é outra estrutura de dados muito útil em certas ocasiões como essa, e é implementada no programa de maneira sequencial.

2.2. Métodos

Na main do programa existe uma função “avaliaExpressao()”, que recebe como parâmetro a string da linha de expressão lida e avalia se ela é válida ou não. A função utliza de uma biblioteca de fluxo de string (“sstream”) para ler as palavras da string separados por espaço. Primeiramente ela remove o comando “LER” e depois verifica a qual notação ele deve considerar para avaliar. Então ela utiliza de um loop para analisar termo por termo, analisando seu tipo e comparando-o sempre com o termo anterior para conferir se a sequência é permitida. Ao longo da execução vários testes são feitos para conferir a validade da expressão, com o auxílio de alguns contadores de operadores, operandos, parênteses, etc. No fim, ela retorna se a expressão é posfixa, infixa ou inválida.

A árvore também conta com diversos métodos, como construtor/destrutor, caminhamentos (inOrder, preOrder, etc), métodos para construir a árvore a partir de uma entrada em notação posfixa ou infixa, outros para imprimir ela em alguma das notações e por fim para calcular o resultado da expressão.

Em primeiro lugar vamos falar dos métodos constroiInfixa() e constroiPosfixa() que, de maneira semelhante ao método avaliaExpressao(), também recebem uma string

como entrada e utilizam de fluxo de strings para analisar os termos da expressão. Um loop cria itens com os termos lidos e os coloca em nós, então dependendo do tipo do item e da notação lida esses nós podem ser empilhados ou receberem como filhos/se tornarem filhos dos últimos nós da pilha. No fim, o último nó restante é a raiz da árvore.

Os métodos de caminhamento são funções recursivas que passeiam pelos nós da árvore em ordens específicas, e são auxiliares para os métodos conversores/impressores. Isso porque, para imprimir a expressão em sua notação infixa, basta imprimir os termos da expressão através de um caminhamento in-order, apenas adicionando os parênteses necessários. A mesma ideia serve para a notação posfixa, porém utilizando dessa vez o caminhamento pos-order e sem adição de parênteses.

O método utilizado para resolver a expressão também é simples, e é uma função recursiva que realiza as operações através de um caminhamento pos order, que retorna, caso o nó contenha um número, o próprio número, e caso contenha um operador, retorne a respectiva operação entre os números retornados pela chamada da função sobre seus dois filhos. Isso se repete que que reste apenas um número no final, que é o resultado da expressão.

A pilha possui alguns métodos para retornar o item em seu topo, para empilhar um novo item e para desempilhar o último item, além de construtor e um método para limpá-la. O item possui dois construtores, sendo um deles o default e outro para determinar seu tipo. Possui funções de impressão, get's e os operadores '=', "==" e "!=" sobrecarregados para ajudar em comparações e atribuições.

3. Análise de Complexidade

avaliaExpressao(): A função possui um loop que itera sob os termos da expressão (O(n)), e em cada etapa realiza até 4 comparações, realizando algumas outras chamadas de custo 1 em cada, porém ao somar chamadas de ordem O(1), continuamos em O(1). Ao multiplicá-los por O(n), temos que a complexidade da função é de ordem O(n).

Caminhamentos: Sabendo que os caminhamentos são funções recursivas com duas chamadas para cada filho e uma impressão em cada nó não nulo, podemos utilizar o teorema mestre, considerando que cada chamada para um filho divide o programa pela metade, ou seja:

$$T(n) = a*T(n/b) + f(n)$$

$$T(n) = \mathbf{2}*T(n/\mathbf{2}) + \mathbf{1}$$

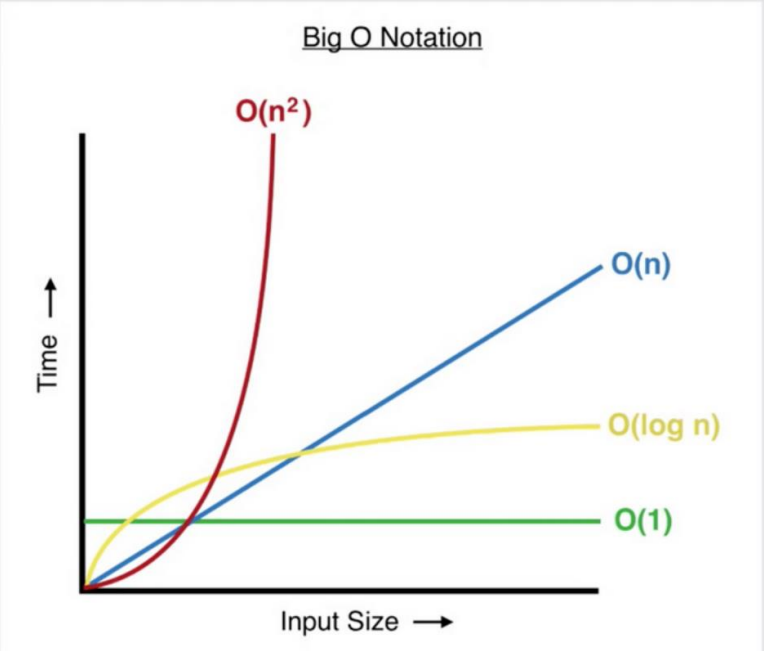
$$n^{(Log_2 2)} = n > 1$$

Ou seja, a complexidade da função é Θ(n).

Conversões/Impressões: Uma vez que são apenas a chamada da função de caminhamento, com uma impressão isolada no fim, temos que a complexidade seria de O(n) + O(1) = O(n)

constroiInfixa() e constroiPosfixa(): De maneira semelhante à função avaliaExpressao(), essas passam por um loop de tamanho n, com algumas comparações e atribuições em cada etapa, ou seja, ordem O(n), porém no fim temos outro loop, que não consegue em nenhum caso ter tamanho maior do que n, ou seja, é no máximo O(n) também. No fim, as funções tem ordem O(n).

Calcula(): Também segue a lógica dos caminhamentos, resultando em ordem O(n).



4. Estratégias de Robustez

Para garantir mais robustez e a corretude do programa diante da entrada do usuário, vários testes são feitos. Em primeiro lugar, a expressão fornecida é avaliada com diversos critérios para decidir se é considerada válida ou não. Só em casos de expressão escrita sem nenhum erro lógico ou sintático o programa lê e armazena a expressão. As outras funções também só são executadas caso alguma expressão já tenha sido armazenada com sucesso. Caso contrário, mensagens de erros são imprimidas no local

esperado da saída. Alguns try's e catch's também foram utilizados para tratar casos como divisões por zero e tentativas de converter palavras ou operadores para números.

5. Análise experimental

Para evitar vazamentos de memória, utilizei o programa Valgrind para identificar possíveis erros. Em alguns momentos foi possível encontrar blocos de memória que não estavam sendo liberados adequadamente, como nesse momento:

```
==10493== HEAP SUMMARY:
==10493==     in use at exit: 21,906 bytes in 36 blocks
==10493==   total heap usage: 168 allocs, 132 frees, 112,940 bytes allocated
==10493==
==10493== 2,520 bytes in 35 blocks are definitely lost in loss record 1 of 2
==10493==    at 0x483BE63: operator new(unsigned long) (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memc
ux.so)
==10493==    by 0x10ABA9: avaliaExpressao(std::__cxx11::basic_string<char, std::char_traits<char>, std::allo
 (main.cpp:50)
==10493==    by 0x10B06A: main (main.cpp:97)
==10493==
==10493== 19,386 bytes in 1 blocks are still reachable in loss record 2 of 2
==10493==    at 0x483DD99: calloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==10493==    by 0x4B72971: monstartup (gmon.c:153)
==10493==    by 0x10A740: __gmon_start__ (in /home/davisakamoto/edds/TP1/bin/main)
==10493==    by 0x483B015: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==10493==    by 0x1FFF00005F: ???
==10493==    by 0x4011B5B: call_init.part.0 (dl-init.c:58)
==10493==    by 0x4011D15: call_init (dl-init.c:30)
==10493==    by 0x4011D15: _dl_init (dl-init.c:86)
==10493==    by 0x4001139: ??? (in /usr/lib/x86_64-linux-gnu/ld-2.31.so)
==10493==
==10493== LEAK SUMMARY:
==10493==    definitely lost: 2,520 bytes in 35 blocks
==10493==    indirectly lost: 0 bytes in 0 blocks
==10493==      possibly lost: 0 bytes in 0 blocks
==10493==    still reachable: 19,386 bytes in 1 blocks
==10493==         suppressed: 0 bytes in 0 blocks
==10493==
```

Algumas vezes também ocorria segmentation fault, e com o uso do valgrind em conjunto com o gdb, foi possível entender as causas dos problemas e consertá-los.

Também foram realizados alguns testes de tempo através da biblioteca "memlog".

6. Conclusões

O trabalho lidou com o problema de leitura, conversão e resolução de expressões númericas em diferentes notações, exigindo a criação de TADs cujas estruturas de dados representavam as expressões algébricas de forma sintática, utilizando de árvores.

Podemos concluir que o trabalho está consistente no que se diz respeito à complexidade em suas funções. Está funcional em respeito a realização dos quatro comandos possíveis e não está permitindo entradas inválidas ou comandos incorretos.

Houve dificuldades na criação da função para a leitura correta das expressões, mas após ter feito a leitura correta, não houve dificuldades para transformar a expressão posfixa para infixa e vice-versa, ou calcular o resultado da expressão. O programa está muito eficiente em questão de desempenho devido ao uso de funções simples somente para leitura e preenchimento.

Através do desenvolvimento das funções, promoveu-se um entendimento mais aprofundado com as análises de complexidade e tempo, reforçando a sua importância não só dentro da parte teórica, mas também como fator importante a ser levado em conta durante o desenvolvimento. Além disso, o exercício da implementação e do uso de estruturas de dados diferentes também foi muito praticado.

7. Bibliografia

- Ziviani, N., Projeto de Algoritmos com Implementações em Pascal e C, 3ª Edição, Cengage Learning, 2011
- Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle.
- Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009
- http://www.deinf.ufma.br/~csalles/comp/comp_parte1.pdf
- https://pt.wikipedia.org/wiki/Notação_polonesa_inversa
- Departamento de Ciência da Computação.
- Universidade Federal de Minas Gerais. Belo Horizonte.

8. Instruções para compilação e execução

O programa requer os arquivos de entrada na pasta bin.

Basta escrever no terminal “make all” que será compilado o programa e gerado os arquivos .o na pasta “obj” e um executável na pasta “bin”. Caso deseje testar entradas diferentes das que já estão lá, é necessário adicionar um arquivo de entrada na pasta bin e adicionar o comando de execução no Makefile no modelo

```
$(EXE) $(BIN)/entrada
```

Sendo que:

1- $(EXE) é o executável

2- $(BIN) é o endereço da pasta bin

3- “entrada” é o nome do arquivo de texto fornecido

Para limpeza dos arquivos objetos e do .run, basta escrever “make clean” no terminal.