Algoritmo e Estrutura de Dados III

Documentação do Trabalho Prático 0

Ramon Gonçalves Gonze

21 de abril de 2017

1. INTRODUÇÃO

A notação polonesa reversa é uma forma diferente do usual de escrever expressões matemática. Este trabalho prático consiste em criar um algoritmo para ler uma expressão em notação polonesa reversa e indicar quais as possíveis operações entre os números encontrarão um determinado resultado. O algoritmo que será apresentado nesta documentação realiza pequenas operações entre dois números (da esquerda para a direita na expressão matemática), salva o resultado no fundo da pilha, e continua realizando operações com números mais à direita da expressão. Após todas as operações, o número que estará no fundo da pilha, será o resultado da expressão.

2. IMPLEMENTAÇÃO

Para a implementação do trabalho foi utilizado um único TAD (Tipo Abstrado de Dado) que representa uma pilha. Foram criadas duas structs, de nomes *Pilha* e *Celula*, que compõe a estrutura de uma pilha. Cada célula da pilha possui um apontador para o anterior e para o próximo elemento da pilha. O conteúdo guardado em cada célula é um vetor de caracteres **e[MAX_EXPRESSAO]**, de tamanho 201.

2.1 FUNÇÕES E PROCEDIMENTOS

A utilização da pilha é feita através das seguintes funções:

Pilha* cria_pilha_vazia();

Esta função cria a cabeça da pilha. É alocada memória para uma nova célula, e seus apontadores *prox* e *ante* recebem **NULL**. Por conter somente atribuições, sua complexidade espacial e temporal é *O(1)*.

void empilha(Pilha *p, char buffer[]);

Esta função cria uma nova célula, atribui a sua variável **e** o conteúdo de **buffer** recebido no parâmetro, e a coloca no topo da pilha. Por conter somente atribuições e a criação de uma única nova célula, sua complexidade espacial e temporal é *O(1)*.

void faz_operacao(Pilha *p, char op);

O objetivo desta função é realizar as operações da notação polonesa inversa. Todos os números recebidos são lidos e empilhados na pilha. Podemos dividir a expressão inteira em partes menores, efetuando cada operação separada e salvando o resultado desta na própria pilha, para posteriormente ser utilizado em uma nova operação. Cada vez que é empilhado um '**?**' na pilha, é uma indicação que deve ser feita uma operação com os dois números que estão abaixo do '**?**' na pilha. A variável **op** contém o código da operação realizada:

- **0** para soma
- **1** para multiplicação

Após a operação ser feita, o resultado é salvo na posição do antepenúltimo número, contando do fundo para o topo da pilha. A penúltima célula e a última (topo) são apagadas. O diagrama abaixo descreve toda a operação:

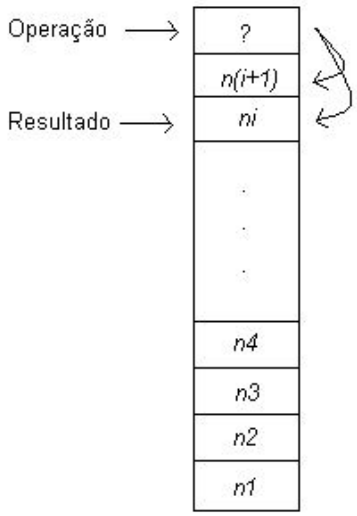

Figura 1: Representação da pilha de operações, sendo *n* números e '?' operações.

É feita a operação $n(i+1) + n$ ou $n(i+1) \times n$, e o resultado salvo na célula onde se encontrava o *ni*.Ao fim da última operação, restará a pilha com somente uma célula, que armazenará o resultado de toda a expressão. A simulação abaixo demonstra todo o processo em um exemplo:

Entrada: 2 3 ? 4 ? 5 6 ? ?

Execução do algoritmo para a máscara 0010 (++*+):

2 3 + 4 + 5 6 * +

5 4 + 5 6 * +

9 5 6 * +

9 30 +

39

Não há laços de repetição, somente atribuições, nem alocação de memória, portanto sua complexidade espacial e temporal é *O(1)*.

void destroi_pilha(Pilha *p);

Esta função percorre toda a pilha, e libera todo o espaço de memória alocado para ela. Cada célula é percorrida uma vez através de um laço **while**. Sua complexidade temporal é *O(n)*, sendo *n* a quantidade de células que a pilha possui, e sua complexidade espacial é *O(1)*.

Programa principal

Para a resolução do problema proposto pelo trabalho prático, o método utilizado para encontrar as possíveis operações que obtém o resultado desejado é o força bruta. São testadas todas as operações possíveis e o resultado final comparado. Para cada teste de operações, é criada uma máscara binária, um vetor de caracteres que contém somente 0's e 1's, no qual *0* equivale a operação soma (+) e *1* equivale a operação multiplicação (×). A quantidade de testes depende da quantidade de operações que a expressão contém. Para cada '**?**' lido, é possível que ele seja um + ou um ×, logo 2 opções. Se há 3 '**?**', teremos 2×2×2 = 8 possibilidades possíveis. Logo, a quantidade de testes é 2^n, sendo *n* a quantidade de '**?**' na expressão. É gerada uma string binária em ordem

crescente, conforme o exemplo abaixo:

- Para 3 operações, temos máscaras de 3 bits:

  1ª máscara gerada: 000 = +++ → correspondente ao número 0 em decimal

  2ª máscara gerada: 001 = ++* → correspondente ao número 1 em decimal

  3ª máscara gerada: 010 = +*+ → correspondente ao número 2 em decimal

  E assim sucessivamente, até a 8ª máscara (3 operações, $2^3 = 8$ testes).

O laço de repetição que cria a máscara, gera *n* bits (quantidade de operações da expressão). Sua complexidade espacial e temporal é *O(n)*.

Após a mascará ser gerada, o vetor **expressão[]** é percorrido, e as operações são efetuadas com o auxílio da pilha. Como a expressão possui no máximo 200 caracteres, então a complexidade temporal e espacial deste laço será *O(1)*. Se ao final, o resultado for o mesmo que o informado na entrada, então a operação é válida, e ela é exibida na tela. Há um laço que percorre a máscara, imprimindo na tela '+' ou '*', para 0's e 1's respectivamente. As máscaras testadas são geradas em ordem lexicográfica, logo a saída também estará ordenada. Como o tamanho da máscara é *n*, então a complexidade temporal deste laço será *O(n)*, e a complexidade espacial será *O(1)*, pois não há alocação de memória.

A complexidade final da função **main** pode ser resumida da seguinte forma:

main()

laço principal $O(2^n)$

laço da máscara de bits *O(n)*

laço das operações *O(1)*

laço da impressão do resultado *O(n)*

destroi_pilha *O(n)*

Complexidade temporal final: $O(n\,2^n)$.

Complexidade espacial final: *O(n)*.

3. AVALIAÇÃO EXPERIMENTAL

A análise empírica dos dados permite avaliar que, na prática, a partir de 25 operações, o tempo de execução é extremamente alto, pelo fato da complexidade temporal do algoritmo ser exponencial. A execução dos testes foi realizada em um computador com processador Intel Core i3, 2.6GHz e 6 GB de RAM. O sistema operacional utilizado foi o Ubuntu 16.04 LTS. O desempenho do algoritmo pode ser demonstrado no Gráfico 1.

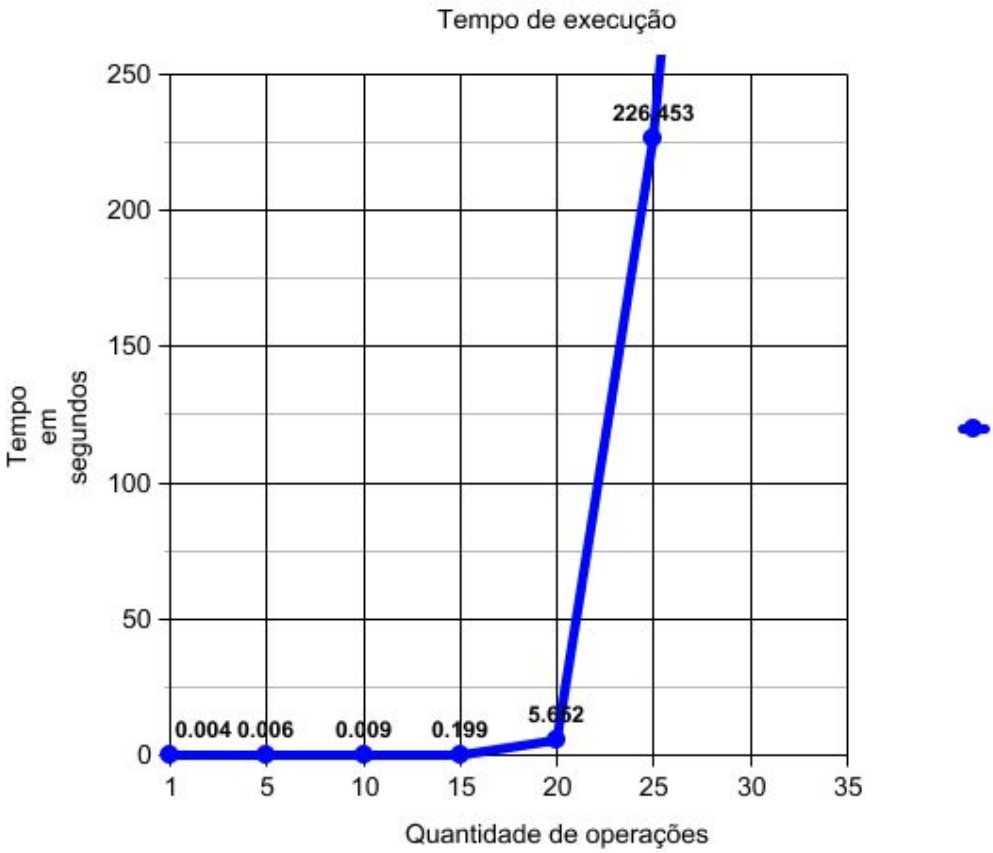

Gráfico 1: Testes de entradas com diferentes quantidades de operações

4. CONCLUSÃO

O objetivo do trabalho prático foi atingido, conforme a especificação do roteiro. As alocações dinâmicas e suas liberações foram feitas corretamente, segundo o

depurador Valgrind. Os 9 testes toys disponibilizados foram testados, e a saídas obtidas pelo algoritmo foram idênticas às fornecidas. O trabalho possui 3 arquivos: *tp0.c*, *pilha.c* e *pilha.h*. Está sendo entregue também um *Makefile*. Para compilar o programa, é necessário abrir o terminal (Linux), acessar o diretório que contém os 4 arquivos e utilizar o comando **make**. Será gerado um arquivo executável de nome **tp0**.