Notação Polonesa Reversa

Luiz Otávio Resende Vasconcelos

Abril 2017

1 Introdução

De vasta utilização nas ciências exatas, com destaque para Ciência da Computação, a Notação Polonesa Reversa foi inventada pelo filósofo e cientista da computação australiano Charles Hamblin em meados dos anos 1950, para habilitar armazenamento de memória de endereço zero. Ela deriva da notação polonesa, introduzida em 1920 pelo matemático polonês Jan Łukasiewicz. (Daí o nome sugerido de notação Zciweisakul.) Hamblin apresentou seu trabalho numa conferência em Junho de 1957, e o publicou em 1957 e 1962.

O presente trabalho foi desenvolvido na linguagem C.
Para executá-lo, basta utilizar os comandos em um bash/unix:
$ make
$ ./polish_notation_reverse

O projeto completo e as etapas do desenvolvimento podem ser encontrados no repositório:
https://github.com/Luiz0tavio/polish_notation_reverse

2 Implementação

2.1 Estrutura de Dados

A estrutura de dados principal para realizar o cálculo escolhida foi uma árvore binária. As primeiras operações são armazenadas nas folhas e a última, na raiz.

Cada nó da árvore possui um struct o qual contém as propriedades:
id : contador para identificação do nó.
grouped : boolena que indica se o nó foi agrupado.
a : primeiro número.
b : segundo número.
ind_a : índice para o primeiro filho.
ind_b : índice para o segundo filho.

Os índices descritos acima referem-se à um array que armazena os structs. A

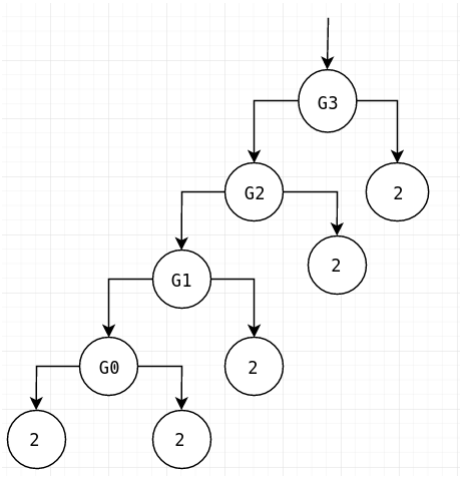

Figura 1: Árvore binária gerada para 2 2 ? 2 ? 2 ? 2 ?

árvore é montada apenas com um elemento fazendo referência ao outro. A escolha da utilização de índices e não posições de memória é devido à utilização de **realloc** (o qual move todo o conteúdo de lugar na memória e da free na memória antiga, logo, seria necessário atualizar cada endereço após o realloc o que seria muito custoso).

2.2 Construção da Árvore

Para a construção da árvore é utilizado um array principal em conjunto com a string de entrada.

A string é lida e cada caracter válido é armazenado em um array de duas posições - cada inserção é feita com mod 2 - até ser encontrado um '?'(operação perdida). Então, temos um array com os dois últimos caracteres válidos. Esse array irá gerar uma nova estrutura de dados.

Caso o array contenha:

2 números, será gerado uma nova folha.

2 'G's, será gerado um nó com dois sub-nós (a função dos caracters 'G' e 'E' será explicada à frente).

1 'G' e um número, será gerado um nó apontando para um sub-nó e para o número.

Após gerado, uma modificação na string original é necessária para que ela possa ser reutilizada.

O caractere '?' é substituído por um 'G' (o que significa que aqueles operadores foram agrupados em uma nova estrutura de dados). Os operadores

utilizados são trocados por 'E' (caractere escolhido para ser ínválido).

Ex.: Utilizando o exemplo(reduzido) da documentação:

2 2 ? 2 2 ? 2 2 ? ? ?

1ª iteração:

E E G 2 2 ? 2 2 ? ? ?

2ª iteração:

E E G E E G 2 2 ? ? ?

3ª iteração:

E E G E E G E E G ? ?

4ª iteração:

E E G E E E E E E G ?

aqui, os dois últimos 'G's são agrupados em um novo 'G' que será a raiz.

A última iteração irá agrupar os últimos 2 'G' e a string estará completa com 'E's.

2.3 Operações

São possíveis 2^n operações para cada entrada, onde n é a quantidade de '?'(operações perdidas) presentes na string de entrada. Para gerar as 2^n é feita uma máscara de bits.

Ex: para n = 2

00
01
10
11

então, os zeros são substituidos por '+' e os uns por '*'. Assim, a prioridade para os operadores '+' já é sanada.

2.4 Cálculo

Para cada um dos grupos de operadores gerados, é feita uma busca em profundidade recursiva: as operações são aplicadas das folhas até a raiz. Os operadores são passados em cada recursão, um a um, começando do último até o primeiro.

Após a recursão terminar, será retornado um valor. Caso seja igual ao passado na segunda linha da entrada, exibe a sequência de operadores. Caso contrário, continua para a próxima sequência.

3 Estrutura do Projeto

Apesar do código estar bem comentado, segue uma breve exposição de como o projeto está particionado.

3.1 main.c

Arquivo principal do projeto. Inicializa as varíaveis, faz a leitura das entradas, faz a chamada das principais funções e exibe o resultado.

3.2 operation.h / operation.c

Responsável por criar e realizar operações com a estrutura de dados componente de cada nó da árvore.

Também é onde está a função recursiva que irá apresentar o resultado.

3.3 stringBuffer.h / stringBuffer.c

Responsável por executar operações com a string de entrada que é utilizada para montar a árvore.

4 Análise Assintótica

4.1 Temporal

No desenvolvimento encontramos 2 loops principais: o primeiro, para manipular a string de entrada e gerar a árvore e o segundo, para fazer o cálculo e escrever na tela.

No primeiro loop principal temos a complexidade $\theta(\mathbf{n})$ com limite assintótico firme em n, onde n é o número de caracteres não vazios da string, pois, será lidos e processados um a um. Há outros outros loops dentro, porém, de limite constante e portanto desprezível.

No segundo loop temos 2^m operações, onde m é o número de caracteres '?' da entrada.
Dentro, temos uma chamada de função que converterá o índice atual do loop em uma sequencia de '+'s e '*'s de tamanho m.
Por fim, fazemos a chamada recursiva na árvore. O custo de operações numa árvore aleatoriamente distribuída é O(log m), e no pior caso O(m)

Podemos então concluir que, no caso médio, o algorítmo tem um comportamento assintótico O(n + 2^m m log m) e no pior caso, O(n + 2^m m^2)

4.2 Espacial

Para a entrada alocamos 51 bytes, para os operadores 16 bytes. Então, temos 5 int's auxiliares que nos custam 40 bytes. Para cada caractere '?' geramos um nó da árvore de tamanho 24 bytes

Podemos então concluir que temos um custo de espaço total de **(91 + m * 24) bytes**

5 Avaliação Experimental

Foram gerados casos de teste limitando os 200 caracteres.
Geramos um casos de teste de 0 a 26 operadores. O tempo de execução por caso de teste é mostrado na Figura 2. Como previsto na análise assintótica

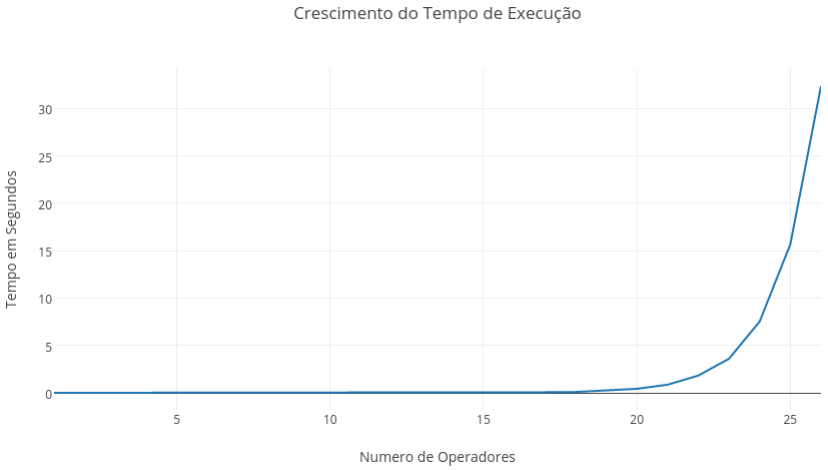

Figura 2: Gráfico Operações / Tempo de execução

temporal, com a complexidade O(n + 2^m m log m), enquanto m é pequeno (no gráfico até cerca de 18), ela se comporta com crescimento linear, ou seja O(n). Passando de um certo ponto, nesse caso cerca de 18, a função exponencial a supera e então, apresenta um crescimento exponencial.

6 Conclusão

A notação polonesa reversa se mostra como uma grande aliada na computação. Entre suas vantagens, podemos citar:

1. Reduz o número de passos lógicos para se perfazerem operações binárias e, posto que as demais operações são ou binárias puras compostas, ou binárias compostas com unitárias ou apenas unitárias, o número total de passos lógicos necessários a um determinado cômputo será sempre menor que aquele que utiliza a sintaxe convencional (lógica algébrica direta)
2. Trabalha com pares ordenados a priori, somente definindo a lei de composição binária aplicável após a eleição e a introdução do desejado par no

cenário de cálculo. Até o momento final, se poderá decidir pela troca ou pela permanência da operação original.

3. Minimiza os erros de computação, automática ou manual assistida;

4. Maximiza a velocidade operacional na solução de problemas.

Apesar da abordagem com pilha ser mais eficiente, a escolha da árvore binária mostrou grande desempenho e facilidade na implementação.