Trabalho Prático I - Estruturas de Dados

Alan Borges Martins Bispo [1]

[1] DCC – UFMG

abmbispo@dcc.ufmg.br

Abstract. *The following paper describes the solution for the problem suggested on Trabalho Pratico 1 of the Data Structure course given by DCC-UFMG. The problem addresses a situation in which Rick Sanches is a lab assistant who has as one of his tasks to make measurements in vials of predetermined sizes, a program must be developed that can calculate the minimum number of vial movements (put/remove) to achieve a desired volume X value in milliliters from a predetermined amount of vials that Rick can use in the operations. The software was developed in C++ and its algorithm will be explain below with the breakdown of its complexity function.*
Keywords*: linked List, Data Structure, C++*

Resumo. *O seguinte relatório descreve a forma pela qual foi resolvido o problema proposto pelo Trabalho Prático 1 da disciplina Estruturas de Dados ofertado pelo DCC - UFMG. O problema proposto aborda uma situação na qual Rick Sanchez é um ajudante de laboratório e possui como uma de suas tarefas fazer medições em frascos de tamanho predeterminados; desta feita, é necessário que se faça um programa que calcule para Rick o número mínimo de operações de tira/coloca dos frascos para atingir um desejado valor X de volume em mililitros a partir de uam quantidade pre determinada de frascos que Rick pode usar nas operações. O programa foi desenvolvimento em C++ e o seu algoritmo é explicado a frente com a discriminação da sua função de complexidade.*
Palavras-chave: Lista encadeada. Estruturas de dados. C++.

1. Implementação

De maneira geral utilizou-se nesse TP a implmentação de uma estrutura de dados conhecida por Lista Encadeada. A necessidade de se valer de uma estrutura de dados desse tipo vem do fato de não sabermos ao certo o tamanho das entradas e, como vermos mais a frente, da quantidade de possibilidades de somas dos frascos. Para isso usamos as seguintes classes:

- List;
- Node;

Abaixo será descrito o funcionamento de cada uma dessas classes e como elas juntas formam a nossa estrutura de dados principal do programa, a Lista Encadeada.

1.1. Node

A classe Node serve como os nós da lista encadeada. Assim sendo ela deve guardar em cada objeto os seguintes atributos, com seus respectivos tipos dentro dos parêntesis: *count_operations (int), mililiters (int), next_node (Node*), previous_node (Node*)*. Do ponto de vista de atributos específicos para a resolução do problema dos frascos do Rick, o meu atributo *count_operations* serve para guardar a quantidade de operações necessárias para se chegar no resultado em mililitros do atributo *mililiters*; os atributos *next_node* e *previous_node* são para a nevegação para frente ou para trás na lista, que é duplamente encadeada.

Essa classe conta com funções *getters* e *setters* dos seus atributos, de forma que o *design pattern* escolhido para o TP prioriza as *best practices* do C++, ou seja, é impossível a partir do escopo dos objetos fazer o acesso diretamente nos atributos do objeto, mas há de se passar antes pelas funções de acesso.

1.2. Lista

A classe Lista será como que o nosso orquestrador dos nós. Ela terá a referência do início e do fim da lista. Seus atributos são: *head (Node*)* e *tail (Node*)*. Possui funções de inserir elementos na lista (*void insert(int mililiters, int count_operations)*), remover elementos da lista (*void remove(int mililiters)*), funções helpers de impressão do estado atual da lista, além de para pegar o primeiro nó de uma lista (*Node* get_first_node()*), remover a primeira posição da lista e retorná-la (*Node* remove_front()*) e limpar a lista (*void clean()*). Além disso há na Lista uma função para chegar do resultado (*int find_solution(int mililiters)*) que será explicada melhor mais a frente.

1.3. Algoritmo

O início do programa se dá num while que verifica a entrada via standard in. Enquanto a entrada não é igual a EOF (ctrl + D), então ele continua lendo os valores com base na específicação. Para cada linha são lidos dois valores no seguinte formato: inteiro (a quantidade de mililitros), um espaço em branco e um char (a operação a ser feita, inserção, remoção ou solução). Então uma estrutura de if-else faz a verificação de qual operação é selecionada e chama a respectiva função.

Com a lista de frascos preenchida pelos testes, o algoritmo para a resolução da quantidade mínima de operações para se chegar ao resultado funciona criando uma lista de possibilidades de frascos (isto é, uma *List possibilities*) idêntica a inicial preenchida pelos testes. Então é feita uma primeira verificação trivial onde se busca o resultado na própria lista, sem ter de passar pelo algoritmo. Não se encontrando, segue-se para a iteração até que se encontre um resultado satisfatório.

Esse passo é uma esrutura de repetição aparentemente eterna (*while(true)*) que trabalha basicamente com dois nós a cada iteração: um nó da lista original de frascos e um da lista de possbilidades; ademais, itera-se sobre a lista de frascos para cada possibilidade na lista de possbilidades, e quando se verifica que nem a soma nem a subtração dos dois frascos da fase atual da iteração são o resultado, adiciona-se eles a lista de possibilidades com um *count_operations++*. Essa operação é repetida infinitamente (por isso temos a aparência de eternidade) até que se encontre o resultado; como é garantido que todas as entradas geram uma combinação válida em algum ponto, então não há problema em fazer isso.

2. Instruções de compilação e execução

O makefile preparado para a execução do código tem as seguintes opções:

1) make: Compila os arquivos de código fonte em umm arquivo chamado *"tp1"*.
2) make mem: Executa o arquivo gerado pela compilação em modo de depuração de memória, assim podemos nos certificar que não existe nenhum vazamento de memória acontecendo durante a execução.
3) make test: Executa os script de testes do trabalho.
4) make clean: Limpa a pasta contendo as saídas da compilação.

3. Complexidade

Seja n o número de frascos disponíveis (ou o tamanho da lista de frascos) e m o número de movimentos necessários para que a solução seja encontrada, temos que:

O código consome nossa lista de operações de maneira contínua, de forma que para cada operação inícial que temos na lista, teremos duas vezes esse tamanho para novas operações. Assim obtemos que a iteração é da forma:

- $n \times 2n \times 2n \ldots$

No entanto, como a iteração ocorrerá um número *m* de vezes que corresponde à quantidade de iterações para se chegar ao resultado, temos que:

- $n \times 2n^{m-1}$

Acima está nossa função de complexidade. Assintóticamente obtemos:

- $O(n^m)$

Isso significa que nosso algoritmo é relativamente eficiente quando se trata de um crescimento em *"n" (quantidade de frascos)*, no entanto o cenário fica um tanto quanto pior quando se trata de um crescimento em *"m"* (quantidade de iterações para se chegar a um resultado).

4. Conclusão

Desta feita, nós somos incapazes de acertar quais casos o algoritmo é ou não eficiente, uma vez que é justamente isso que o algoritmo busca conseguir encontrar de certa forma. Concluimos então que seria mais interessante uma modificação no algoritmo de implementação da solução deste problema.