

INF 213 - Roteiro da Aula Prática 11

Objetivo: praticar hashing.

→ LEMBREM-SE DE USAR PAPEL E CANETA COMO RASCUNHO ANTES DE IMPLEMENTAR <<--

Arquivos fonte e diagramas utilizados nesta aula:

https://drive.google.com/open?id=1UcmSOc6ssjpfe71S_-Ws3qAEWglHgib

Etapa 1

Hashing possui inúmeras aplicações. Exemplos:

- Estrutura de dados tabela hash.
- Armazenamento seguro de senhas em banco de dados.
- Cálculo de verificadores (exemplo: dígito verificador de CPF, md5sum de arquivos).

No Brasil, o CPF é um número composto por 9 dígitos + 2 dígitos verificadores. Assim como no caso do md5sum, o objetivo desses dígitos verificadores é reduzir a chance de aceitação de valores errados (quando um usuário fornece um número de CPF, por exemplo, podemos calcular os dígitos verificadores dos 9 primeiros números e, assim, verificar se houve algum erro de digitação).

Antes de prosseguir, pense nas seguintes perguntas (e discuta com um colega/professor):

- 1) se os dígitos verificadores não corresponderem aos fornecidos pelo usuário, isso significa que houve algum erro de digitação (supondo, claro, que o usuário tentou digitar seu CPF corretamente)?
- 2) se os dígitos verificadores corresponderem aos fornecidos pelo usuário, isso significa que não houve erros de digitação?

O “hash” do CPF é calculado da seguinte forma: multiplica-se cada dígito do CPF (da esquerda para a direita) por um dos números 10, 9, 8, ..., 2 e, então, calcula-se a soma S desses produtos.

A seguir, calcula-se o resto da divisão entre $10S$ e 11.

Exemplo, se os 9 primeiros dígitos de um CPF for: 123555876 (123555876-20, com os dois dígitos verificadores), então $S = 1 \cdot 10 + 2 \cdot 9 + 3 \cdot 8 + 5 \cdot 7 + 5 \cdot 6 + 5 \cdot 5 + 8 \cdot 4 + 7 \cdot 3 + 6 \cdot 2 = 207$

O primeiro dígito será: $(10 \cdot 207) \% 11 = 2$ (se o resto for 10 → o primeiro dígito será 0)

O segundo dígito é calculado de forma similar, mas agora começando do 11 e considerando os 10 primeiros números (9 originais + o primeiro dígito verificador calculado acima). Assim, o segundo dígito verificador do CPF 123555876 será:

$$S = 1*11 + 2*10 + 3*9 + 5*8 + 5*7 + 5*6 + 8*5 + 7*4 + 6*3 + 2*2 = 253$$

O segundo digito sera: $(253*10)\%11 = 0$

Crie um programa (com nome **geraDigitosCPF.cpp**) que le um numero N e, entao, N strings (cada uma representando um CPF composto de 9 digitos). Seu programa devera, entao, imprimir N linhas contendo, em cada linha, os dois digitos verificadores do CPF correspondente.

Entrada	Saida esperada
4	20
123555876	09
123456789	11
111111111	12
078123587	

Etapa 2

Considere a classe MyHashMap (similar a vista em sala).

Termine a implementacao do método set e teste sua implementacao usando o programa testaMyHashMapString.cpp.

Etapa 3

Considere o programa testaMyHashMapTamanhoBaldes.cpp.

Compile-o e execute-o redirecionando o arquivo Frankenstein.txt para sua entrada padrao (i.e., vamos guardar as palavras do livro Frankenstein em uma tabela hash e ver a uniformidade com que nossa funcao hash distribui as palavras na tabela).

Para executar, digite "time ./a.out 1 < Frankenstein.txt" (o numero 1 indica que usaremos a primeira implementacao da funcao hash ; o comando time sera utilizado para medir o tempo de execucao do programa)

A seguir, implemente pelo menos mais 4 funcoes hash distintas (use um numero distinto para identificar cada uma) e veja como elas se comportam (avale o tempo de execucao do programa usando o programa time e verifique tambem a distribuicao das chaves nos baldes). Tente criar tanto funcoes hash boas quanto ruins.

Finalmente, mude o tamanho padrao da tabela hash para 1000 e meca os tempos (para cada funcao hash) novamente.

Anote os tempos e suas conclusões em uma folha de papel (ou crie um arquivo para isso).

Discuta suas conclusões e as vantagens/desvantagens de cada função hash com o professor. (esta etapa não será avaliada de forma automática pelo Submittity)

Etapa 4

Uma importante operação em tabelas hash é a operação de redimensionar a tabela. A ideia é que se os baldes ficarem muito cheios a tabela poderá ser recriada com um tamanho (número de baldes) maior (o objetivo é que, com uma tabela maior, espera-se que cada balde tenha menos elementos).

A operação de rehashing pode ser executada de forma automática (ou seja, a tabela hash chama a operação de rehashing quando detectar, por exemplo, que os baldes estão “cheios demais”) ou manual (nesse caso o usuário deverá escolher manualmente o novo tamanho da tabela).

Implemente uma função “reHash(int newSize)” que, dado um novo tamanho para a tabela hash, a redimensiona para o novo tamanho (esse tamanho poderá ser menor, maior ou igual ao atual!).

Note que, ao redimensionar uma tabela, todas as chaves devem ser reinseridas nos novos baldes corretos.

Por exemplo, suponha que usamos uma função hash para strings que retorna 0 para strings que começam com a letra a, 1 para strings que começam com b, 2 para strings que começam com c e assim por diante. Em uma tabela hash de tamanho 4 as palavras “abc” e “elefante” seriam colocadas no balde 0. Porém, se a tabela fosse redimensionada para tamanho 10, então “abc” iria para o balde 0 e “elefante” iria para o balde 4.

Submissão da aula prática:

A solução deve ser submetida até as 18 horas da próxima Segunda-Feira utilizando o sistema submittity (submittity.dpi.ufv.br). Envie todos os arquivos fonte (tanto os arquivos .h e .cpp fornecidos neste laboratório quanto os que você implementou). Atualmente a submissão só pode ser realizada dentro da rede da UFV.