CES-12 - Relatório Lab 1 - Hash

Aluno: Bruno Costa Alves Freire

29 de Abril de 2020

1.

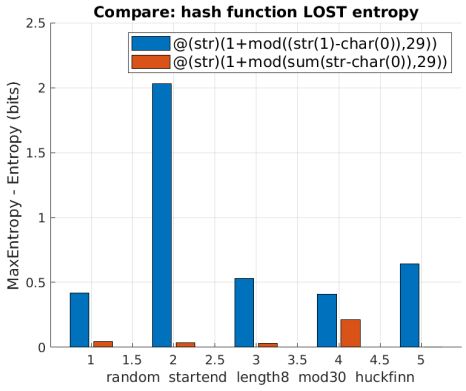
(1.1) (pergunta mais simples e mais geral) Por que necessitamos escolher uma boa função de *hashing*, e quais as consequências de escolher uma função ruim?

O objetivo de se utilizar uma hashtable é armazenar os dados de maneira mais homogênea na memória, visando assim viabilizar uma execução rápida de todas as operações usuais de dicionários (inserções, buscas e deleções). É possível fazer todas em $O(1)$ sob as devidas hipóteses (assumindo que o tamanho da tabela é proporcional à quantidade de elementos inseridos, utilizando listas duplamente ligadas, e claro, assumindo o caso médio). A principal hipótese para que o esquema de hash funcione bem, é que nossa função de hashing distribua as chaves dos elementos armazenados de maneira mais homogênea possível, evitando ao máximo que haja colisões nos slots. O problema das colisões é que elas são responsáveis por aumentar os tempos de buscas, inserções e deleções. Se uma função de hashing provoca muitas colisões para um conjunto arbitrário de dados, ela irá aumentar o tempo médio de cada uma dessas operações, ao mesmo tempo em que irá subutilizar o espaço da tabela.

(1.2) Por que há uma diferença significativa entre considerar apenas o 1º caractere ou a soma de todos?

Em geral, uma função de hashing que incorpore mais informação das chaves para calcular sua posição é superior a uma função que incorpore menos informação. Nesse caso, como utilizamos uma função de hashing por divisão de tamanho 29, e o alfabeto utilizado possui apenas 26 letras (ainda que outros símbolos possam aparecer, em geral as letras do alfabeto são muito mais comuns), não apenas teremos alguns buckets sistematicamente subutilizados, como também teremos alguns buckets especialmente sobrecarregados devido ao fato de que palavras com determinadas iniciais são mais comuns do que outras (não são muitas palavras que começam com 'z' ou 'x' por exemplo). Por outro lado, a soma dos caracteres é algo com uma distribuição bem menos concentrada, devido aos diferentes comprimentos possíveis e também a ocorrência mais democrática das letras ao longo das palavras.

Estatisticamente, podemos constatar essa diferença observando os gráficos das figuras 1 e 2.



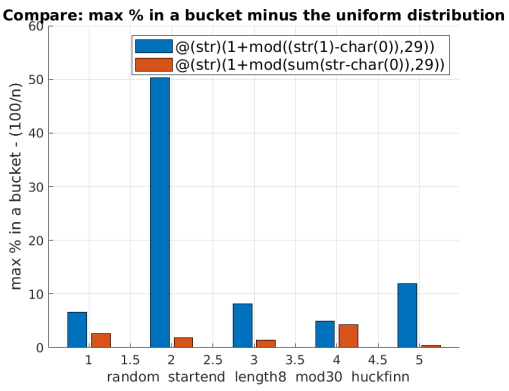


Figura 1: Perda de entropia da distribuição dos datasets para as duas funções de hashing.

Figura 2: Ocupação dos buckets para cada dataset com as duas funções de hashing.

(1.3) Por que um *dataset* apresentou resultados muito piores do que os outros, quando consideramos apenas o 1º caractere?

O dataset `startend.txt` apresenta, a partir da linha 185 (de um total de 700), longas sequências de palavras que começam com a mesma letra (várias com 'c', e em seguida blocos com palavras iniciando com as primeiras letras do alfabeto). É claro que levando em conta apenas a primeira letra, a maior parte das palavras é hasheada num pequeno número de slots. Essa má distribuição pode ser visualizada na figura 3. Os demais datasets apresentam uma distribuição melhor para essa função, com entropias maiores que 4 bits.

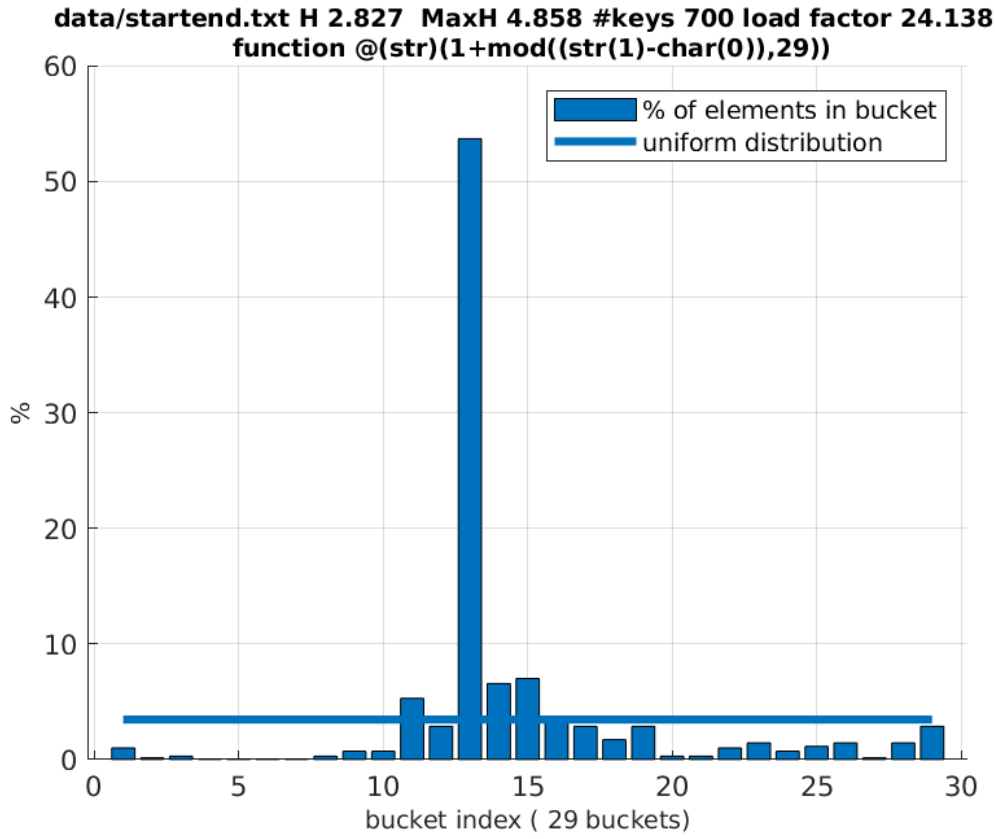


Figura 3: Distribuição do dataset `startend.txt` nos buckets com a função de hash mod1stChar29.

2.

(2.1) Com uma tabela de hash maior, o hash deveria ser mais fácil, afinal temos mais posições na tabela para espalhar as strings. Hash com tamanho 30 não deveria ser sempre melhor do que com tamanho 29? Por que não é este o resultado? (atenção: o arquivo mod30 não é o único resultado onde tamanho 30 é pior do que tamanho 29)

De nada adianta ter uma tabela maior se a função de hash associada não tiver uma distribuição mais uniforme das chaves armazenadas. No caso, uma eventual diferença de desempenho pode ocorrer a depender dos datasets. Um dataset pode eventualmente estar melhor distribuído com respeito a uma função de hashing do que outra. Um caso extremo é exemplificado pelo dataset em `mod30.txt`, o qual foi claramente construído de modo a concentrar a maior parte das chaves em um único slot através da função de hash com tamanho 30. Fora esse dataset, a diferença nas perdas de entropia e na concentração das chaves nos buckets entre as funções de hash por divisão de tamanho 29 e 30 foram ínfimas.

(2.2) Uma regra comum é usar um tamanho primo (e.g. 29) e não um tamanho com vários divisores, como 30. Que tipo de problema o tamanho primo evita, e porque a diferença não é muito grande no nosso exemplo?

A preferência por tamanhos primos está relacionada à quantidade de subgrupos próprios do grupo aditivo $\mathbb{Z}_n$. Em outras palavras, quando usamos um tamanho com algum divisor, chaves que tenham valores múltiplos desse divisor serão sempre hasheadas a um subgrupo especifico dos slots. Por exemplo, se usarmos o tamanho 30, e tivermos muitas chaves com tamanhos múltiplos de 3, todas essas chaves cairão em slots múltiplos de 3 (0, 3, 6, ...), dessa forma ocupando apenas um terço da tabela. Para cada divisor do tamanho da tabela, teremos associado um subgrupo de slots. Com muitos divisores, serão muitos os valores de chaves que serão hasheados para o mesmo subgrupo de slots. Para reduzir esse problema, tomamos números com a menor quantidade possível de divisores, os primos.

Agora, perceba que a presença de divisores só é um problema se considerarmos que as chaves ocorrem com frequência em múltiplos de um dado divisor. Num dataset ideal, as chaves são distribuídas de maneira mais uniforme, não causando o problema mencionado. No caso presente, todos os datasets exceto o `mod30.txt` são razoavelmente equilibrados nesse sentido.

(2.3) note que o arquivo mod30 foi feito para atacar um hash por divisão de tabela de tamanho 30. Como este ataque funciona?

Conforme podemos ver na figura 4, o dataset `mod30.txt` foi construído de modo que aproximadamente 74% das chaves são hasheadas para o mesmo bucket (no caso, o 4º slot). Isso é feito escolhendo palavras cuja soma dos caracteres módulo 30 deixe sempre o mesmo resto, no caso 3.

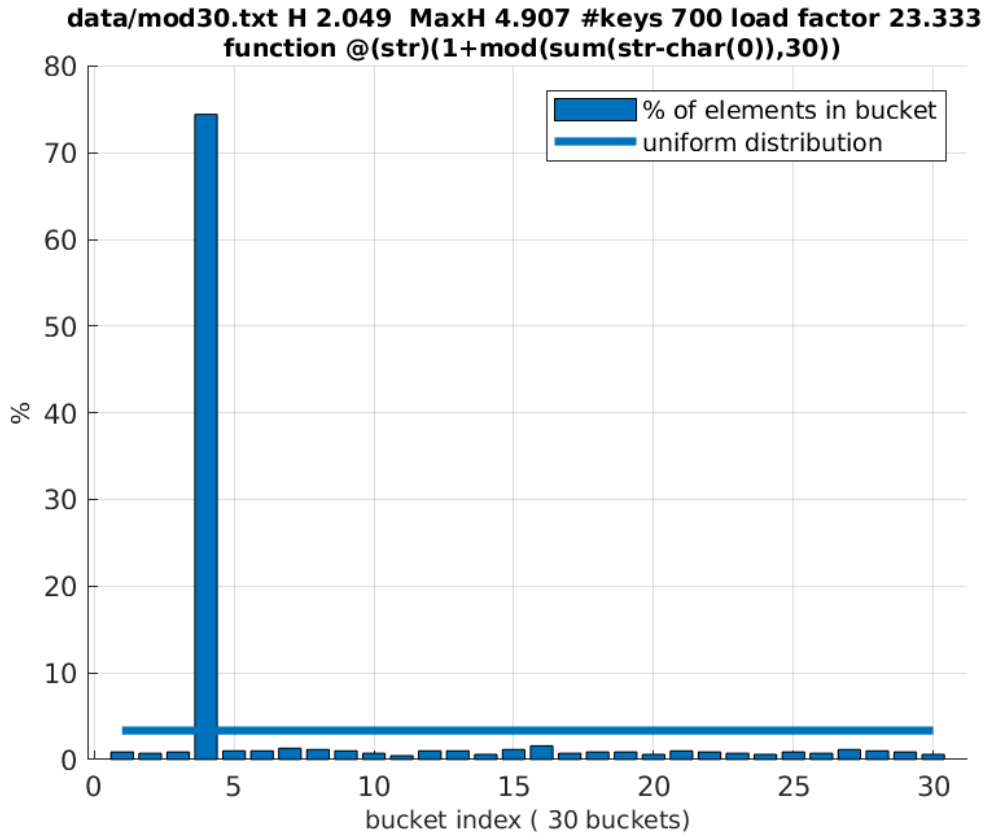


Figura 4: Distribuição do dataset `mod30.txt` nos buckets com a função de hash modsumchar30.

3.

(3) Com tamanho 997 (primo) para a tabela de hash ao invés de 29, não deveria ser mais fácil? Afinal, temos 997 posições para espalhar números ao invés de 29. Por que às vezes o hash por divisão com 29 buckets obtém melhores resultados do que com 997? Por que a versão com produtório (prodint) é melhor? Por que este problema não apareceu quando usamos tamanho 29?

Analisando a tabela hash para o dataset `length8.txt`, podemos notar que todas as palavras se concentraram numa faixa de cerca de 100 slots para o tamanho de 997, conforme vemos na figura 5. Ocorre que nesse dataset, todas as palavras tem 8 letras (e em sua maioria são letras minúsculas). Ocorre que os dígitos do alfabeto estão numa faixa entre 65 e 122 da tabela ASCII, de modo que o valor da função de hash fica entre 520 e 976. Considerando ainda que as minúsculas são mais comuns e ficam na metade superior (entre 97 e 122), teremos uma concentração maior entre 776 e 976. Ou seja, a faixa de valores da função de hash nesse dataset é muito pequena para o tamanho da tabela. O mesmo não ocorre com o tamanho 29, uma vez que ao tomarmos o resto módulo 29, cobrimos a tabela toda várias vezes.

Repetindo essa análise para o dataset `mod30.txt`, notamos que as palavras se concentram em alguns slots esparsos, igualmente espaçados, vide a figura 6. Como mencionado anteriormente, quase todas as palavras desse dataset possuem a soma dos caracteres da forma $30 \cdot k + 3$. Como o comprimento das palavras é limitado, há uma quantidade limitada de valores para esse k. Estimando o comprimento máximo das palavras em 20, e o valor ASCII máximo dos caracteres em 120, teríamos somas de até 2400, mas como os valores estão espaçados de 30 em 30, teríamos então no máximo 80 valores possíveis para as somas. Por essa razão, vemos alguns picos dispersos na hash table de tamanho 997. Quando usamos o tamanho 29, isso não era tão aparente pois os picos eventualmente cobriam a tabela toda.

Por fim, analisando o dataset `huckfinn.txt`, o qual se trata de uma transcrição de uma obra literária, temos o aspecto da figura 7. Aparentemente há um padrão de "franjas" na ocupação dos

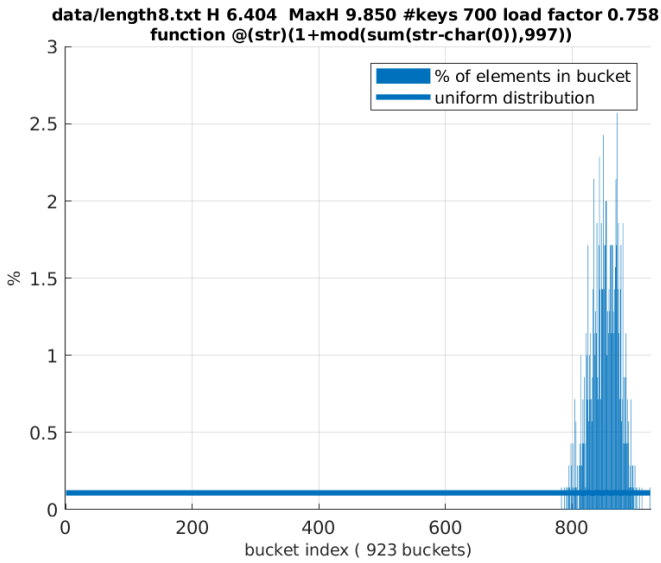


Figura 5: Distribuição do dataset `length8.txt` nos buckets com a função de hash modsumchar1k.

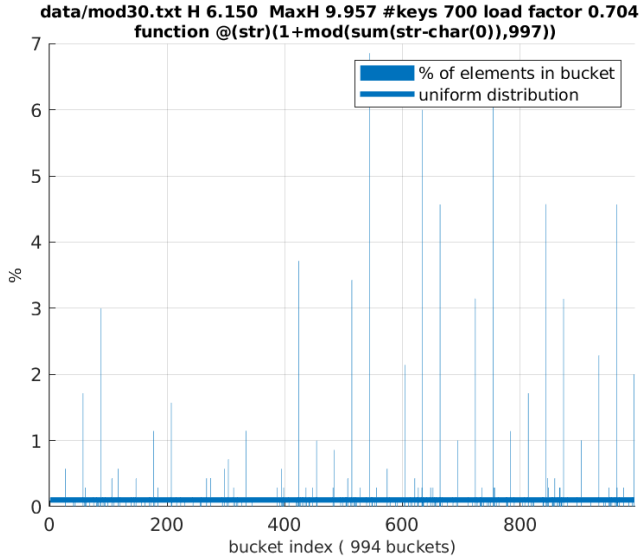


Figura 6: Distribuição do dataset `mod30.txt` nos buckets com a função de hash modsumchar1k.

buckets. Uma possível explicação é que as palavras mais comuns da língua são formadas por alguns caracteres mais frequentes, contidos numa faixa curta da tabela ASCII, e que cada franja seja correspondente (aproximadamente) às palavras de um determinado tamanho. Então poderíamos pensar que as palavras que possuem entre 4 e 7 letras são mais comuns. Novamente, não observamos esse efeito na tabela de tamanho 29 pois essas franjas eventualmente cobrem e se sobrepoem em toda a tabela.

Então a resposta unificada para a primeira parte da pergunta se resume no fato de que o tamanho da tabela é grande demais para os valores de chaves gerados. Por isso as chaves se concentram em regiões dispersas da tabela (o hash raramente precisa de fato realizar a operação de resto). E isso é corroborado pelo bom funcionamento da versão com produtos, conforme consta nas figuras 8, 9 e 10. Produtos de 8 elementos entre 65 e 122 facilmente ultrapassam 997 e a dinâmica da divisão por 997 se torna mais homogênea, e a mesma explicação vale para o Huckleberry Finn. Para os palavras do dataset `mod30.txt`, o formato padronizado que existia não ocorre para produtos entre os caracteres, apenas somas.

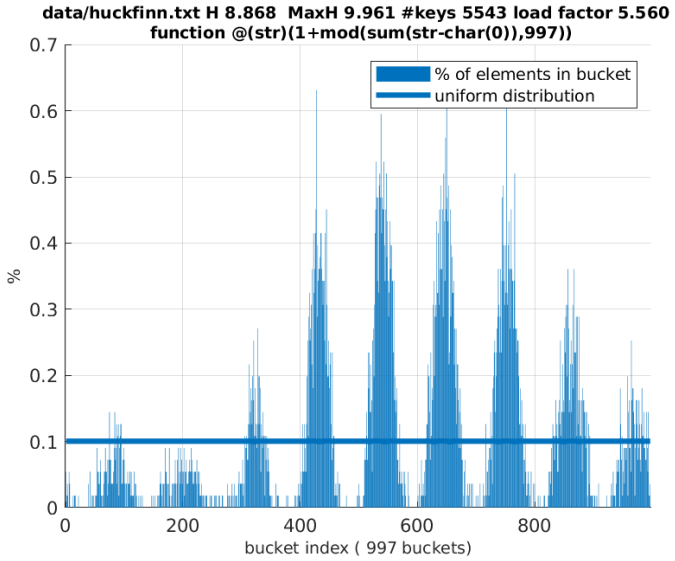


Figura 7: Distribuição do dataset `huckfinn.txt` nos buckets com a função de hash modsumchar1k.

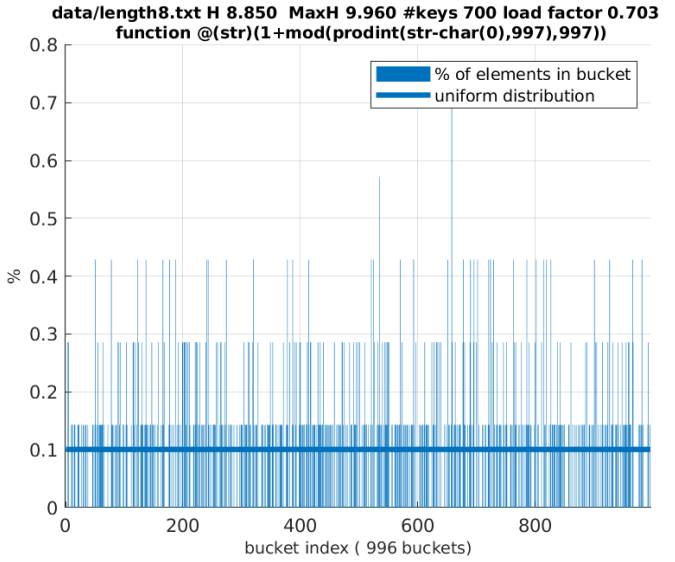


Figura 8: Distribuição do dataset `length8.txt` nos buckets com a função de hash modprodchar1k.

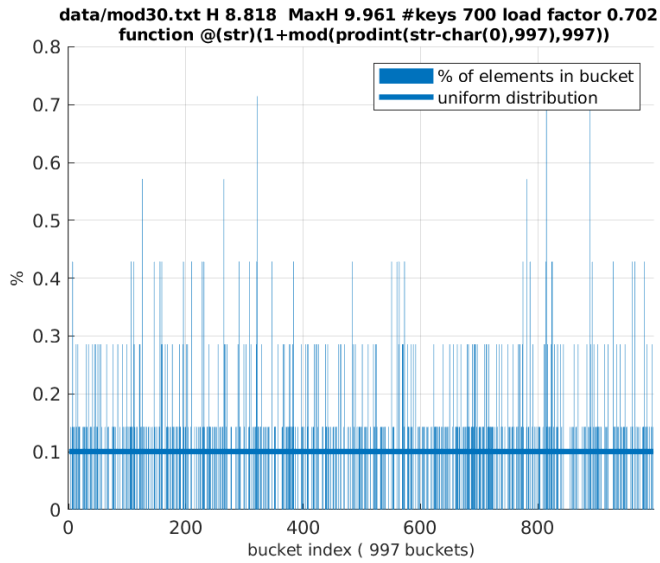


Figura 9: Distribuição do dataset `mod30.txt` nos buckets com a função de hash modsumprod1k.

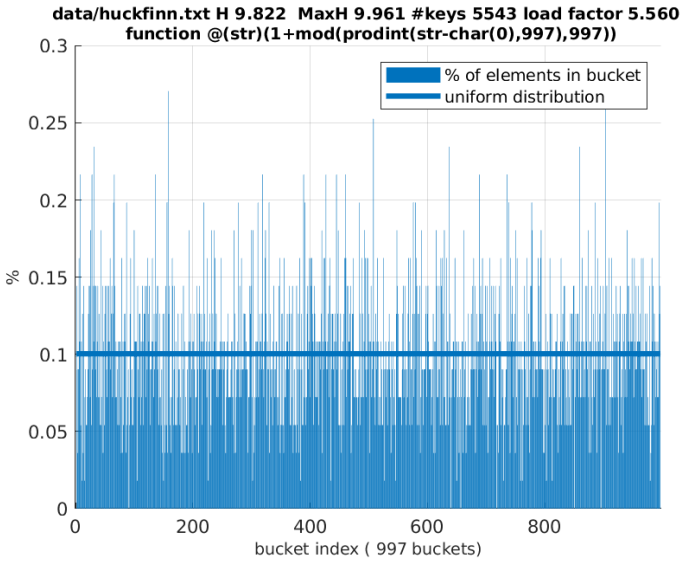


Figura 10: Distribuição do dataset `huckfinn.txt` nos buckets com a função de hash modsumchar1k.

4.

(4) Hash por divisão é o mais comum, mas outra alternativa é hash de multiplicação (NÃO É O MESMO QUE prodint.m). É uma alternativa viável? Por que hashing por divisão é mais comum?

Comparando o desempenho da função de hashing por divisão com a função de hashing por multiplicação (ambas de tamanho 29), não se nota diferenças razoáveis de desempenho. A depender do dataset uma função performa melhor do que a outra. Sendo assim, o método de multiplicação se mostra uma alternativa viável, com a vantagem de ser menos sensível à escolha do tamanho. No entanto, uma possível razão para justificar a preferência pelo método da divisão seja a sua simplicidade, e também uma maior velocidade no cálculo do hash de uma chave (por envolver menos operações).

5.

(5) Qual a vantagem de Closed Hash sobre Open Hash, e quando escolheríamos Closed Hash ao invés de Open Hash? (pesquise! É suficiente um dos pontos mais importantes)

No Closed Hashing, em vez de armazenarmos listas ligadas dinamicamente, os dados são armazenados diretamente num vetor de hashing. Dessa forma, evitamos o uso de ponteiros e economizamos um pouco de memória no processo, podendo criar uma hash table com mais slots usando a mesma quantidade de memória, contribuindo até para reduzir a quantidade de colisões. Por outro lado, ficamos com um espaço de armazenamento limitado, portanto só é adequado utilizar Closed Hashing quando temos uma quantidade máxima de elementos a serem armazenados definida. Em geral, o desempenho das operações de hashing piora muito com o aumento do fator de carga, portanto a situação ideal é quando temos um fator de carga limitado a alguma porcentagem.

Nesse esquema de hashing, o tratamento de colisões pode ser feito por *sondagem* (ou *probing*) da tabela, onde a cada inserção, sondamos a tabela em busca de um slot livre, através de uma função de sondagem, que irá gerar uma sequência de slots a serem examinados para cada chave, até encontrar o slot livre. A operação de busca é feita utilizando o mesmo esquema de sondagem. Nesse cenário, uma boa função de sondagem é importante para reduzir as colisões na inserção e na busca. O grande problema é a operação de deleção. Devido ao fato de se utilizar sondagem para as buscas e inserções, a posição de um elemento na tabela irá depender do histórico de operações de inserção/deleção. Isso faz com que a operação de busca possa se tornar arbitrariamente custosa, não sendo mais função do fator de carga. Dito isso, o Closed Hashing normalmente é evitado quando precisamos fazer deleção das chaves.

Em resumo, a situação que favorece o uso do Closed Hashing é quando temos uma quantidade máxima de chaves a serem armazenadas definida (algum valor que não seja alto demais) e quando não precisamos fazer a deleção das chaves armazenadas.

6.

(6) Suponha que um atacante conhece exatamente qual é a sua função de hash (o código é aberto e o atacante tem acesso total ao código), e pretende gerar dados especificamente para atacar o seu sistema (da mesma forma que o arquivo mod30 ataca a função de hash por divisão com tamanho 30). Como podemos implementar a nossa função de hash de forma a impedir este tipo de ataque? Pesquise e explique apenas a idéia básica em poucas linhas

A única forma de evitar um ataque é escolher nossa função de hashing aleatoriamente no início da execução, de forma que o atacante não tenha a informação necessária para produzir o seu dataset de ataque. Contudo, o conjunto de funções de hashing do qual iremos sortear nossa função deve ser *universal*, isto é, a chance de duas chaves serem mapeadas no mesmo slot pela função escolhida deve ser menor ou igual à chance de dessas chaves serem mapeadas no mesmo slot caso cada slot fosse escolhido aleatoriamente. Em termos mais rigorosos, se temos o conjunto de funções $\mathcal{H}$ e uma tabela de tamanho m, dadas duas chaves k_1 e k_2, devemos ter $|\{h \in \mathcal{H} : h(k_1) = h(k_2)\}| \leq \frac{|\mathcal{H}|}{m}$. Sob essas hipóteses, a estratégia de universal hashing garante boas propriedades estatísticas, isto é, bom desempenho no caso médio.