MC-202
Tabela de Espalhamento

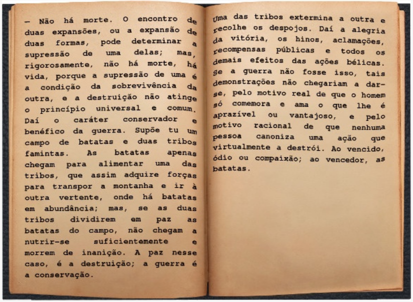
Marcelo S. Reis
msreis@ic.unicamp.br

Universidade Estadual de Campinas

Segundo semestre de 2023

Introdução

Queremos contar o número de ocorrências de cada palavra da biblioteca



— Não há morte. O encontro de duas expansões, ou a expansão de duas formas, pode determinar a supressão de uma delas; mas, rigorosamente, não há morte, há vida, porque a supressão de uma é a condição da sobrevivência da outra, e a destruição não atinge o princípio universal e comum. Daí o caráter conservador e benéfico da guerra. Supõe tu um campo de batatas e duas tribos famintas. As batatas apenas chegam para alimentar uma das tribos, que assim adquire forças para transpor a montanha e ir à outra vertente, onde há batatas em abundância; mas, se as duas tribos dividirem em paz as batatas do campo, não chegam a nutrir-se suficientemente e morrem de inanição. A paz nesse caso, é a destruição; a guerra é a conservação.

Uma das tribos extermina a outra e recolhe os despojos. Daí a alegria da vitória, os hinos, aclamações, recompensas públicas e todos os demais efeitos das ações bélicas. Se a guerra não fosse isso, tais demonstrações não chegariam a dar-se, pelo motivo real de que o homem só comemora e ama o que lhe é aprazível ou vantajoso, e pelo motivo racional de que nenhuma pessoa canoniza uma ação que virtualmente a destrói. Ao vencido, ódio ou compaixão; ao vencedor, as batatas.

- no idioma, há cerca de milhares de palavras (≈ 435.000)
- mas no total, há milhões de ocorrências!

Exemplo

dia:	6 ocorrências
escola:	13 ocorrências
gratuito:	1 ocorrência
ilha:	8 ocorrências
jeito:	5 ocorrências
lata:	2 ocorrências

Queremos acessar uma palavra como se fosse um vetor:

```
ocorrencias["ilha"] = 8
```

Primeiras opções:

- Vetor - acesso/escrita em $\mathrm{O}(n)$
  - inserir uma nova palavra leva $\mathrm{O}(1)$
- Vetor ordenado - acesso/escrita em $\mathrm{O}(\lg n)$
  - inserir uma nova palavra leva $\mathrm{O}(n)$
- ABB balanceada - acesso/escrita/inserção em $\mathrm{O}(\lg n)$

Conseguimos fazer em $\mathrm{O}(1)$?

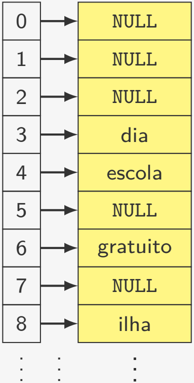
Caso fácil

Se tivéssemos apenas uma palavra começando com cada letra era fácil

- bastaria ter um vetor de 26 posições

a	0	
b	1	
c	2	
d	3	dia
e	4	escola
f	5	
g	6	gratuito
h	7	
i	8	ilha
⋮	⋮	⋮

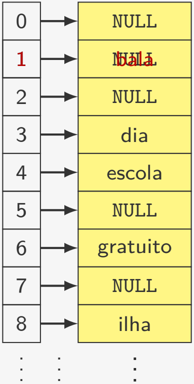
Palavras que começam com a mesma letra



Ideia:

- uma lista ligada para cada letra
- guardamos os ponteiros para as listas em um vetor

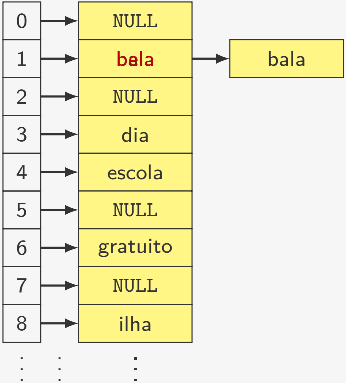
Palavras que começam com a mesma letra



Inserindo “bala”:

- descobrimos a posição pela primeira letra
- atualizamos o vetor para apontar para o nó de “bala”

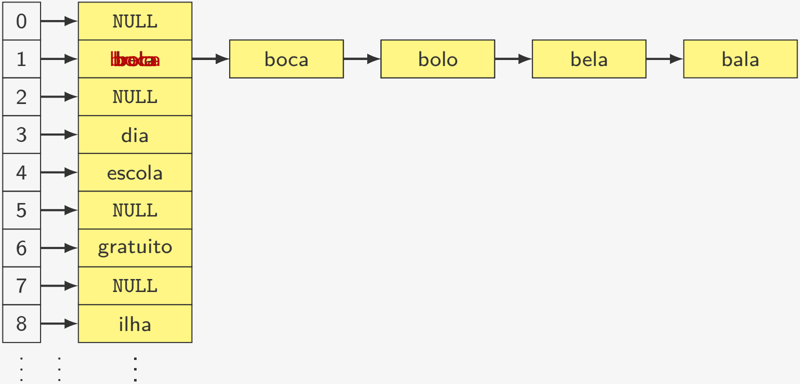
Palavras que começam com a mesma letra



Inserindo “bela”:

- descobrimos a posição pela primeira letra
- temos uma colisão com “bala”
- inserimos no começo da lista da letra `b`

Palavras que começam com a mesma letra



Após a inserção de várias palavras começando com b:

- inserimos "bolo", "boca", "broca"
- a tabela ficou degenerada em lista

Espalhamento com Encadeamento Separado

broca ⇝ `h("broca") = 3`

boca ⇝ `h("boca") = 0`

bolo ⇝ `h("bolo") = 5`

bela ⇝ `h("bela") = 2`

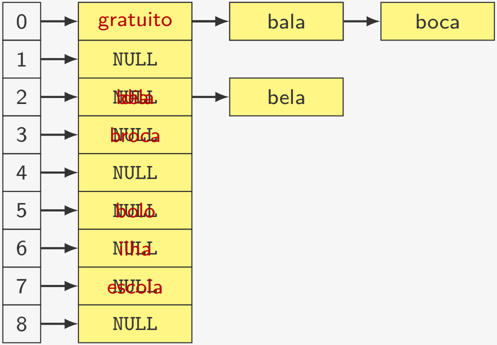
bala ⇝ `h("bala") = 0`

dia ⇝ `h("dia") = 2`

escola ⇝ `h("escola") = 7`

gratuito ⇝ `h("gratuito") = 0`

ilha ⇝ `h("ilha") = 6`



Corrigindo:

- vamos tentar espalhar melhor
- usamos um hash da chave (palavra)
- vamos associar a chave a um número inteiro (entre 0 e 8)

Tabela de Espalhamento

Uma função de hashing associa

- um elemento de um conjunto (strings, números, etc.)
- a um número inteiro de tamanho conhecido

Uma tabela de espalhamento é um TAD para conjuntos dinâmicos com certas propriedades:

- os dados são acessado por meio de um vetor de tamanho conhecido
- a posição do vetor é calculada por uma função de hashing

Características das tabelas de espalhamento

Restrições

- estimativa do tamanho do conjunto de dados deve ser conhecida
- bits da chave devem estar disponíveis
  - em uma ABB, basta uma função de comparação

Tempo das operações

- depende principalmente da função de hashing escolhida
- chaves bem espalhadas: tempo “quase” $\mathrm{O}(1)$
  - se temos n itens
  - uma tabela de tamanho M
  - tempo de acesso é o tempo de calcular a função de hashing mais $\mathrm{O}(n/M)$
- chaves muito agrupadas: pior caso de tempo $\mathrm{O}(n)$
  - vira uma lista ligada com todos os elementos

Obtendo funções de hashing

Uma boa função de hashing deve espalhar bem:

- A probabilidade de uma chave ter um hash específico é (aproximadamente) $1/M$
- Ou seja, esperamos que cada lista tenha n/M elementos

Métodos genéricos (que funcionam bem na prática):

1. Método da divisão
2. Método da multiplicação

Hashing perfeito: Se conhecermos todos as chaves a priori, é possível encontrar uma função de hashing injetora

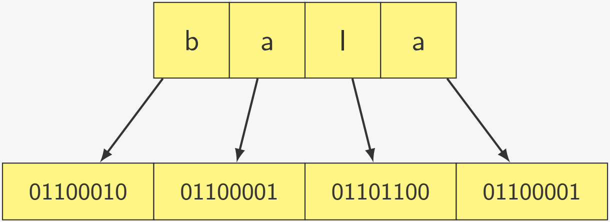
- isto é, não temos colisões
- tais funções podem ser difíceis de encontrar

Interpretando chaves

Pressupomos que as chaves são números inteiros

E se não forem?

- Reinterpretamos a chave como uma sequência de bits



Assim, "bala" se torna o número 1.650.551.905

- Esse número pode explodir rapidamente
- Veremos como contornar isso para strings...

Método da divisão

- obtemos o resto da divisão pelo tamanho M do hashing

$$h(x) = x \textbf{ mod } M$$

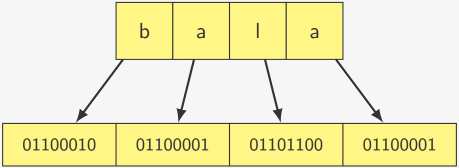
Exemplo:

$$h(\text{“bala”}) = 1.650.551.905 \textbf{ mod } 1783 = 277$$

Escolhendo M:

- escolher M como uma potência de 2 não é uma boa ideia
  - considera apenas os bits menos significativos
- normalmente escolhemos M como um número primo longe de uma potência de 2

Método da divisão para Strings



Como podemos calcular o número x que representa "bala"?

- $x = \texttt{'b'} \cdot 256^3 + \texttt{'a'} \cdot 256^2 + \texttt{'l'} \cdot 256^1 + \texttt{'a'} \cdot 256^0$

que pode ser rescrito como

- $x = (((\texttt{'b'}) \cdot 256 + \texttt{'a'}) \cdot 256 + \texttt{'l'}) \cdot 256 + \texttt{'a'}$

Mas x poderia ser muito grande e estourar um `int`...

Ao invés de calcular $x \bmod M$, calculamos

$(((\texttt{'b'} \bmod M) \cdot 256 + \texttt{'a'} \bmod M) \cdot 256 + \texttt{'l'} \bmod M) \cdot 256 + \texttt{'a'} \bmod M$

Método da multiplicação

- multiplicamos por um certo valor real A e obtemos a parte fracionária
- escolhemos A conveniente, por exemplo $A = (\sqrt{5} - 1)/2$
- posição relativa no vetor não depende de M (pode ser $M = 1024$)

$$h(x) = \lfloor M\,(A \cdot x \bmod 1) \rfloor$$

Exemplo:

$$\begin{aligned} h(\text{“bala”}) &= \lfloor 1024 \cdot [((\sqrt{5} - 1)/2 \cdot 1.650.551.905) \bmod 1] \rfloor \\ &= \lfloor 1024 \cdot [1020097177{,}4858876 \bmod 1] \rfloor \\ &= \lfloor 1024 \cdot 0{,}4858876 \rfloor \\ &= \lfloor 497{,}5489024 \rfloor = 497 \end{aligned}$$

O uso da razão áurea como valor de A é sugestão de Knuth

Interface do TAD

```c
#define MAX 1783

typedef struct no *p_no;

struct no {
  char chave[10];
  int dado;
  p_no prox;
};

typedef struct hash *p_hash;

struct hash {
  p_no vetor[MAX];
};

p_hash criar_hash();

void destruir_hash(p_hash t);

void inserir(p_hash t, char *chave, int dado);

void remover(p_hash t, char *chave);

p_no buscar(p_hash t, char *chave);
```

Exemplo de implementação

```
int hash(char *chave) {
  int i, n = 0;
  for (i = 0; i < strlen(chave); i++)
    n = (256 * n + chave[i]) % MAX;
  return n;
}

void inserir(p_hash t, char *chave, int dado) {
  int n = hash(chave);
  t->vetor[n] = inserir_lista(t->vetor[n], chave, dado);
}

void remover(p_hash t, char *chave) {
  int n = hash(chave);
  t->vetor[n] = remover_lista(t->vetor[n], chave);
}
```

Quebrando um programa que usa hashing

Sabendo a função de hashing, podemos prejudicar o programa:

- insira muitos elementos com o mesmo hash

Como nos proteger de um adversário malicioso?

- Podemos escolher a função de hashing aleatoriamente

Uma boa função de hashing aleatória:

- fixe p um primo maior do que M
- escolha $a \in \{1, \ldots, p\}$ e $b \in \{0, \ldots, p\}$ uniform. ao acaso
- defina $h_{a,b}(k) = ((ak + b) \bmod p) \bmod M$
- sabemos que essa função espalha bem
  - a probabilidade de colisão é no máximo $1/M$
  - é um hashing universal

Endereçamento aberto

Existe uma alternativa para a implementação de tabela de espalhamento

Endereçamento aberto:

- os dados são guardados no próprio vetor
- colisões são colocadas em posições livres da tabela

Características:

- evita percorrer usando ponteiros e alocação e deslocação de memória (`malloc` e `free`)
- se a tabela encher, deve recriar uma tabela maior
  - e mudar a função de hashing
- remoção é mais complicada

Endereçamento aberto com sondagem linear

broca $\rightsquigarrow$ `h("broca") = 3`

boca $\rightsquigarrow$ `h("boca") = 0`

bolo $\rightsquigarrow$ `h("bolo") = 5`

bela $\rightsquigarrow$ `h("bela") = 2`

bala $\rightsquigarrow$ `h("bala") = 0`

dia $\rightsquigarrow$ `h("dia") = 2`

escola $\rightsquigarrow$ `h("escola") = 7`

gratuito $\rightsquigarrow$ `h("gratuito") = 0`

ilha $\rightsquigarrow$ `h("ilha") = 6`

0	boca
1	bala
2	bela
3	broca
4	dia
5	bolo
6	gratuito
7	escola
8	ilha

Inserindo:

- procuramos posição
- se houver espaço, guardamos
- se não houver espaço, procuramos a próxima posição livre (módulo M)

Busca em endereçamento aberto

Como fazer uma busca com endereçamento aberto?

- Basta simular a inserção:
  - Calcule a função de hashing
  - Percorra a tabela em sequência procurando pela chave
  - Se encontrar a chave, devolva o item correspondente
  - Se encontrar um espaço vazio, devolva `NULL`

O que é um espaço vazio em um vetor?

- Se for um vetor de ponteiros, pode ser `NULL`
- Se não for, precisa ser um elemento dummy
  - ou um valor que nunca será usado
  - ou ter um campo indicando que é dummy

Remoção em endereçamento aberto

Como fazer a remoção com endereçamento aberto?

- Não podemos apenas remover os elementos da tabela
  - Por quê? Quebraria a busca...
- Opção 1: rehash dos elementos seguintes do bloco
  - removemos os elementos até a próxima posição vazia
  - recalculamos o hash de cada um e reinserimos na tabela
  - pode ser custoso e difícil de implementar
- Opção 2: trocamos por um valor dummy
  - valor indica que o item foi removido
  - mas não pode ser o mesmo que indica espaço vazio
- Opção 3: marcamos o item como removido
  - usamos um campo adicional

Inserção e Busca Revisitadas

Se fizermos a remoção marcando o item como removido, precisamos mudar a inserção e a busca

Inserção:

- Calculamos a função hashing e temos um resultado h
- Inserimos na primeira posição vazia ou com item removido a partir de h

Busca:

- Calculamos a função hashing e temos um resultado h
- Procuramos cada posição a partir de h em sequência
  - Se encontrar o item, verifique se ele não foi removido
  - Passe por cima de itens removidos
  - Pare ao encontrar uma posição vazia
- Cuidado para não ciclar...

Hashing duplo

É como a sondagem linear:

- Estratégia mais geral para lidar com conflitos
- Ao invés de saltarmos sempre de 1, saltamos de $hash_2(k)$
- Onde $hash_2$ é uma segunda função de hashing

Isso é,

$$h(k, i) = (hash_1(k) + i \cdot hash_2(k)) \mod M$$

Cuidados:

- $hash_2(k)$ nunca pode ser zero
- $hash_2(k)$ precisa ser coprimo com M
  - garante que as sequências são longas

Exemplos:

- Escolha M como uma potência de 2 e faça que $hash_2(k)$ seja sempre ímpar
- Escolha M como um número primo e faça que $hash_2(k) < M$

Sondagem linear e Hashing duplo[1]

Sondagem linear - número de acessos médio por busca

n/M	$1/2$	$2/3$	$3/4$	$9/10$
com sucesso	1.5	2.0	3.0	5.5
sem sucesso	2.5	5.0	8.5	55.5

Hashing duplo - número de acessos médio por busca

n/M	$1/2$	$2/3$	$3/4$	$9/10$
com sucesso	1.4	1.6	1.8	2.6
sem sucesso	1.5	2.0	3.0	5.5

De qualquer forma, é muito importante não deixar a tabela encher muito:

- Você pode aumentar o tamanho da tabela dinamicamente
- Porém, precisa fazer um rehash de cada elemento para a nova tabela

[1] Baseado em Sedgewick, R. Algorithms in C, third edition, Addison-Wesley. 1998.

Conclusão

Hashing é uma boa estrutura de dados para

- inserir, remover e buscar dados pela sua chave rapidamente
- com uma boa função de hashing, essas operações levam tempo $O(1)$
- mas não é boa se quisermos fazer operação relacionadas a ordem das chaves

Escolhendo a implementação:

- Sondagem linear é o mais rápido se a tabela for esparsa
- Hashing duplo usa melhor a memória
  - mas gasta mais tempo para computar a segunda função de hash
- Encadeamento separado é mais fácil de implementar
  - Usa memória a mais para os ponteiros

Conclusão

Além disso, funções de hashing têm várias outras aplicações, ex:

- Para evitar erros de transmissão, podemos, além de informar uma chave, transmitir o resultado da função de hashing.
  - dígitos verificadores
  - sequências de verificação para arquivos (MD5 e SHA)
- Guardamos o hash de uma senha no banco de dados ao invés da senha em si
  - evitamos vazamento de informação em caso de ataque
  - mas temos que garantir que a probabilidade de duas senhas terem o mesmo hash seja ínfima...