AED2 - Aulas 02 e 03
Implementação de hash tables

Queremos implementar uma tabela de símbolos para:

- armazenar itens que possuem chave e valor.
- as chaves estão distribuídas num universo U bastante grande,
- mas o conjunto de itens S é bem menor.

Neste cenário, duas possibilidades de implementação simples seriam:

- um vetor diretamente indexado pelas chaves
    - vantagem: tempo de acesso constante (\Theta(1)).
    - desvantagem: tamanho proporcional a |U| (\Theta(|U|)).
- uma lista encadeada
    - vantagem: tamanho proporcional a |S| (\Theta(|S|)).
    - desvantagem: tempo de acesso proporcional a |S| (\Theta(|S|)).

Tabelas de espalhamento (hash tables)

Implementação bastante popular e eficiente para tabelas de símbolos, isso porque Hash tables propriamente implementadas:

- possuem tamanho proporcional a |S| (\Theta(S)),
- suportam operações de consulta, inserção e remoção muito eficientes, i.e., tempo constante (\Theta(1)) por operação.
    - a eficiência das operações depende da hash table ter tamanho adequado e uma boa função de espalhamento (hash function)
        - vale destacar que é fácil implementar uma função de espalhamento problemática.
    - além disso, a hash table não dá garantia de eficiência de pior caso,
        - pois sempre podem existir conjuntos de dados patológicos.

Implementação de hash tables:

- usar um vetor de tamanho M, com M proporcional a |S|.
- usar uma função de espalhamento (hash function) h: U -> {0, …, M-1}
    - mínimo necessário: h deve converter cada chave para um índice do vetor, i.e.,
        - h(chave) = chave % M

```
int hash(Chave chave, int M)
{
   return chave % M;
}
```

- h(chave) = (a * chave + b) % M

```
int hash(Chave chave, int M)
```

```
{
   return (17 * chave + 43) % M;
}
```

- objetivo desejado: h deve ser rápida de calcular, ocupar pouco espaço e espalhar as chaves uniformemente pela extensão do vetor.
  - note que usar uma função aleatória não é viável. Por que?
  - supondo que a chave é uma string, a seguinte função é um exemplo interessante

```
int hash(Chave chave, int M)
{
   int i, h = 0;
   int primo = 127;
   for (i = 0; chave[i] != '\0'; i++)
       h = (h * primo + chave[i]) % M;
   return h;
}
```

- Funções de espalhamento de referência:
  - FarmHash, MurmurHash3, SpookyHash, MD5
- Testar o desempenho de diferentes funções de espalhamento (suas ou da literatura) com os dados do seu problema é essencial para realizar uma escolha embasada.

* colisões são inevitáveis:
  - uma colisão ocorre quando h mapeia duas chaves diferentes para a mesma posição do vetor.
  - não apenas colisões são inevitáveis como são comuns.
    - considere o "paradoxo" do aniversário para obter uma intuição
      - considere n pessoas e um ano com 365 dias
      - a chance de um par qualquer de pessoas aniversariar no mesmo dia é 1/365
      - mas temos (n choose 2) = n(n-1)/2 ~= n^2/2 pares
      - assim, a probabilidade de um par ocorrer ~= (1/365)*(n^2/2).

$$Pr(1 \text{ par coincidir}) \simeq (1/365)(n^2/2)$$

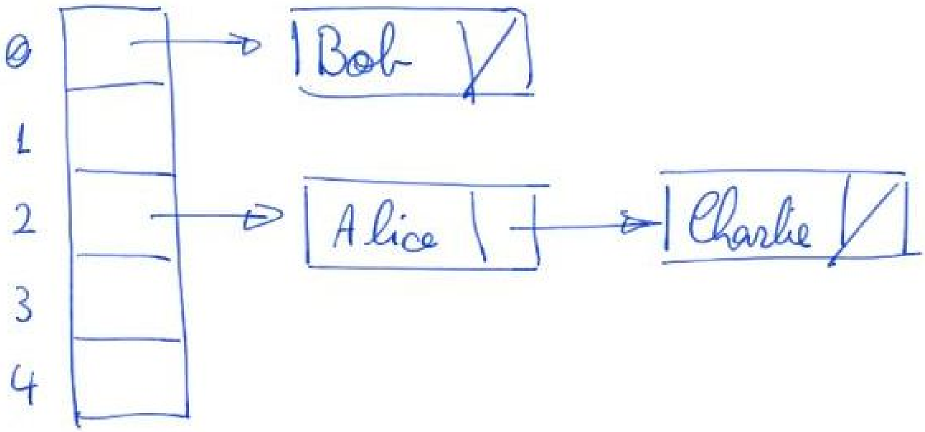
$$\text{Suponha } Pr(1 \text{ par coincidir}) = 1/2$$

$$\frac{1}{2} = \frac{1}{365} \cdot \frac{n^2}{2} \Rightarrow n^2 = \frac{2 \cdot 365}{2}$$

$$\Rightarrow n = \sqrt{365} \simeq 19$$

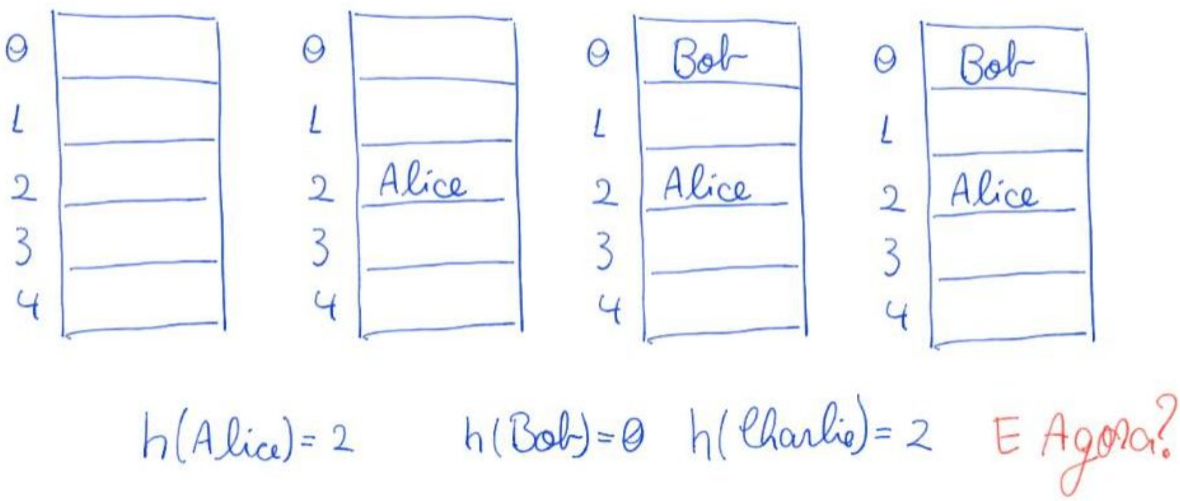
- onde está o erro da expressão acima?
- Generalizando a fórmula anterior, trocamos o número de dias no ano por M. Assim,
  - a probabilidade de uma colisão é 1/2 quando
    - n ~= raiz(M)
  - e devemos encontrar colisões quando
    - n ~= 2 raiz(M)
- Note que para M grande, digamos 10^6, devemos encontrar as primeiras colisões quando
  - apenas 2 mil elementos forem inseridos na tabela
  - ou seja, apenas 0,2% da tabela estiver ocupada.

- alternativas para tratar colisões:
  - listas encadeadas.

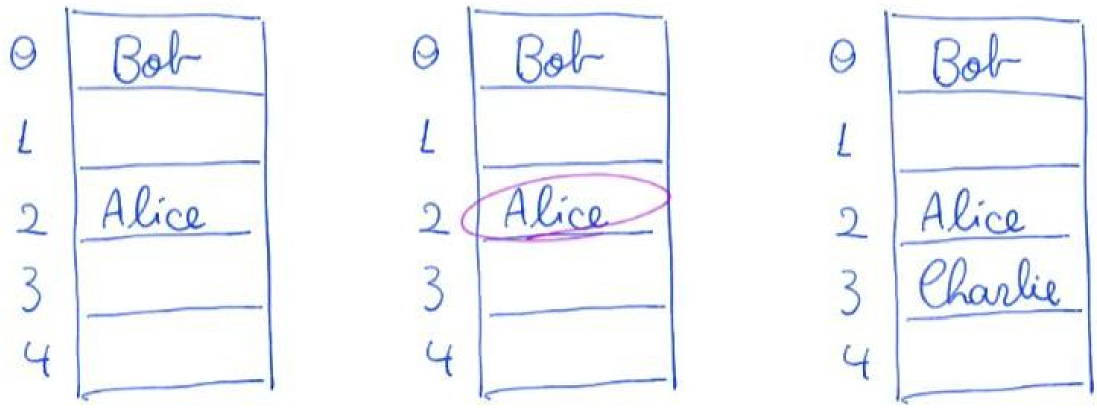


- inserção leva tempo constante, mas consulta e remoção dependem da qualidade da função de espalhamento e do tamanho da hash table.
- prós: remoção é simples de implementar.
- contra: ocupa mais espaço.

  - endereçamento aberto

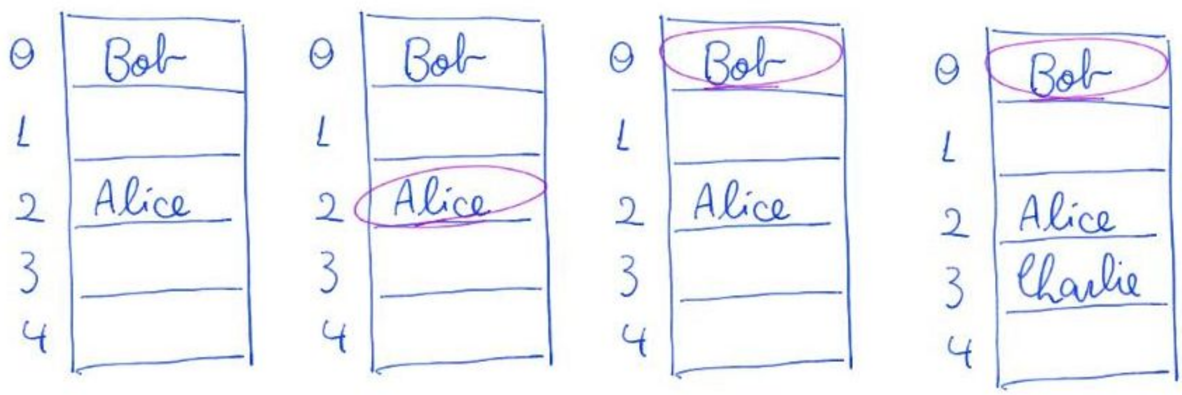


- sondagem (probing)
    - linear: offset segue uma função linear (i) a partir da posição inicial.



h(Charlie) = 2 Como a posição está ocupada, tenta a posição +1 até encontrar uma posição vazia.

        - contra: costuma gerar aglomerações
    - quadrática: offset segue uma função quadrática (i^2) a partir da posição inicial.
    - em ambos os casos i é o número da tentativa de re-endereçamento após a primeira.
- re-espalhamento (rehashing ou double hashing).



$h(\text{Charlie}) = 2$ usando outra função de espalhamento $g()$ calcula $g(\text{Charlie}) = 3$ e usa isso como deslocamento a partir da posição $h(\text{Charlie})$.

Como $(2 + 1.3) \% 5 = 0$ atinge outra posição ocupada adicionamos novamente o deslocamento, i.e., $(2 + 2.3) \% 5 = 3$.

        - prós: endereçamento aberto ocupa menos espaço.
        - contra: remoção é mais complicada de implementar.

- carga de uma hash table
    - carga = |S| / M
    - qual estratégia para tratamento de colisões permite cargas maiores que 1?
        - observe que o tempo de acesso esperado numa hash table com listas encadeadas é da ordem de 1 + carga.
        - no caso de endereçamento aberto bem implementado esse tempo cresce de acordo com a função 1/(1 - carga).
            - esse resultado deriva do número esperado de moedas que precisamos jogar até obter o primeiro sucesso.
            - isso significa tempo constante para carga <= 70%,
            - e crescimento veloz quando carga se aproxima de 100%.
    - como hash tables são estruturas dinâmicas pode ser necessário redimensioná-la de tempos em tempos.
        - uma regra prática é não deixar a carga passar de 70%.
        - quando isso acontecer tome um vetor de tamanho 2M
        - e re-espalhe os itens nesse novo vetor usando uma versão modificada da sua função de hash, i.e.,
            - com % 2M no final