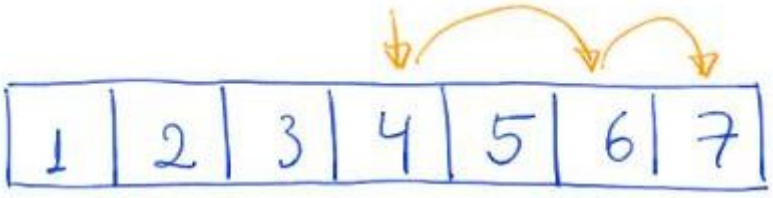
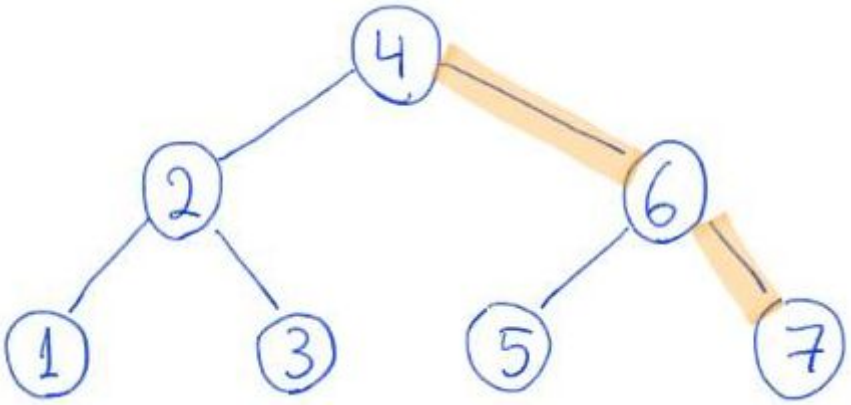
AED2 - Aulas 07 e 08
Hash tables

Vamos ver o que as estruturas de dados para busca
- que estudamos até agora têm em comum.

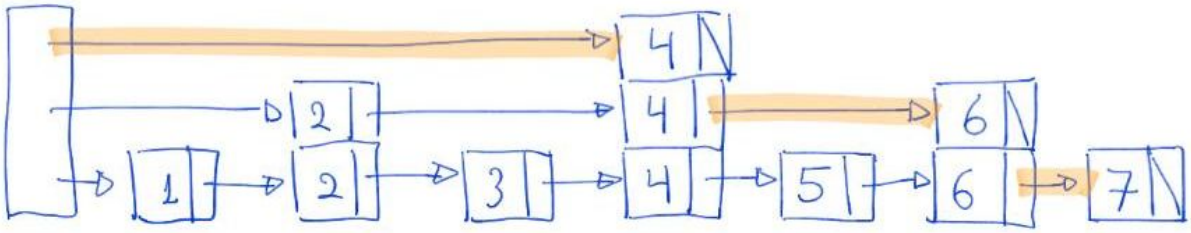
Para tanto, considere a busca pelo elemento 7 em
- vetor ordenado



- Árvore binária de busca balanceada



- Skip lists



Note que, todas são baseadas em dividir sucessivamente o espaço de busca.
- Com isso, acabam com tempo de busca proporcional a log n,
    - sendo n o número de itens no espaço de busca.

Será que conseguimos fazer melhor utilizando uma abordagem diferente?
- Uma ideia é utilizar um vetor diretamente indexado pelas chaves.
- Uma vantagem dessa abordagem é que o tempo de acesso
    - é constante em relação ao número de elementos, i.e., O(1).
- Se as chaves estiverem num intervalo pequeno,

    - por exemplo, inteiros entre 1 e 1000, isso é viável,
        - pois só precisamos alocar um vetor com mil posições.
- Infelizmente (ou não), esse raramente é o caso.

Como exemplo, considere que queremos criar uma lista de telefones.

- Neste caso, as chaves são os nomes das pessoas,
    - os quais, em geral, não correspondem a inteiros entre 1 e mil.
        - De fato, nomes em geral não correspondem a inteiros.
- Insistindo na ideia, podemos converter nomes em números inteiros,
    - por exemplo, considerando a representação binária de cada nome.
        - e usar esse número para indexar o vetor.
- Qual o problema dessa abordagem?
    - Qual será o tamanho do vetor resultante?
        - No caso dos nomes, teríamos uma posição
            - para cada nome possível.
        - Considerando que cada caracter tem 26 possibilidades
            - e um nome pode ter até 30 caracteres, o vetor teria
                - 26^30 ~= 2,81 * 10^42
                    - ~= 2,81 * (10^3)^14
                    - ~= 2,81 * (2^10)^14
                    - ~= 2^141 posições.
        - Como comparação, o armazenamento total disponível na Terra
            - é da ordem de 10^25 bits.
    - Em geral, o tamanho seria da ordem de 2^n,
        - sendo n o número de bits da chave.

O tamanho do vetor no exemplo anterior deixa claro que essa abordagem é inviável.

- mas existe uma estrutura de dados que
    - suporta busca, inserção e remoção em tempo constante
        - i.e., O(1),
    - e ocupa espaço proporcional ao número de elementos armazenados.

Tabelas de espalhamento (hash tables)

Trata-se de uma implementação bastante popular e eficiente

- para tabelas de símbolos.

Hash tables **propriamente implementadas** suportam operações de

- consulta, inserção e remoção muito eficientes,
    - i.e., tempo constante (O(1)) por operação.

A eficiência das operações depende da hash table ter

- tamanho adequado,
- bom tratamento de colisões e
- uma boa função de espalhamento (hash function).

Vamos detalhar cada um desses tópicos.

Sobre funções de espalhamento, vale destacar que

- elas também são úteis em outros contextos, como
    - verificação de integridade de dados transmitidos e
    - validação de identificadores, como RGs e CPFs.
- Além disso, é fácil implementar uma função de espalhamento problemática,
    - ou seja, que não espalha bem os dados.

Também é importante destacar limitações das hash tables, como

- não ter boa garantia de eficiência de pior caso,
    - pois sempre existem conjuntos de dados patológicos,
        - i.e., cujas chaves são todas mapeadas para a mesma posição,
    - por melhor que seja sua função de espalhamento.
        - Isso porque, estamos mapeando um conjunto grande U
            - para apenas M valores.
- Além disso, ao contrário de outras implementações para tabelas de símbolos,
    - nas hash tables os dados não ficam ordenados.
        - Por isso, operações como
            - mínimo, máximo, sucessor, antecessor, rank e seleção
                - não são eficientes.

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.

Usar um vetor de tamanho M,

- com M proporcional a |S|.

Usar uma função de espalhamento (hash function)

- h: U -> {0, …, M-1}.

Implementação de hash functions

Mínimo necessário: h deve converter cada chave para um índice do vetor, i.e.,

- h(chave) = chave % M

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

        - Nesta função, chaves próximas
            - tendem a cair próximas ou na mesma posição.
    - h(chave) = (a * chave + b) % M

```c
int hash(Chave chave, int M)
{
   return (17 * chave + 43) % M;
}
```

        - Nesta função chaves próximas são mais espalhadas,
            - mas por um fator constante.
    - Além disso, em ambas os dígitos menos significativos
        - podem ser os únicos relevantes,
            - dependendo do valor de M.
        - Por exemplo, considere M = 10 ou 100.

Objetivo desejado: h deve ser

- determinística,
- rápida de calcular,
- ocupar pouco espaço e
- espalhar as chaves uniformemente pela extensão do vetor,
    - pois, idealmente, cada item deveria receber uma posição exclusiva,
        - como no vetor indexado por chaves que nos inspirou.

Note que, uma função uniforme aleatória tem várias características que desejamos.

- Por que?
- Apesar disso, ela não pode ser usada.
    - Por que?

Supondo que a chave é uma string, a seguinte função

- faz com que todo caracter tenha influência no resultado.

```c
int hash(Chave chave, int M)
{
   int i, h = 0;
   for (i = 0; chave[i] != '\0'; i++)
       h += chave[i];
   h = h % M;
```

```
    return h;
}
```

- No entanto, chaves próximas tendem a cair próximas
    - e caracteres com valores múltiplos de M são irrelevantes.

A seguinte função tenta melhorar esses aspectos

- multiplicando cada caractere por um número primo.

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

- Vale notar que, basta que o número seja primo em relação a M,
    - i.e., não tenha divisores comuns com M.
        - Extra: lembrem que já conhecemos um algoritmo eficiente
            - para determinar se dois números têm divisores comuns.
- Observe que, chaves que são anagramas, ou seja,
    - compostas pelos mesmos caracteres em diferentes ordens,
        - são mapeadas para a mesma posição.
- Além disso, chaves cujos caracteres somam o mesmo valor
    - continuam caindo na mesma posição.

A seguinte função evita esses problemas

- usando uma ideia inspirada na notação posicional.

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

- Vale destacar a importância do número multiplicado

        - ser primo com relação a M, pois
            - caso contrário as posições múltiplas de mdc(M, primo)
                - serão privilegiados ou até exclusivas.
- Também é importante que o primo tenha valor próximo de M,
    - caso contrário em pequenos intervalos de chaves,
        - as menores serão mapeadas consistentemente
            - para posições menores.

Vale notar que, podemos implementar uma função semelhante à anterior,

- sem usar a operação de potência. Isso porque,
    - c1*p + c2*p^2 + c3*p^3 + … = p (c1 + p (c2 + p (c3 + …)))

```
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]);
   h = h % M;
   return h;
}
```

- Note que, esta função tem a mesma ideia da notação posicional,
    - mas os índices menores são multiplicados pelas maiores potências.
- Uma preocupação é que, em todas as nossas funções,
    - o valor de h pode crescer tanto
        - a ponto de ocorrer erro numérico de estouro de variável.

Para evitar esse tipo de erro, podemos usar

- a seguinte propriedade do resto
    - (a + b) % M = (a % M + b % M) % M.
- Assim, chegamos à seguinte função de espalhamento

```
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;
}
```

Eficiência de tempo:

- Nossas funções tem eficiência proporcional ao número de dígitos da chave.
    - Por isso, podem não ser eficientes com chaves muito grandes.
- Mas, com relação ao número de itens na nossa tabela,
    - o tempo é constante, i.e., O(1).

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 boa escolha.

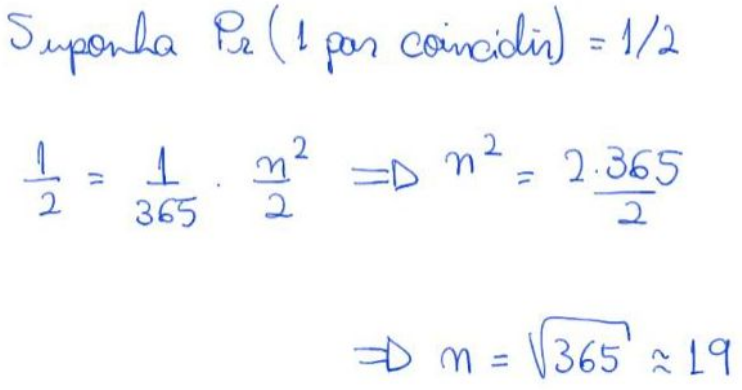
Colisões são inevitáveis

Uma colisão ocorre quando h mapeia duas chaves diferentes

- para a mesma posição do vetor.

Colisões não são apenas inevitáveis, mas são comuns.

- Considere o “paradoxo” do aniversário para obter uma intuição.
- Num grupo com 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).

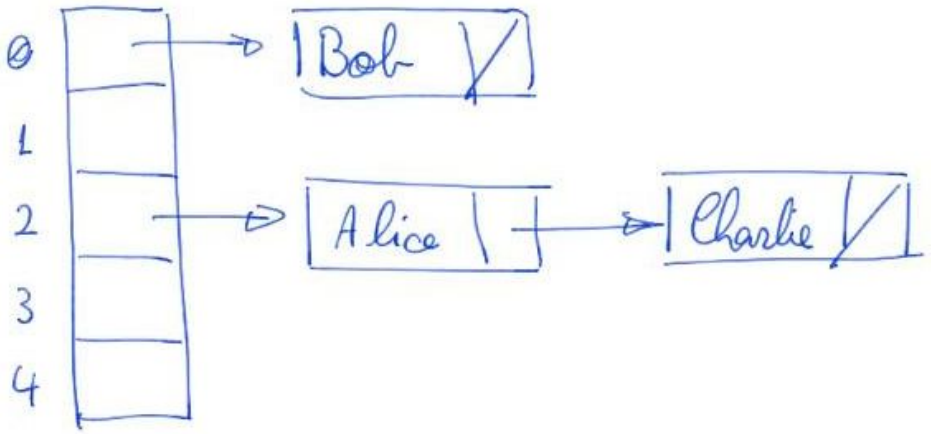


        - Onde está o erro da expressão acima?
- Generalizando a fórmula anterior, trocamos
    - o número de dias no ano por M.
- Assim, a probabilidade de uma colisão é 1/2 quando n ~= raiz(M)
    - e é muito provável encontrar colisões quando n ~= 2 raiz(M).
- Note que, para M grande, digamos 10^6,
    - devemos encontrar as primeiras colisões quando
        - 2 M^½ = 2 (10^6)^½ = 2 (10^3) = 2 mil elementos

        - forem inseridos na tabela.
    - Ou seja, quando 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.
    - Exemplo de código com listas ligadas.

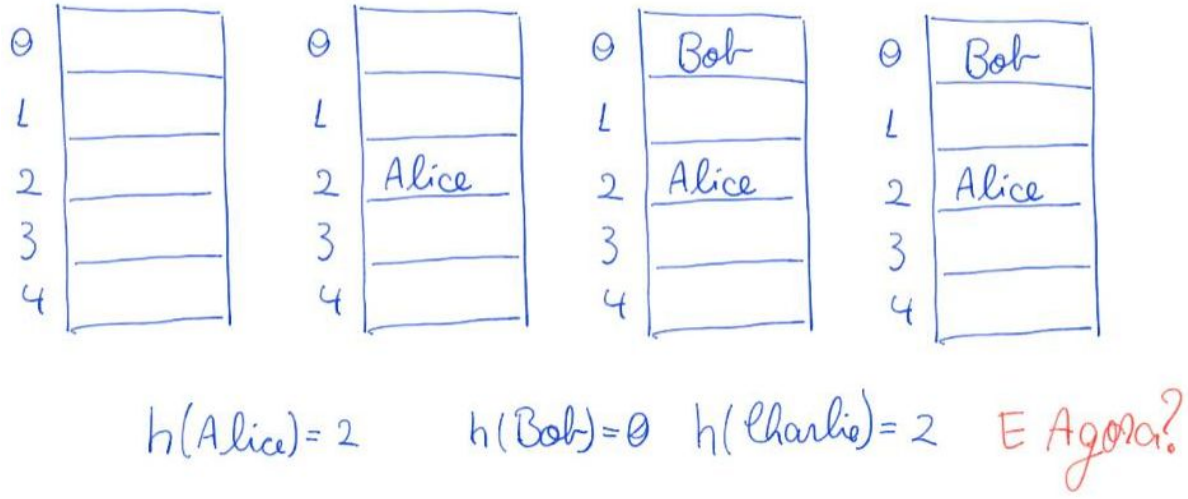
```
typedef struct celTS CelTS;
struct celTS
{
   Chave chave;
   Valor valor;
   CelTS *prox;
};

static CelTS **tab = NULL;
static int nChaves = 0;
static int M; // tamanho da tabela

void stInit(int max)
{
   int h;
   M = max;
   nChaves = 0;
```

```
    tab = mallocSafe(M * sizeof(CelTS *));
    for (h = 0; h < M; h++)
        tab[h] = NULL;
}

Valor stSearch(Chave chave)
{
    CelTS *p;
    int h = hash(chave, M);
    p = tab[h];
    while (p != NULL && strcmp(p->chave, chave) != 0)
        p = p->prox;
    if (p != NULL) // se encontrou devolve o valor
        return p->valor;
    return 0; // caso contrário devolve 0. E se o valor for 0? Como
contornar esse problema?
}

void stInsert(Chave chave, Valor valor) // inserção ou edição
{
    CelTS *p;
    int h = hash(chave, M);
    p = tab[h];
    while (p != NULL && strcmp(p->chave, chave))
        p = p->prox;
    if (p == NULL) // se não encontrou insere nova célula na lista
    {
        p = mallocSafe(sizeof(*p));
        p->chave = copyString(chave);
        nChaves += 1;
        p->prox = tab[h];
        tab[h] = p;
    }
    p->valor = valor; // atualiza valor do item
}
```

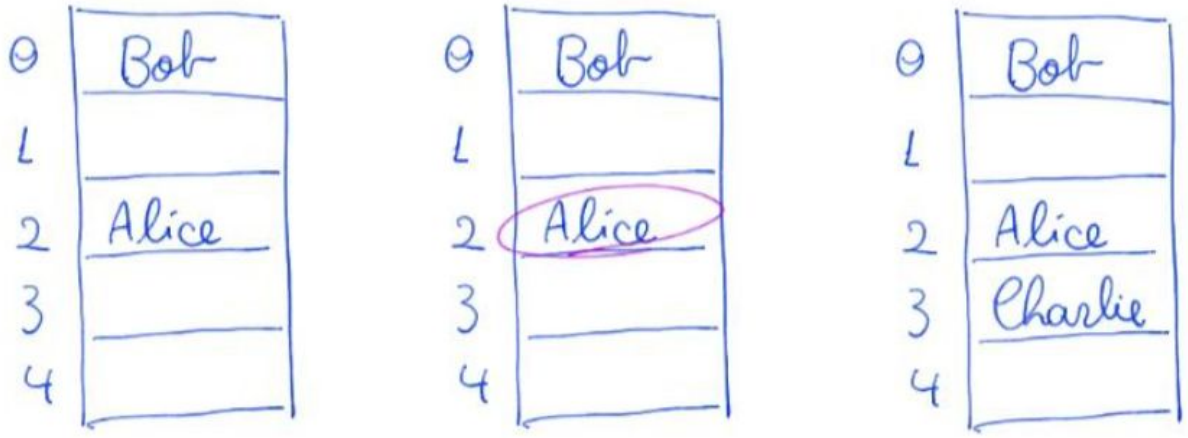
```
void stDelete(Chave chave)
{
   CelTS *p, *aux;
   int h = hash(chave, M);
   p = tab[h];
   if (p == NULL) // se lista está vazia não tem o que remover
       return;
   if (strcmp(p->chave, chave) == 0) // remoção na cabeça da lista
   {
       tab[h] = p->prox;
       free(p->chave);
       free(p);
       nChaves--;
       return;
   }
   // remoção no restante da lista
   while (p->prox != NULL && strcmp((p->prox)->chave, chave) != 0)
       p = p->prox;
   if (p->prox != NULL) // caso em que o próximo é o valor por
remover
   {
       aux = p->prox;
       p->prox = aux->prox;
       free(aux->chave);
       free(aux);
       nChaves--;
   }
}

void stFree()
{
   CelTS *p = NULL, *q = NULL;
   int h;
   for (h = 0; h < M; h++) // libera cada lista
```

```
    {
        p = tab[h];
        while (p != NULL)
        {
            q = p;
            p = p->prox;
            free(q->chave); // liberando a chave (string) de cada
célula
            free(q);        // antes de liberar a célula
        }
    }
    free(tab); // então libera a tabela
    tab = NULL;
    nChaves = 0;
}
```

- Endereçamento aberto.



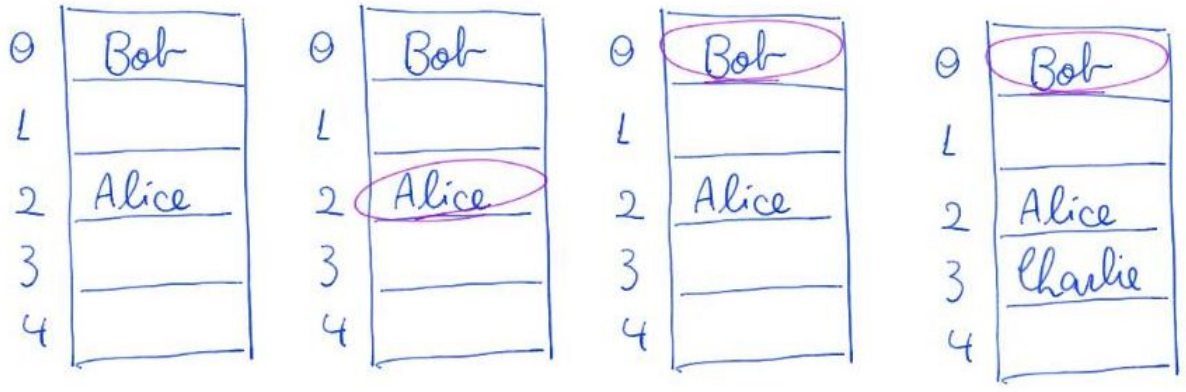
  - Sondagem (probing).
    - Linear: offset segue uma função linear (i)
      - a partir da posição inicial.



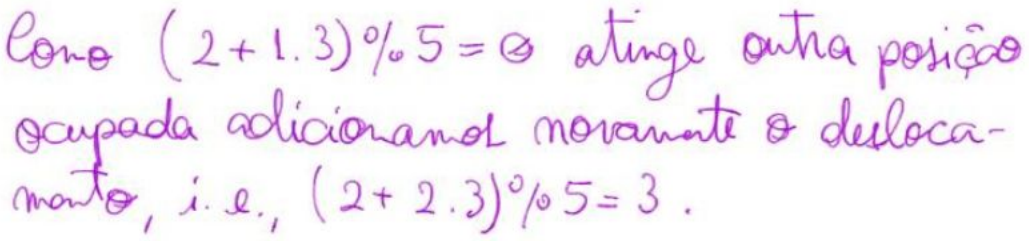
h(Charlie) = 2 Como a posição está ocupada, tenta a posição +1 até encontrar uma posição vazia.

- 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.

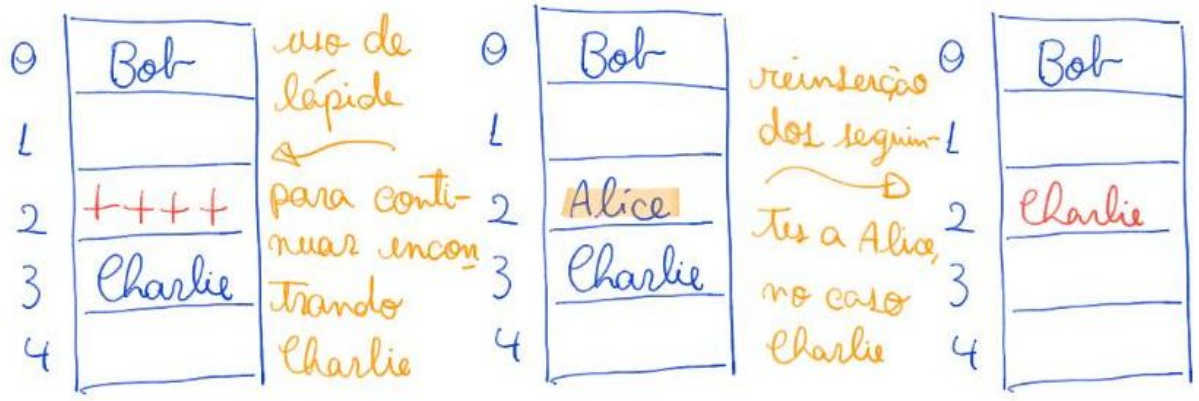
- Re-espalhamento (rehashing ou double hashing).



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



    - Prós: evita gerar aglomerações, já que
        - cada chave é re-espalhada com um offset próprio.
    - Prós: endereçamento aberto ocupa menos espaço.
    - Contra: número de elementos limitados ao tamanho da tabela
        - e remoção é mais complicada de implementar.
            - Opções são o uso de lápides,
                - que marcam uma posição previamente ocupada,
            - e o reposicionamento/reinserção de todos
                - cuja posição foi afetada pelo elemento removido.



    - Exemplo de código com sondagem linear.

```c
#define LIVRE(h) (tab[h].chave == NULL)
#define INCR(h) (h = h == M - 1 ? 0 : h + 1)
//#define INCR(h) (h = (h + 1) % M)

typedef struct celTS CelTS;
struct celTS
{
   Chave chave;
   Valor valor;
};

static CelTS *tab = NULL;
static int nChaves = 0;
static int M; // tamanho da tabela
```

```
void stInit(int max)
{
   int h;
   M = max;
   nChaves = 0;
   tab = mallocSafe(M * sizeof(CelTS));
   for (h = 0; h < M; h++)
       tab[h].chave = NULL;
}

Valor stSearch(Chave chave)
{
   int h = hash(chave, M);
   while (!LIVRE(h) && strcmp(tab[h].chave, chave) != 0)
       INCR(h);
   if (!LIVRE(h)) // se encontrou devolve o valor
       return tab[h].valor;
   return 0; // caso contrário devolve 0. E se o valor for 0? Como
contornar esse problema?
}

void stInsert(Chave chave, Valor valor) // inserção ou edição
{
   CelTS *p;
   int h = hash(chave, M);
   while (!LIVRE(h) && strcmp(tab[h].chave, chave) != 0)
       INCR(h);
   if (LIVRE(h)) // se não encontrou insere
   {
       if (nChaves == M - 1) // nunca preenche a última posição. Por
que?
       {
           printf("Tabela cheia\n");
           return;
```

```
        }
        tab[h].chave = copyString(chave);
        nChaves++;
    }
    tab[h].valor = valor; // atualiza valor do item
}

void stDelete(Chave chave)
{
    int h = hash(chave, M);
    while (!LIVRE(h) && strcmp(tab[h].chave, chave) != 0)
        INCR(h);
    if (LIVRE(h)) // se não encontrou não tem o que remover
        return;
    // remover a chave da tabela
    free(tab[h].chave);
    tab[h].chave = NULL;
    nChaves--;
    // reespalhar as chaves seguintes, cujas posições podem ter sido
afetadas pelo elemento removido
    for (INCR(h); !LIVRE(h); INCR(h))
    {
        Chave chave = tab[h].chave;
        Valor valor = tab[h].valor;
        tab[h].chave = NULL;
        stInsert(chave, valor);
        free(chave);
    }
}

void stFree()
{
    int h;
    for (h = 0; h < M; h++) // liberando as chaves (strings)
        if (!LIVRE(h))
```

```
            free(tab[h].chave);
    free(tab); // antes de liberar a tabela
    tab = NULL;
    nChaves = 0;
}
```

Carga de uma hash table

Carga (load) de uma hash table = |S| / M, sendo

- S o conjunto de dados armazenados e M o tamanho da tabela.

Qual família de estratégias para tratar colisões permite cargas maiores que 1?

- Uma estratégia aloca espaço adicional para cada item que chega.
    - Qual é essa estratégia?
- A outra apenas busca outra posição para o novo item.
    - Qual é essa estratégia?

Observe que, numa hash table com listas encadeadas

- o tempo de acesso esperado é da ordem de 1 + carga.
    - Isso porque, é necessário tempo constante (O(1)) para
        - resolver a hash function,
            - encontrando assim a posição do item na tabela,
    - mais o tempo necessário para percorrer a lista ligada,
        - que tem comprimento médio |S| / M = carga,
            - já que são |S| itens espalhados por M posições.
- Observe a importância da hash function espalhar bem os itens
    - para que valha essa eficiência.
- No pior caso, se todos os itens forem direcionados para a mesma posição,
    - teremos a (in)eficiência de uma única lista encadeada.

No caso de uma hash table com endereçamento aberto bem implementado

- esse tempo cresce de acordo com a função 1 / (1 - carga).
- O resultado deriva do número esperado de moedas
    - que precisamos jogar até obter o primeiro sucesso.
- A metáfora faz sentido porque, se tanto a função de hash
    - quanto a sondagem / reespalhamento forem bem implementados
        - então cada tentativa de alocar o item tem
            - probabilidade de sucesso = (1 - carga) e
            - probabilidade de fracasso = carga.
- Sendo E o número esperado de moedas até o primeiro sucesso, temos que
    - E = 1 + Prob(Fracasso) * E = 1 + carga * E,

            - Isso porque, se a primeira moeda falhou
                - estamos novamente diante do problema original.
        - Portanto, (1 - carga) E = 1 ⇒ E = 1 / (1 - carga).
- Isso significa fator de tempo constante (e baixo) para carga <= 70%,
    - e crescimento veloz quando carga se aproxima de 100%.
- Considere o tempo necessário para
    - encontrar uma posição para o (M - 1)-ésimo item.

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.
- Nesse processo, usar uma versão modificada da sua função de hash,
    - i.e., com % 2M no final e
        - possivelmente usando um primo maior.