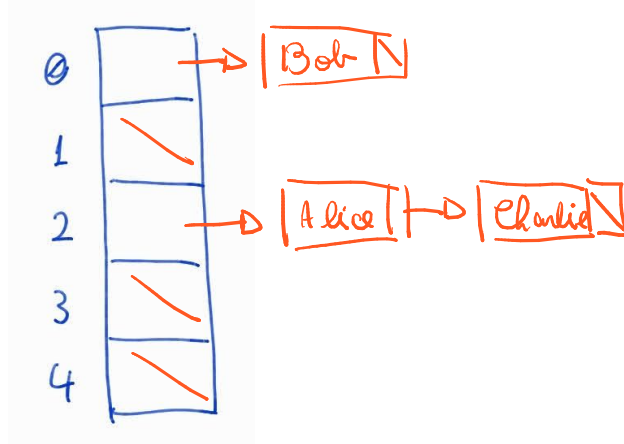


Hash tables, tratando colisões e dimensionando carga

Tratando colisões usando listas ligadas

Cada posição do vetor é uma lista ligada.



$$M = 5$$

$$h(\text{Alice}) = 2$$

$$h(\text{Bob}) = 0$$

$$h(\text{Charlie}) = 2$$

- Inserção pode levar tempo constante, mas consulta e remoção
 - dependem do tamanho da lista, que depende
 - da qualidade da função de espalhamento e
 - do tamanho da tabela Hash.
- Prós: remoção é simples de implementar.
- Contra: ocupa mais espaço.

Exemplo de implementação de tabela Hash 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;
```

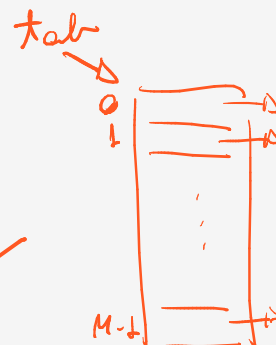
```
    nChaves = 0;
```

```
    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?
}

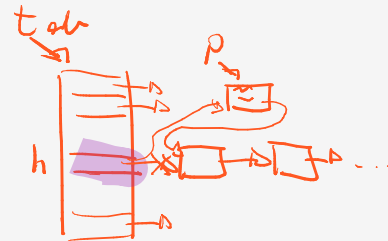
```

- Quiz1: 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 no início da 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
}

```

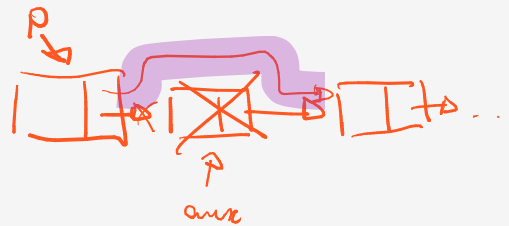
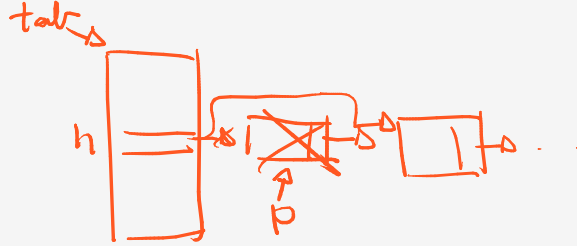


- Quiz2: Como separar a inserção da edição?
- Quiz3: Como implementar inserção que mantém os itens em ordem na lista?

```

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) { // se 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) da célula
             free(q);      // antes de liberar a célula
            )
        }
    }
    free(tab); // então libera a tabela
    tab = NULL;
    nChaves = 0;
}

```

posição

pozicije

$$M = 5$$

$$h(\text{Bob}) = 0$$

E Agoré?

- tenta uma nova de acordo com uma função de offset (deslocamento).
- Sondagem Linear: offset segue uma função linear (i),
 - a partir da posição inicial,
 - sendo i o número da tentativa de re-endereçamento.

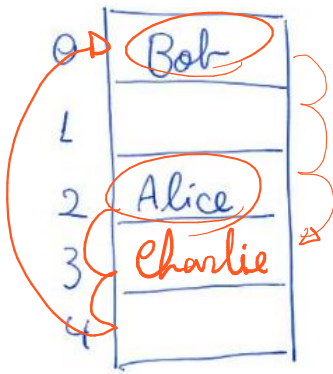
$$h(\text{Charlie}) = 2$$

$\% M$

- **Contra:** costuma gerar aglomerações.

Re-espalhamento (rehashing ou double hashing), abordagem em que

- o offset é calculado usando uma outra função de hash $g()$ sobre a chave.



$$h(\text{Charlie}) = 2$$

$$g(\text{Charlie}) = 3$$

$$(2 + 3) \% M = 0$$

$\uparrow$
 $M = 5$

$$g(\text{Alice}) = 1$$

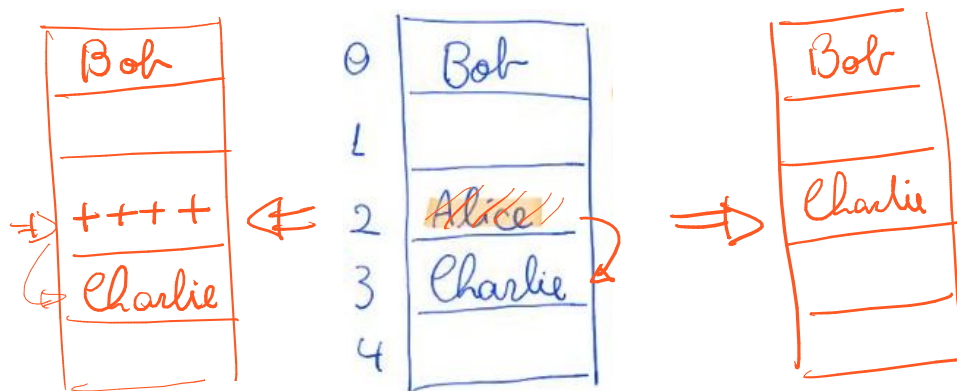
$$(2 + 2 \times 3) \% 5 =$$

- Prós: evita gerar aglomerações, já que
 - cada chave é re-espalhada com um offset próprio.

Considerações sobre endereçamento aberto:

- Prós: endereçamento aberto ocupa menos espaço.
- Contra: número de elementos limitados ao tamanho da tabela
 - e remoção é mais complicada de implementar.
- Opções para remoção são
 - o uso de lápides,
 - que marcam uma posição previamente ocupada,
 - e o re-espalhamento/reinserção de todos
 - cuja posição foi afetada pelo elemento removido.

$$h(\text{Charlie}) = 2$$



Exemplo de implementação de tabela Hash com sondagem linear:

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

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

```
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
}
```

$h(chave) = k$



```
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) { // não ocupa 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--;

    // re-espalhar as chaves seguintes, cujas posições podem ter sido
    afetadas pelo elemento removido.
    for (1.INCR(h); 2.!LIVRE(h); 3.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 tabela Hash

razão

Carga (load) de uma tabela Hash = $|S| / M$, sendo

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

Quiz: qual estratégia para tratar colisões permite cargas maiores que 1?

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

Observe que, numa tabela Hash com listas ligadas

- o tempo de acesso esperado é da ordem de $1 + \text{carga}$.
- Isso porque, é necessário tempo constante ($O(1)$) para
 - resolver a hash function, encontrando a posição do item na tabela,
- mais o tempo necessário para percorrer a lista ligada,
 - que tem comprimento médio $|S| / M = \text{carga}$,
 - 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 ligada.

No caso de uma tabela Hash com endereçamento aberto bem implementado

- esse tempo cresce de acordo com a função $1 / (1 - \text{carga})$.
- 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 principal
 - quanto a sondagem / reespalhamento forem bem implementados
 - então cada tentativa de alocar o item tem
 - probabilidade de fracasso = *carga*
 - probabilidade de sucesso = $1 - \text{carga}$
- Sendo E o número esperado de moedas até o primeiro sucesso, temos que
$$E = 1 + \text{Prob. (Fracasso)} E = 1 + \text{carga} \cdot E$$
 - Isso porque, se a primeira moeda falhou
 - estamos novamente diante do problema original.
 - Portanto, $(1 - \text{carga}) E = 1 \Rightarrow E = 1 / (1 - \text{carga})$
- Na prática, isso significa tempo constante (e baixo) para carga $\leq 70\%$,
 - e crescimento veloz quando carga se aproxima de 100%.
- Para perceber isso, considere o tempo necessário para
 - encontrar uma posição para o (M - 1)-ésimo item.

carga = 1/2
$$E = \frac{1}{(1 - \frac{1}{2})} = 2$$

carga = (M - 1) / M
$$E = \frac{1}{1 - \frac{(M-1)}{M}} = \frac{M}{M - M + 1}$$

E = M

Como tabelas Hash 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/re-insira 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.

35%

*Quiz: implemen
e redimensionamento dinâmico*