Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

AULA 16

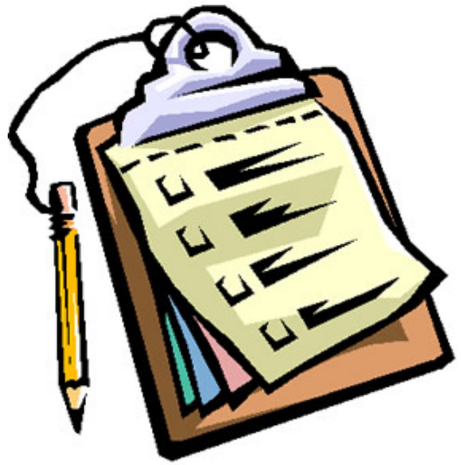
Estrutura de dados básico I (EDB1)

Prof. Msc. Janiheryson Felipe (Felipe)

Natal, RN
2023

OBJETIVOS DA AULA

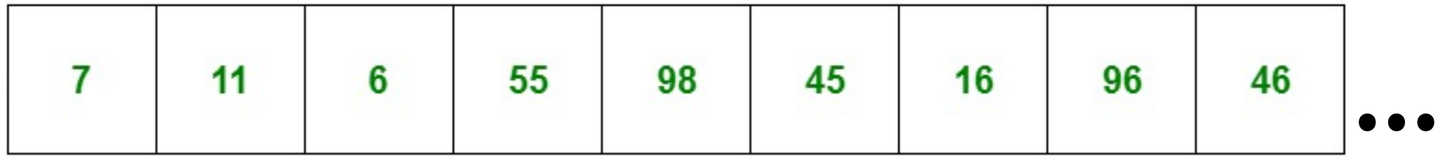
- Apresentar os conceitos de dispersão
  - Conhecer a estrutura das tabelas hash
  - Entender o que é uma colisão e como tratá-la;
  - Entender o processo de endereçamento direto;

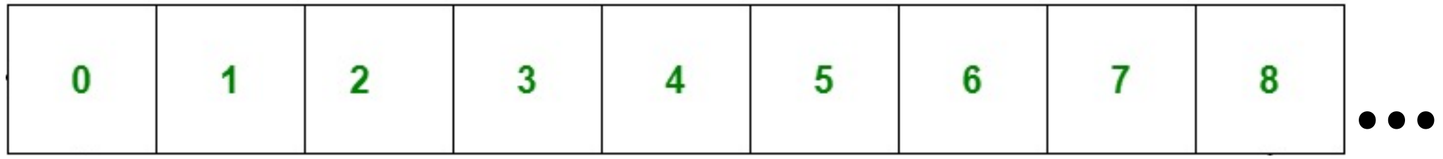


TABELAS DE DISPERSÃO

PROBLEMA DO ENDEREÇAMENTO DIRETA

Imagine que existe uma uma empresa dá a cada um de seus funcionário uma matricula aleatória com três dígitos. Essa matricula é armazenada em um vetor (sequencial), na qual a matrícula tambem será o index desse vetor. Desta forma serão necessarios 1000 posições nesse vetor para armazenar todos as posibilidades.





TABELAS DE DISPERSÃO (HASH TABLE)

Tabelas de dispersão, também conhecidas como hash tables, são estruturas de dados utilizadas para armazenar e recuperar dados de maneira eficiente. Elas são amplamente utilizadas em programação devido à sua capacidade de fornecer acesso rápido aos dados com uma complexidade média de tempo constante (O(1)) para operações como inserção, busca e remoção.

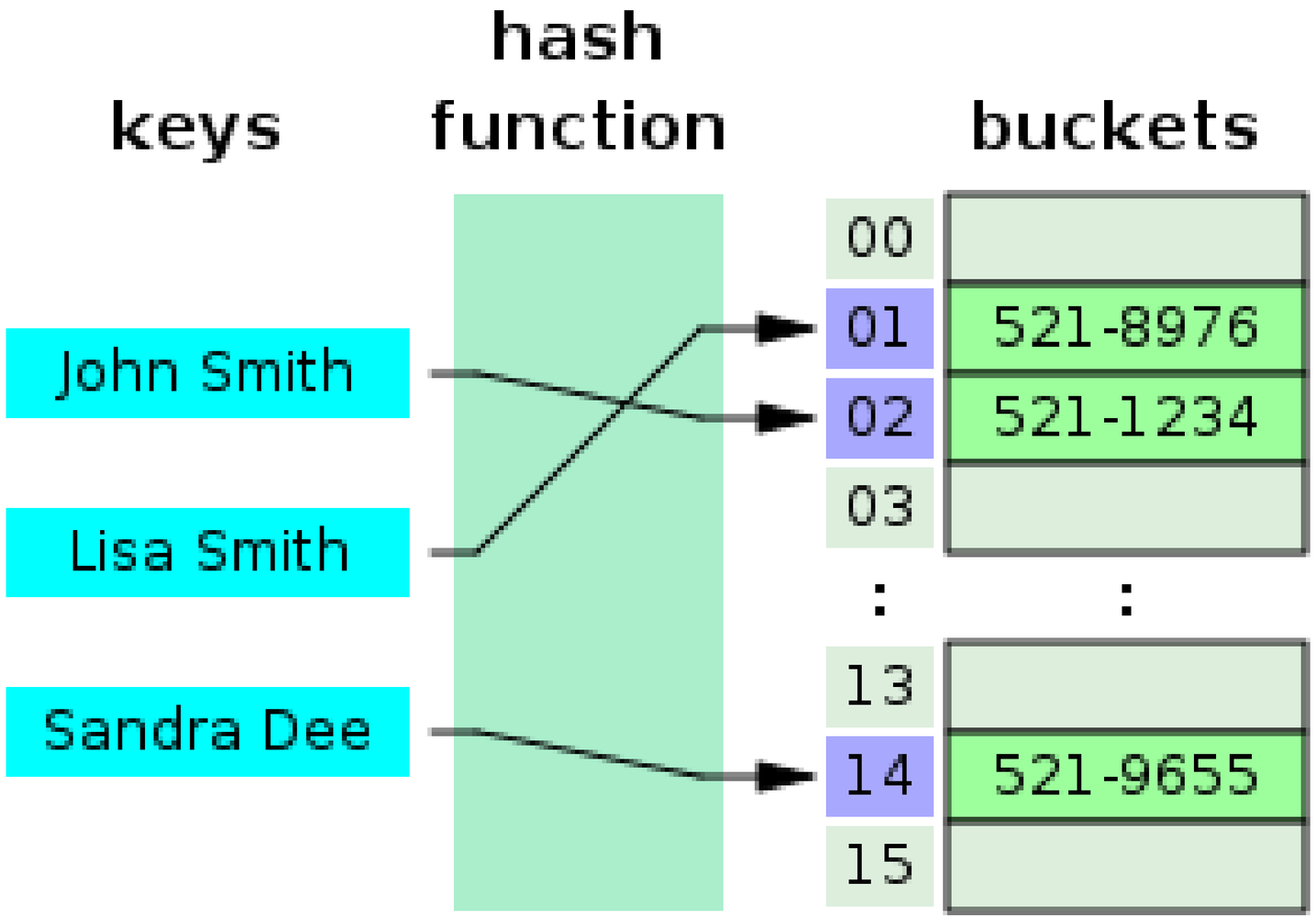
TABELAS DE DISPERSÃO (HASH TABLE)

Uma tabela de dispersão é composta por uma matriz (array) de "slots" ou "buckets", onde os dados são armazenados. Cada slot na tabela pode conter um ou mais elementos, dependendo da implementação específica. A posição de um elemento dentro da tabela de dispersão é determinada através de uma função de dispersão (hash function), que mapeia a chave do elemento para um índice na tabela.

TABELAS DE DISPERSÃO (HASH TABLE)

A função de dispersão calcula um valor numérico chamado "hash code" a partir da chave do elemento. Esse valor é então usado para determinar a posição do elemento na tabela. Idealmente, a função de dispersão deve distribuir os elementos de forma uniforme por toda a tabela para evitar colisões, ou seja, dois elementos diferentes que mapeiam para o mesmo slot. No entanto, colisões podem ocorrer e precisam ser tratadas.

TABELAS DE DISPERSÃO (HASH TABLE)



TABELAS DE DISPERSÃO (HASH TABLE)

Para garantir um bom desempenho de uma tabela de dispersão, é importante escolher uma boa função de dispersão que minimize as colisões e distribua uniformemente os elementos. Além disso, é necessário ajustar o tamanho da tabela de acordo com a quantidade esperada de elementos, pois uma tabela muito pequena pode aumentar o número de colisões, enquanto uma tabela muito grande pode desperdiçar espaço.

CARACTERÍSTICAS DESEJÁVEIS DAS FUNÇÕES DE HASH

- Produzir um número baixo de colisões
  - Difícil, pois depende da distribuição dos valores de chave

    - Exemplo: Pedidos que usam o ano e mês do pedido como parte da chave
    - Se a função h realçar estes dados, haverá muita concentração de valores nas mesmas faixas

CARACTERÍSTICAS DESEJÁVEIS DAS FUNÇÕES DE HASH

- Ser facilmente computável
  - Se a tabela estiver armazenada em disco (nosso caso), isso não é tão crítico, pois a operação de I/O é muito custosa, e dilui este tempo
  - Das 3 condições, é a mais fácil de ser garantida
- Ser uniforme
  - Idealmente, a função h deve ser tal que todos os compartimentos possuam a mesma probabilidade de serem escolhidos
  - Difícil de testar na prática

CARACTERÍSTICAS DAS FUNÇÕES DE HASH

- OBS:
  - A mesma função de hash usada para inserir os registros é usada para buscar os registros

h(x) = x mod 7

Encontrar o registro de chave 90
- 90 mod 7 = 6

Encontrar o registro de chave 7
- 7 mod 7 = 0
- Compartimento 0 está vazio: registro não está armazenado na tabela

Encontrar o registro de chave 8
- 8 mod 7 = 1

0	
1	50
2	23
3	10
4	11
5	
6	90

FATOR DE CARGA

- O fator de carga de uma tabela hash é a = n/m, onde n é o número de registros armazenados na tabela
- O número de colisões cresce rapidamente quando o fator de carga aumenta
  - Uma forma de diminuir as colisões é diminuir o fator de carga
  - Mas isso não resolve o problema: colisões sempre podem ocorrer

EXEMPLOS DE FUNÇÕES DE HASH

- Algumas funções de hash são bastante empregadas na prática por possuírem algumas das características anteriores:
  - Método da Divisão
  - Método da Dobra
  - Método da Multiplicação

MÉTODO DA DIVISÃO

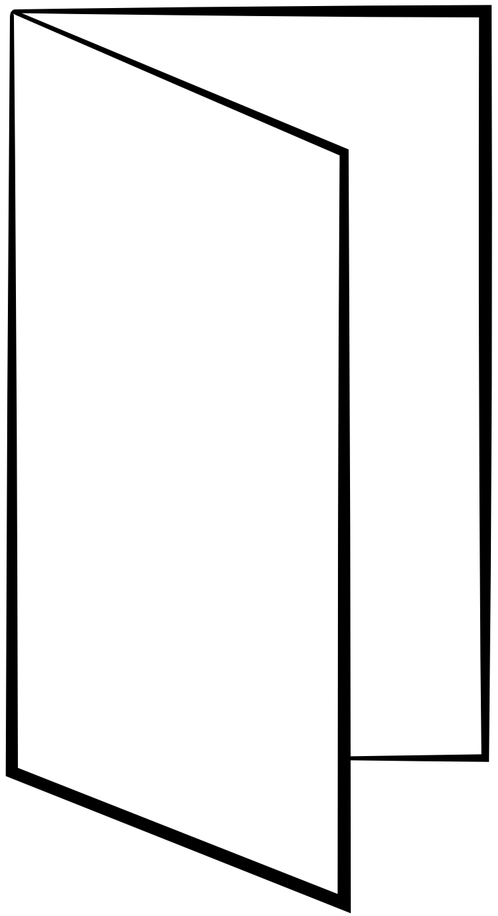
Uso da função mod:

- **h(x) = x mod m**
  - onde m é a dimensão da tabela

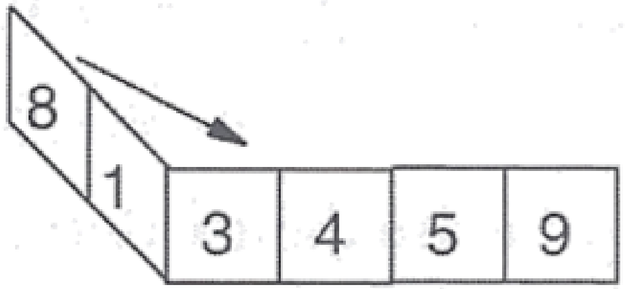
- Alguns valores de m são melhores do que outros
- Estudos apontam bons valores de m:
  - Escolher m de modo que seja um número primo não próximo a uma potência de 2;
  - ou Escolher m tal que não possua divisores primos menores do que 20

MÉTODO DA DOBRA

- Suponha a chave como uma sequencia de dígitos escritos em um pedaço de papel
- O método da dobra consiste em “dobrar” este papel, de maneira que os dígitos se superponham
- Os dígitos então devem ser somados, sem levar em consideração o “vai-um”

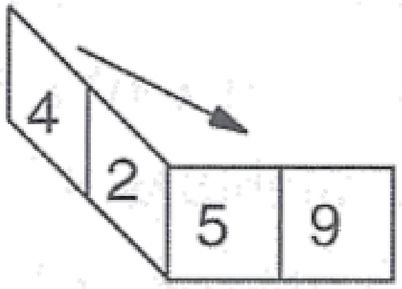


MÉTODO DA DOBRA



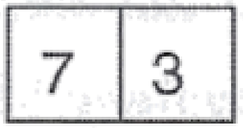
8+4=~~1~~2

1+3=4



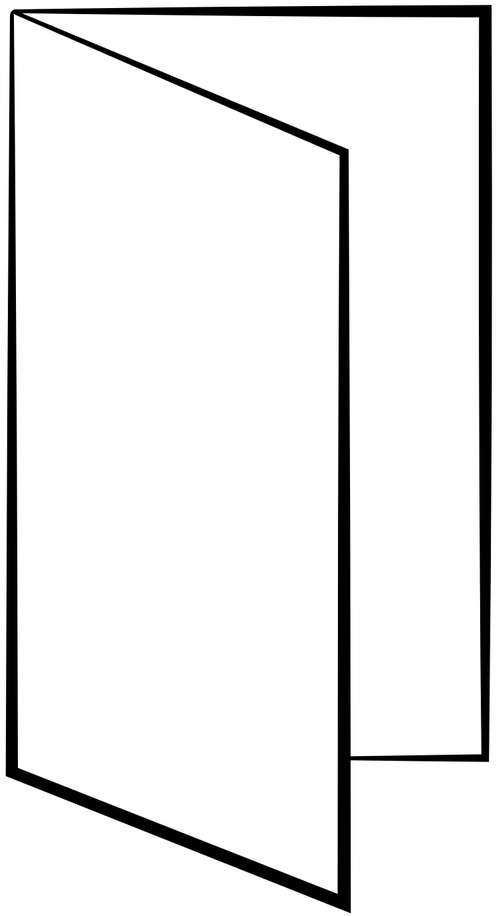
4+9=~~1~~3

2+5=7



MÉTODO DA DOBRA

- A posição onde a dobra será realizada, e quantas dobras serão realizadas, depende de quantos dígitos são necessários para formar o endereço base
- O tamanho da dobra normalmente é do tamanho do endereço que se deseja obter



MÉTODO DA MULTIPLICAÇÃO

- Multiplicar a chave por ela mesma
- Armazenar o resultado numa palavra de b bits
- Descartar os bits das extremidades direita e esquerda, um a um, até que o resultado tenha o tamanho de endereço desejado

MÉTODO DA MULTIPLICAÇÃO

Exemplo:

- chave 12 12 x 12 = 144
- 144 representado em binário: 10010000
- Armazenar em 10 bits: 0010010000
- Obter endereço de 6 bits (endereços entre 0 e 63)

MÉTODO DA MULTIPLICAÇÃO

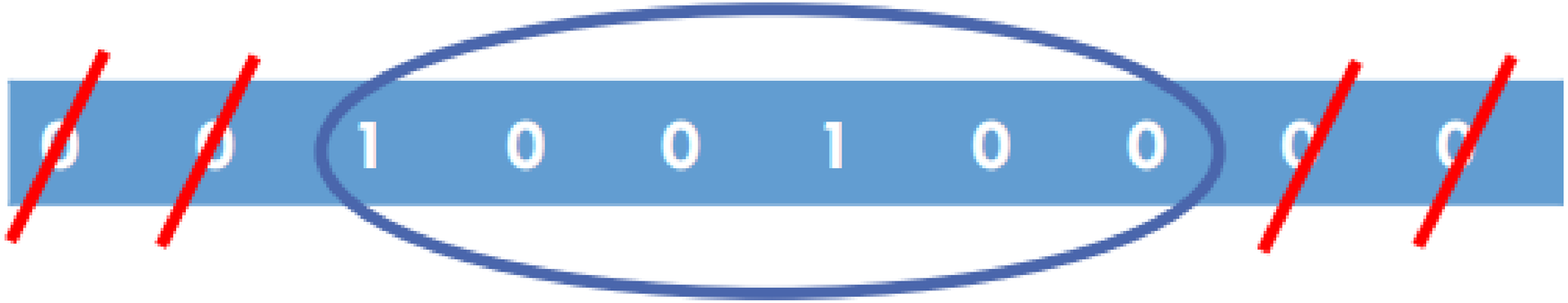
Exemplo:

- chave 12 12 x 12 = 144
- 144 representado em binário: 10010000
- Armazenar em 10 bits: 0010010000
- Obter endereço de 6 bits (endereços entre 0 e 63)

MÉTODO DA MULTIPLICAÇÃO

Exemplo:

- chave 12 12 x 12 = 144
- 144 representado em binário: 10010000
- Armazenar em 10 bits: 0010010000
- Obter endereço de 6 bits (endereços entre 0 e 63)



= endereço 36

PRINCIPAIS USOS

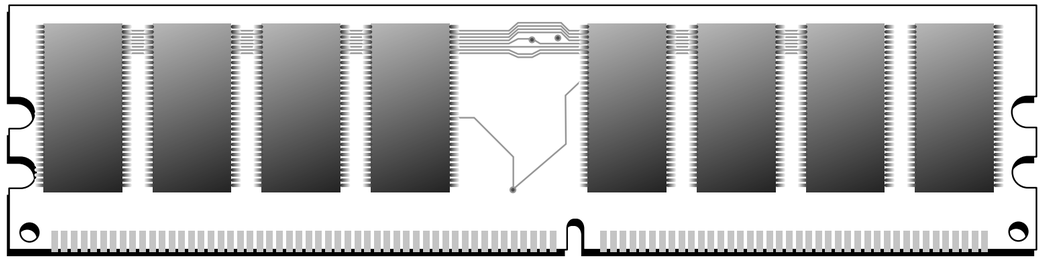
As tabelas de dispersão são amplamente utilizadas em diversas aplicações, como caches de memória, bancos de dados, compiladores e algoritmos de busca e indexação eficientes. Elas fornecem uma maneira rápida e eficiente de armazenar e recuperar dados, desde que a função de dispersão seja bem escolhida e as colisões sejam tratadas adequadamente.

COMPARANDO COM OUTRAS ESTRUTURAS

Apesar das tabelas de dispersão terem em média tempo constante de busca, o tempo gasto no desenvolvimento é significativo. Avaliar uma boa função de espalhamento é um trabalho duro e profundamente relacionado à estatística. Na maioria dos casos soluções mais simples como uma lista encadeada devem ser levados em consideração. Isso não é um problema a não ser que você esteja criando todas as suas estruturas de dados a partir do zero.

COMPARANDO COM OUTRAS ESTRUTURAS

Os dados na memória ficam aleatoriamente distribuídos, o que também causa sobrecarga no sistema. Além disso, e mais importante, o tempo gasto na depuração e remoção de erros é maior do que nas árvores balanceadas, que também podem ser levadas em conta para solução do mesmo tipo de problema.



COMPARANDO COM OUTRAS ESTRUTURAS

No entanto, a maioria das linguagens de programação modernas e bibliotecas auxiliares para linguagens de baixo e médio nível dispõem de boas estruturas de dados usando tabelas de dispersão implementadas de forma bem consistente, prontas para serem usadas, mitigando não só a questão de tempo de desenvolvimento e depuração, como tendo soluções para o espalhamento em memória. Com isso em mente, as limitações, abaixo, podem ser uma consideração mais importante.

LIMITAÇÕES

A tabela de dispersão é uma estrutura de dados do tipo dicionário, que não permite armazenar elementos repetidos, recuperar elementos sequencialmente (ordenação), nem recuperar o elemento antecessor e sucessor. Para otimizar a função de dispersão é necessário conhecer a natureza da chave a ser utilizada. No pior caso, a ordem das operações pode ser $O(N^2)$, caso em que todos os elementos inseridos colidirem. As tabelas de dispersão com endereçamento aberto podem necessitar de redimensionamento.

COLISÕES E TABELAS HASH

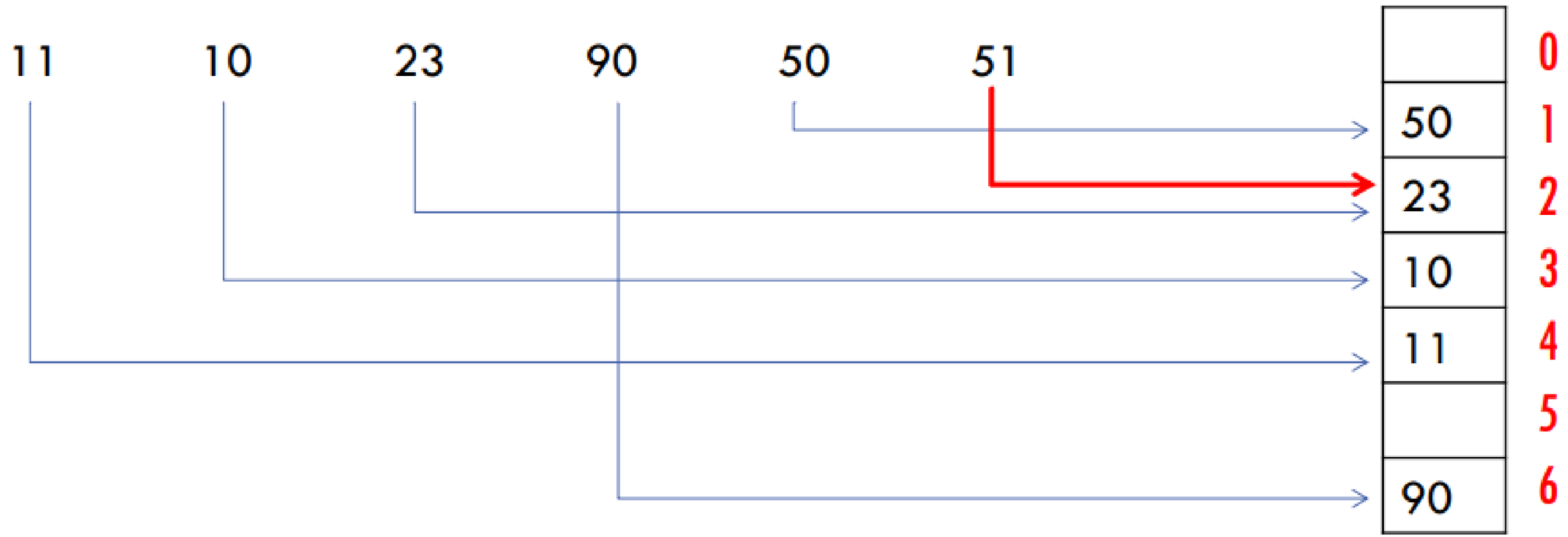
Uma colisão em uma tabela hash ocorre quando dois ou mais elementos têm o mesmo valor de hash e, consequentemente, são mapeados para o mesmo slot na tabela. Isso pode acontecer devido à natureza limitada do espaço de armazenamento da tabela e à possibilidade de ter um número maior de elementos do que slots disponíveis.

COLISÕES E TABELAS HASH

As colisões são inevitáveis em tabelas hash, especialmente quando a quantidade de elementos é grande ou quando a função de dispersão não é capaz de distribuir uniformemente os elementos pela tabela. No entanto, é importante lidar com as colisões de forma adequada para garantir o correto funcionamento da tabela.

COLISÕES E TABELAS HASH

$h(x) = x \bmod 7$



A CHAVE 51 COLIDE COM A CHAVE 23 E NÃO PODE SER INSERIDA NO ENDEREÇO 2!
SOLUÇÃO: USO DE UM PROCEDIMENTO ESPECIAL PARA ARMAZENAR A CHAVE 51 (TRATAMENTO DE COLISÕES)

ENCADEAMENTO DIRETO

Cada slot da tabela contém uma lista encadeada na qual os elementos com o mesmo valor hash são armazenados. Quando ocorre uma colisão, o novo elemento é simplesmente adicionado à lista correspondente.

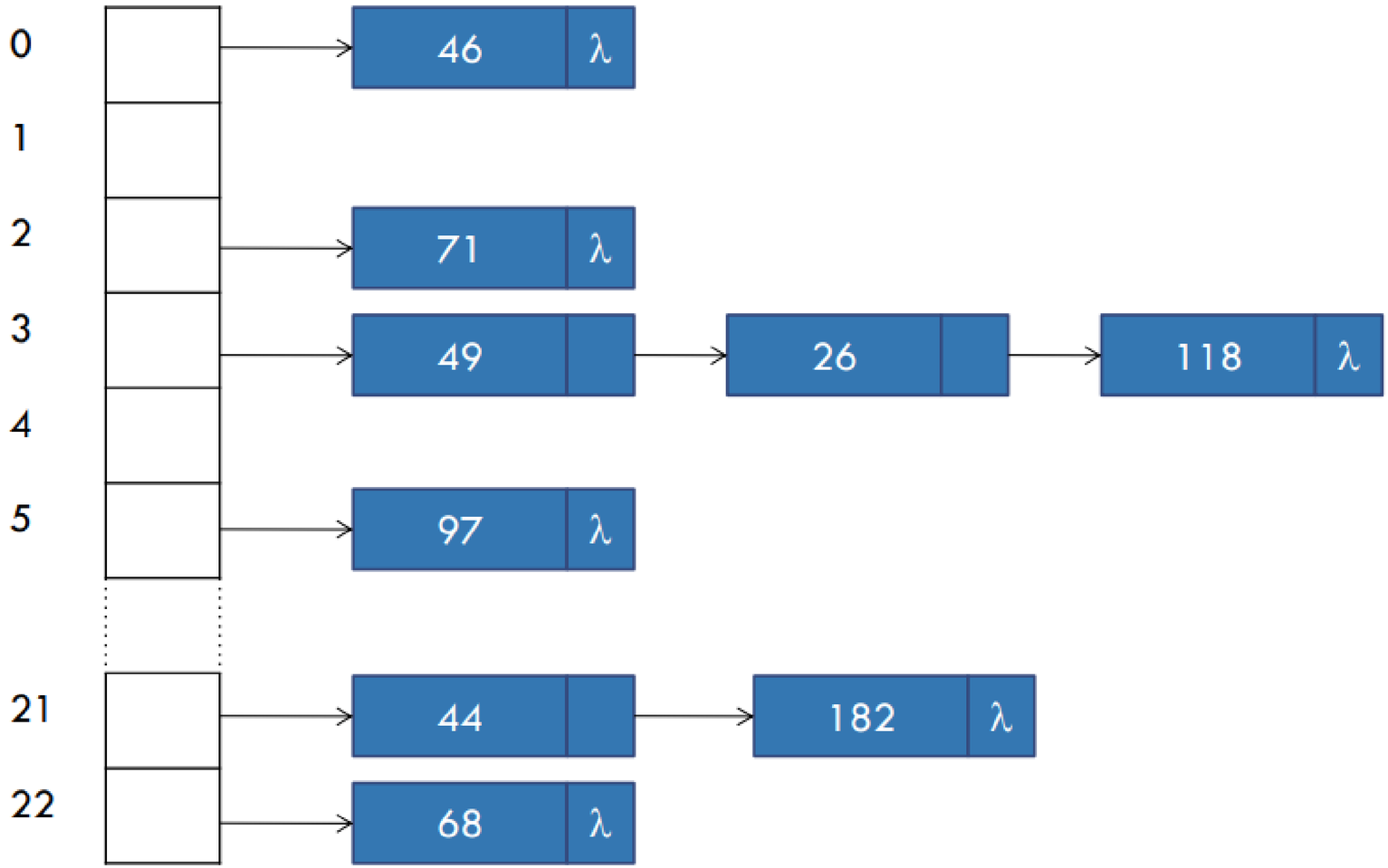
A inserção na tabela requer uma busca e inserção dentro da lista encadeada; uma remoção requer atualizar os índices dentro da lista, como se faria normalmente.

ENCADEAMENTO DIRETO

Estruturas de dados alternativas podem ser utilizadas no lugar das listas encadeadas. Por exemplo, se utilizarmos uma árvore balanceada, podemos melhorar o tempo médio de acesso da tabela de dispersão para nlogn ao invés de n^2. Mas como as listas de colisão são projetadas para serem curtas, a sobrecarga causada pela manutenção das árvores pode fazer o desempenho cair.

ENCADEAMENTO DIRETO

h(x) = x mod 23



ENCADEAMENTO DIRETO

Busca por um registro de chave x:

1. Calcular o endereço aplicando a função h(x)
2. Percorrer a lista encadeada associada ao endereço
3. Comparar a chave de cada nó da lista encadeada com a chave x, até encontrar o nó desejado
4. Se final da lista for atingido, registro não está lá

ENCADEAMENTO DIRETO

Inserção de um registro de chave x

1. Calcular o endereço aplicando a função h(x)
2. Buscar registro na lista associada ao endereço h(x)
3. Se registro for encontrado, sinalizar erro
4. Se o registro não for encontrado, inserir no final da lista

ENCADEAMENTO DIRETO

Exclusão de um registro de chave x

1. Calcular o endereço aplicando a função h(x)
2. Buscar registro na lista associada ao endereço h(x)
3. Se registro for encontrado, excluir registro
4. Se o registro não for encontrado, sinalizar erro

IMPLEMENTAÇÃO EM MEMÓRIA

```
#define LIBERADO 0
#define OCUPADO 1


typedef struct aluno {
    int matricula;
    float cr;
    int prox;
    int ocupado;
} TAluno;


//Hash é um vetor que será alocado dinamicamente
typedef TAluno *Hash;
```

IMPLEMENTAÇÃO EM MEMÓRIA

```
TAluno *aloca(int mat, float cr, int status, int prox) {
    TAluno *novo = (TAluno *) malloc(sizeof(TAluno));
    novo->matricula = mat;
    novo->cr = cr;
    novo->ocupado = status;
    novo->prox = prox;
    return novo;
}

void inicializa(Hash *tab, int m) {
    int i;
    for (i = 0; i < m; i++) {
        tab[i] = aloca(-1, -1, LIBERADO, -1);
    }
}
```

IMPLEMENTAÇÃO EM MEMÓRIA

```
int busca(Hash *tab, int m, int mat, int *achou) {
    *achou = -1;
    int temp = -1;
    int end = hash(mat, m);
    while (*achou == -1) {
        TAluno *aluno = tab[end];
        if (!aluno->ocupado) {//achou compartimento livre -- guarda para
 retorná-lo caso chave não seja encontrada
            temp = end;
        }
        if (aluno->matricula == mat && aluno->ocupado) {
            //achou chave procurada
            *achou = 1;
        } else {
            if (aluno->prox == -1) {
                //chegou no final da lista encadeada
                *achou = 0;
                end = temp;
            } else {
                //avança para o próximo
                end = aluno->prox;
            }
        }
    }
    return end;
}
```

IMPLEMENTAÇÃO EM MEMÓRIA

```
void exclui(Hash *tab, int m, int mat) {
     int achou;
     int end = busca(tab, m, mat, &achou);
     if (achou) {
          //remove marcando flag para liberado
           tab[end]->ocupado = LIBERADO;
     } else {
          printf("Matrícula não encontrada. Remoção não realizada!");
     }
 }
```

TABELA HASH DINÂMICA

Em vez de lidar com colisões individualmente, uma tabela hash dinâmica pode ser redimensionada automaticamente à medida que novos elementos são inseridos. Quando a taxa de ocupação da tabela atinge um determinado limite, a tabela é redimensionada para aumentar o número de slots disponíveis, reduzindo assim a probabilidade de colisões.

REHASHING

Em alguns casos, quando uma colisão ocorre, a função de dispersão pode ser aplicada novamente para calcular um novo valor de hash a ser usado para encontrar um slot diferente na tabela. Esse processo é conhecido como rehashing.

DÚVIDAS???