DCC
DEPARTAMENTO DE
CIÊNCIA DA COMPUTAÇÃO

UFMG
UNIVERSIDADE FEDERAL
DE MINAS GERAIS

Estrutura de Dados

Pesquisa em memória primária: Hashing

Professores: Luiz Chaimowicz e Raquel Prates

Hashing

- Algoritmos vistos efetuam comparações para localizar uma chave.
- Hashing usar outra estratégia: transformação aritmética sobre a chave de pesquisa
  - Os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.
  - Busca por meio de operações aritméticas que transformam a chave em endereços em uma tabela.

Hashing

- Um método de pesquisa com o uso da transformação de chave é constituído de duas etapas principais:
  1. Computar o valor da **função de transformação**, a qual transforma a chave de pesquisa em um endereço da tabela.
  2. Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é necessário existir um método para **lidar com colisões**.

Função de Transformação

- Uma função de transformação deve mapear chaves em inteiros dentro do intervalo [0...M − 1], onde M é o tamanho da tabela.

- A função de transformação ideal é aquela que:
  - Seja simples de ser computada.
  - Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer.

Hashing

- Exemplo:
  - Usa o resto da divisão por M (onde k é a chave)

    F(k) = k % M

  - F(x) = x % 10;

  - F(1) = 1 % 10 = 1
  - F(22) = 22 % 10 = 2
  - F(43) = 43 % 10 = 3
  - F(57) = 57 % 10 = 7
  - F(125) = 125 % 10 = 5

Hashing – Chaves não numéricas

- Chave[i] corresponde à representação ASCII do i-ésimo caractere da chave.
- Exemplo: Considerando a i-ésima letra do alfabeto representada por i e a função de transformação h(Chave) = Chave mod M (M=10).

  A = 1 → 1

  L = 12 → 2

  G = 7 → 7

  O = 15 → 5

Chaves não numéricas

- Se a chave for um string, soma-se o valor dos seus caracteres

  A = 1 → 1

  L = 12 → 2

  G = 7 → 7

  O = 15 → 5

  ALGO = 1+12+7+15 = 35 → 5

- Como tratar anagramas:

  Exemplo: ALGO, GALO, GOLA e LAGO

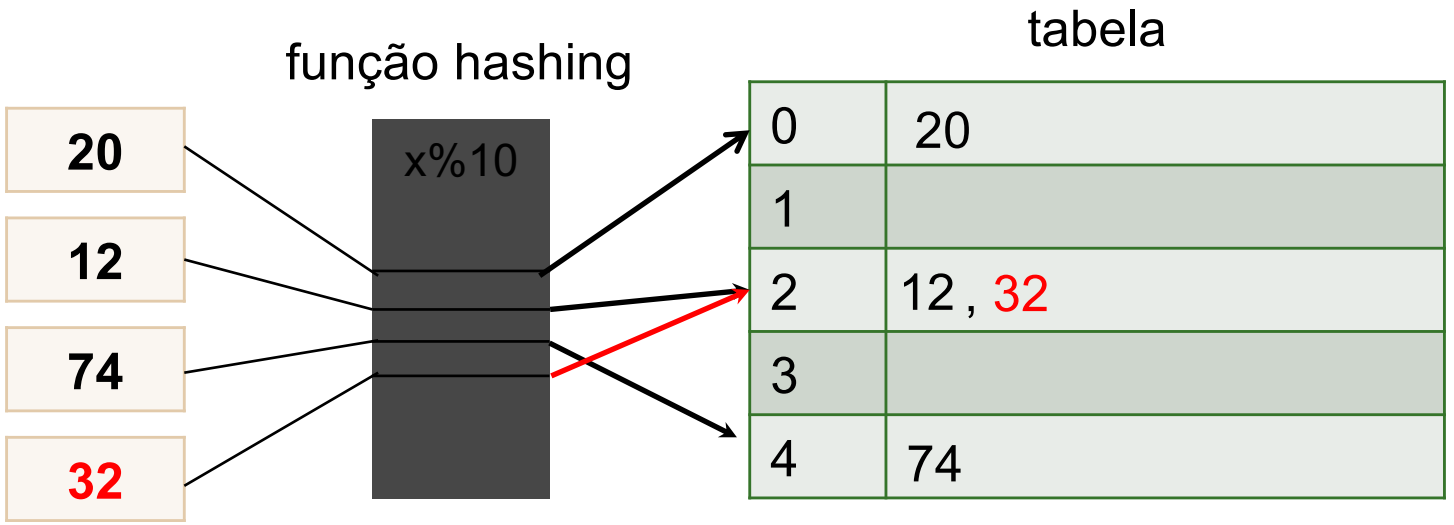
- Pode-se atribuir um peso, *p[i]* à posição da letra na palavra, $1 \leq i \leq n$.

$$H = \sum_{i=1}^{n} Chave[i] \times p[i]$$

Hashing

- Principal Problema: colisões
  - Exemplo:

Função de hashing: f(x) = x % 10



Hashing: colisões

- Qualquer que seja a função de transformação, algumas colisões irão ocorrer fatalmente, e tais colisões têm de ser resolvidas de alguma forma.
- Mesmo que se obtenha uma função de transformação que distribua os registros de forma uniforme entre as entradas da tabela, existe uma alta probabilidade de haver colisões.

Hashing: colisões

- Hashing – Paradoxo do Aniversário
  - Chave é a data de nascimento
  - Tabela possui 365 entradas
  - A probabilidade p de se inserir 2 itens sem colisão em uma tabela de tamanho 365 considerando um conjunto com mais de 23 itens é:
    - p ~ 50.7%

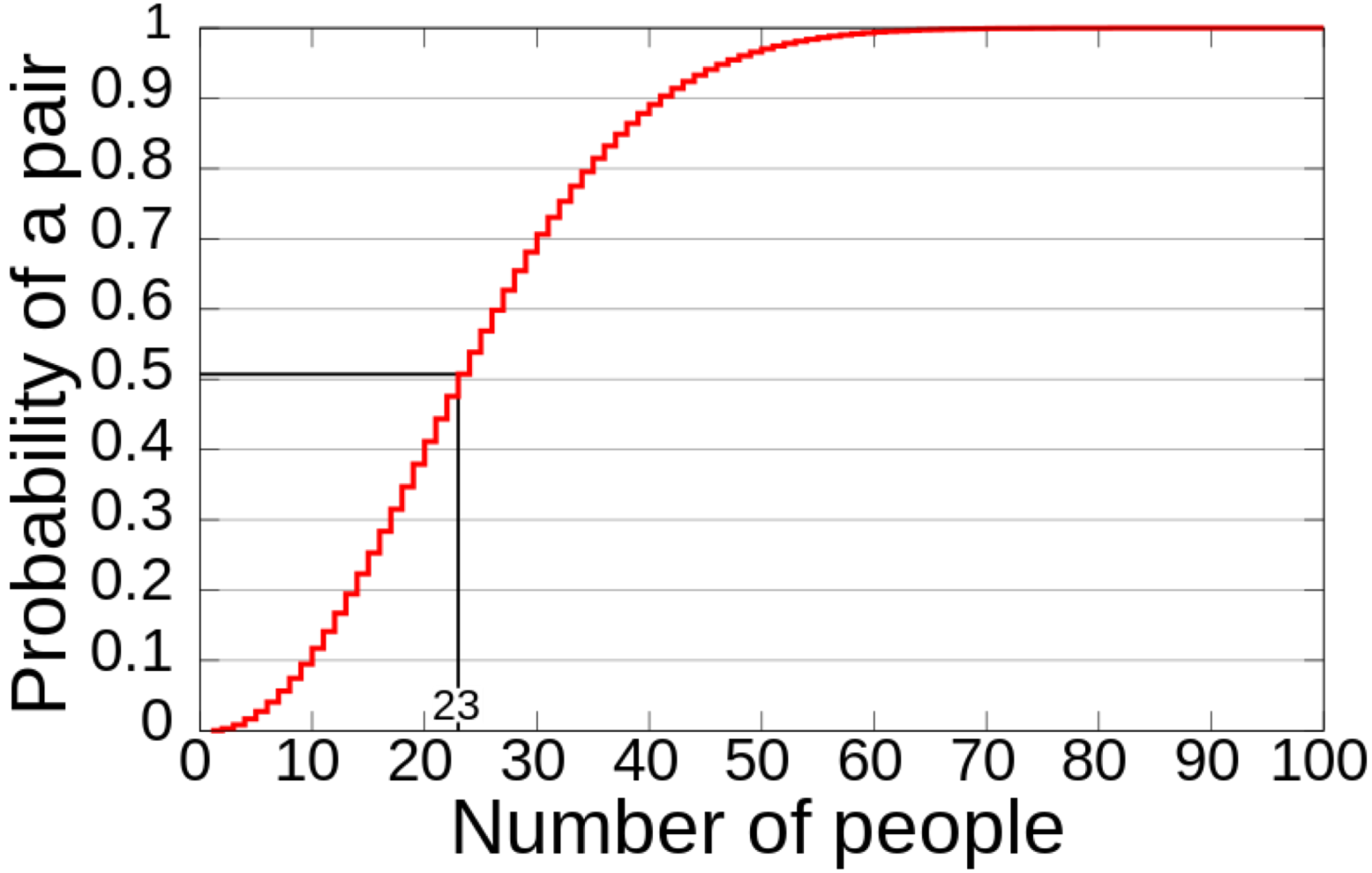
Hashing: colisões

- O paradoxo do aniversário (Feller,1968, p. 33), diz que em um grupo de 23 ou mais pessoas, juntas ao acaso, existe uma chance maior do que 50% de que 2 pessoas comemorem aniversário no mesmo dia.

n	P(n)
10	12%
20	41%
23	50.7%
30	70%
50	97%

Hashing: colisões

- Paradoxo do aniversário



Hashing: colisões

- A probabilidade p de se inserir N itens consecutivos sem colisão em uma tabela de tamanho M é:

$$p = \frac{M-1}{M} \times \frac{M-2}{M} \times \ldots \times \frac{M-N+1}{M} = \prod_{i=1}^{N} \frac{M-i+1}{M} = \frac{M!}{(M-N)!M^N}$$

Hashing: colisões

Seja uma tabela com 50.063.860 de posições (M):

Chaves (N)	Chance de colisão	Fator de carga (N/M)
1000	0.995%	0.002%
2000	3.918%	0.004%
4000	14.772%	0.008%
6000	30.206%	0.012%
8000	47.234%	0.016%
10000	63.171%	0.020%
12000	76.269%	0.024%
14000	85.883%	0.028%
16000	92.248%	0.032%
18000	96.070%	0.036%
20000	98.160%	0.040%
22000	99.205%	0.044%

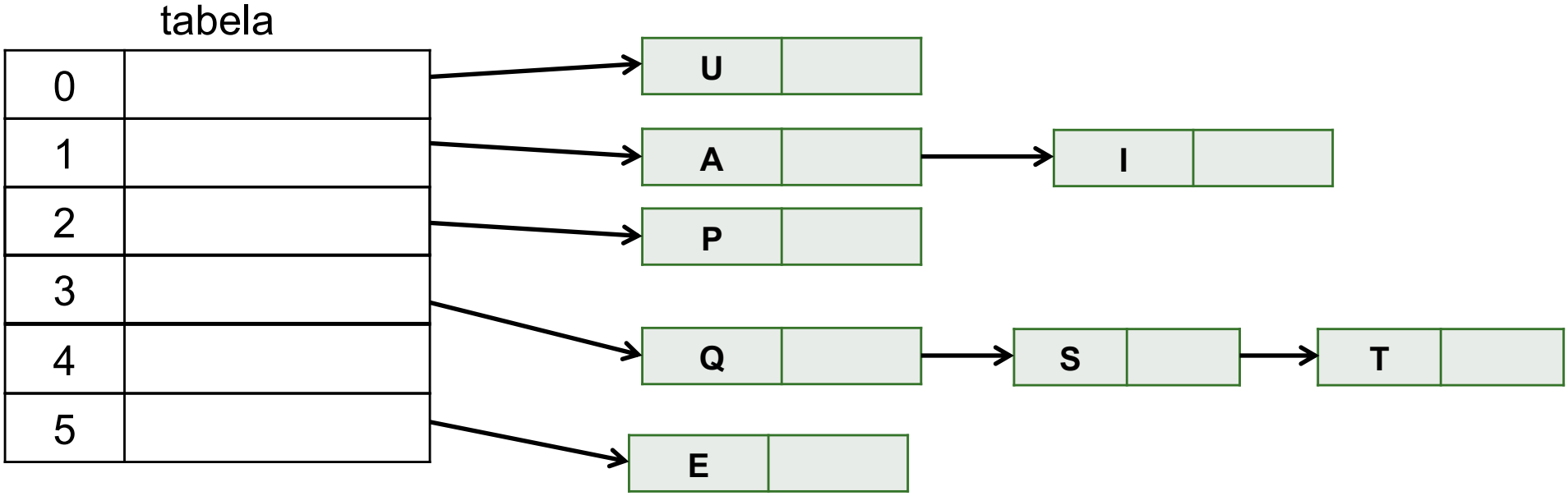
Resolução de Colisões

- Encadeamento (Listas Encadeadas)
- Endereçamento aberto

RESOLUÇÃO DE COLISÕES – LISTA ENCADEADA

Resolução de Colisões - Encadeamento

- Cria uma lista encadeada para cada endereço da tabela.
- Todas as chaves com mesmo endereço na tabela são encadeadas em uma lista linear.

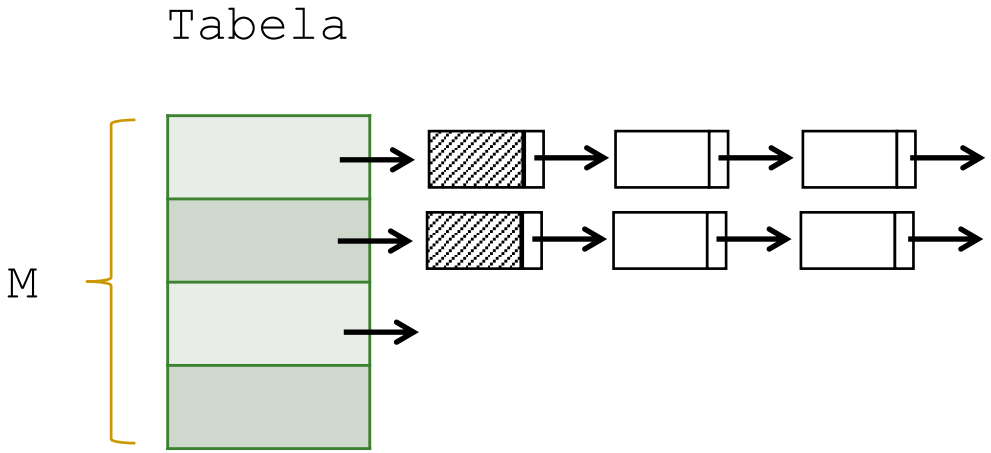


Estrutura de Dados para Encadeamento

```
class Hash_LE
{
    public:
        Hash_LE();
        TipoItem Pesquisa(TipoChave chave);
        void Insere(TipoItem item);
        void Remove(TipoChave chave);

    private:
        static const int M = 7;
        int Hash(TipoChave Chave);
        ListaEncadeada Tabela[M];
};
```

Observação:
O construtor default já inicializa o vetor com as listas vazias



Recapitulando a Classe Lista Encadeada

```
class ListaEncadeada : public Lista {
    public:
        ListaEncadeada();
        ~ListaEncadeada();

        TipoItem GetItem(int pos);
        void SetItem(TipoItem item, int pos);
        void InsereInicio(TipoItem item);
        void InsereFinal(TipoItem item);
        void InserePosicao(TipoItem item, int pos);
        TipoItem RemoveInicio();
        TipoItem RemoveFinal();
        TipoItem RemovePosicao(int pos);
***     TipoItem RemoveItem(TipoChave c);
        TipoItem Pesquisa(TipoChave c);
        void Imprime();
        void Limpa();
    private:
        TipoCelula* primeiro;
        TipoCelula* ultimo;
        TipoCelula* Posiciona(int pos, bool antes);
};
```

*** Método adicionado para a remoção de um item específico

Recapitulando a Classe Lista Encadeada

```
class TipoItem
{
    public:
        TipoItem();
        TipoItem(TipoChave c);
        void SetChave(TipoChave c);
        TipoChave GetChave();
        void Imprime();
***     bool Vazio();

    private:
        TipoChave chave;
        // outros membros
};
```

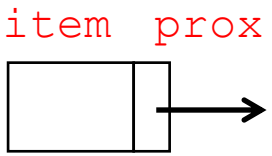
***** Método adcionado para testar se um item é vazio (chave == -1)**

```
class TipoCelula
{
    public:
        TipoCelula();

    private:
        TipoItem item;
        TipoCelula *prox;

    friend class ListaEncadeada;
};

TipoCelula::TipoCelula()
{
    item.SetChave(-1);
    prox = NULL;
}
```



Hash Listas Encadeadas - Pesquisa

```
TipoItem Hash_LE::Pesquisa(TipoChave chave){
    int pos;
    TipoItem item;

    pos = Hash(chave);
    item = Tabela[pos].Pesquisa(chave);
    return item;
}
```

Aplica a função hash na chave, para saber em qual lista procurar

Chama o método de pesquisa da classe ListaEncadeada

Retorna o item encontrado ou um item vazio (chave == -1) se não encontrar

Hash Listas Encadeadas - Pesquisa

```
// Retorna o item encontrado ou um item vazio
// (chave == -1) se não estiver presente

TipoItem  ListaEncadeada::Pesquisa(TipoChave c) {
    TipoItem aux; // construtor seta o item para -1;
    TipoCelula *p;

    p = primeiro->prox;
    while (p!=NULL) {
        if (p->item.GetChave() == c) {
            aux = p->item;
            break;
        }
        p = p->prox;
    }
    return aux;
};
```

Inicializa p com o primeiro da lista

Enquanto p não chegar no fim da lista

Se a chave do item da lista é a procurada

aux recebe o elemento da lista, termina o loop

Se não, passa para o próximo elemento da lista

Retorna aux

Hash Listas Encadeadas - Insere

```
void Hash_LE::Insere(TipoItem item){
    TipoItem aux;
    int pos;

    aux = Pesquisa(item.GetChave());
    if(!aux.Vazio())
        throw("Erro: Item já está presente");

    pos = Hash(item.GetChave());
    Tabela[pos].InsereFinal(item);
}
```

Verifica se o item já está presente

Aplica a função hash na chave para indicar em qual lista Inserir

Insere no final da lista, chamando o InsereFinal da classe ListaEncadeada

Hash Listas Encadeadas - Insere

```
void ListaEncadeada::InsereFinal(TipoItem item) {
    TipoCelula *nova;

    nova = new TipoCelula();
    nova->item = item;
    nova->prox = NULL;
    ultimo->prox = nova;
    ultimo = nova;
    tamanho++;
};
```

Hash Listas Encadeadas - Remove

- Aplica a função hash na chave para indicar em qual lista o item vai estar
- Chama o RemoveItem da classe ListaEncadeada
  - O RemoveItem da classe ListaEncadeada gera uma exceção se o item não estiver presente

```
void Hash_LE::Remove(TipoChave chave){
    int pos;

    pos = Hash(chave);
    Tabela[pos].RemoveItem(chave);
}
```

Hash Listas Encadeadas - Remove

```
TipoItem  ListaEncadeada::RemoveItem(TipoChave c) {
    TipoItem aux; TipoCelula *p, *q;

    // Posiociona p na célula anterior ao item procurado
    p = primeiro;
    while ( (p->prox!=NULL) && (p->prox->item.GetChave() != c) )
        p = p->prox;

    // remove a célula contendo o item, retornando-o
    if(p->prox == NULL)
        throw "Erro: item não está presente";
    else {
        q = p->prox;
        p->prox = q->prox;
        aux = q->item;
        delete q;
        tamanho--;
        if(p->prox == NULL) ultimo = p;
    }
    return aux;
};
```

Procura o elemento a ser retirado e p aponta para o anterior

Se o elemento não está na lista, lança uma exceção

q é o elemento a ser retirado

retira q da lista

aux recebe o elemento sendo removido

chama o destrutor para q

atualiza os campos da lista: tamanho e último

Encadeamento: Análise

- Tamanho esperado de cada lista: N/M
  - Assumindo que qualquer item do conjunto tem igual probabilidade endereçado para qualquer entrada de T.
  - N: número de registros, M: tamanho da tabela

- Operações Pesquisa, InsereHashing e RetiraHashing: O(1 + N/M)
  - 1: tempo para encontrar a entrada na tabela
  - N/M: tempo para percorrer a lista

- M ~ N, tempo se torna constante.

RESOLUÇÃO DE COLISÕES – ENDEREÇAMENTO ABERTO

Resolução de Colisões: Endereçamento Aberto

- Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, não há necessidade de se utilizar apontadores.

- **Endereçamento aberto**: chaves são armazenadas na própria tabela.

- Para tabela com tamanho M (M > N), pode-se utilizar os espaços vazios da própria tabela para resolver as colisões.

Resolução de Colisões: Endereçamento Aberto

- Quando encontra uma colisão, procura localizações alternativas (h_j).

- **Hashing linear**

$$h_j = (h(x) + j) \bmod M, \quad \text{para } 1 \leq j \leq M-1$$

- Hashing quadrático

$$h_j = (h(x) + j^2) \bmod M$$

Endereçamento Aberto: Exemplo

- Suponha que a i-ésima letra do alfabeto é representada pelo número i e a função de transformação abaixo é utilizada:
  - h(Chave) = Chave mod M

- O resultado da inserção das chaves L U N E S na tabela, usando hashing linear (j = 1) para resolver colisões é mostrado a seguir.
  - Considere M = 7

Endereçamento Aberto: Exemplo

```
h(L) = h(12) = 5
h(U) = h(21) = 0
h(N) = h(14) = 0
h(E) = h(5)  = 5
h(S) = h(19) = 5
```

	T
0	U
1	N
2	S
3	
4	
5	L
6	E

Endereçamento Aberto

- Pesquisa

	T
0	U
1	N
2	S
3	
4	
5	L
6	E

Procura **S**

h(S) = 19 = 5

OK!

Retira **N**

Procura **S** **???**

Endereçamento Aberto

- O que fazer quando um elemento é retirado?
  - Possíveis problemas na busca sequencial após a colisão…

- Solução: Diferenciar o status da posição: "**Vazia**" x "**Retirada**"
- Posição com status de "**Retirada**"
  - Para a pesquisa, posição é considerada ocupada
  - Para a inserção, posição é considerada vazia

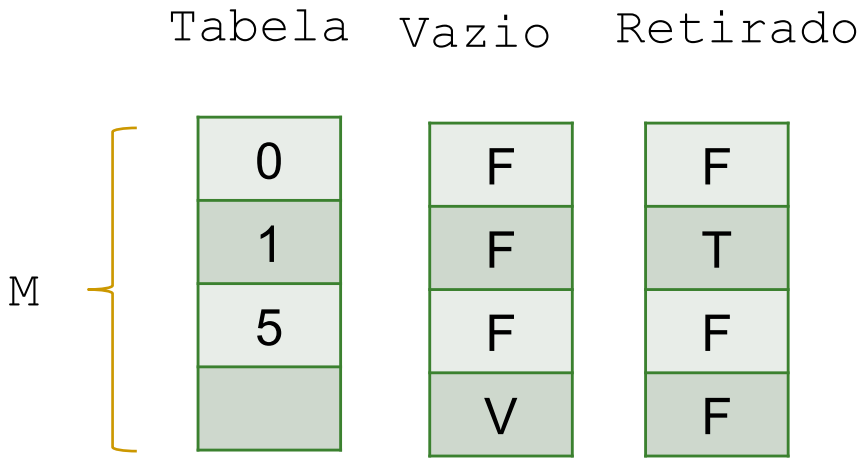
Endereçamento Aberto: Implementação

```
class Hash_EA
{
  public:
    Hash_EA();

    TipoItem Pesquisa(TipoChave chave);
    void Insere(TipoItem item);
    void Remove(TipoChave chave);

  private:
    static const int M = 7;
    int Hash(TipoChave Chave);
    TipoItem Tabela[M];
    bool vazio[M];
    bool retirado[M];
};
```

Vetores de Flags para indicar posições vazias e retiradas



Endereçamento Aberto: Implementação

```
TipoItem Hash_EA::Pesquisa(TipoChave Chave) {
    TipoItem aux; // construtor seta o item para -1;
    int pos, i;

    pos = Hash(Chave);
    i = 0;
    while ( (i<M) && !vazio[(pos+i)%M] &&
            (Tabela[(pos+i)%M].GetChave() != Chave) )
        i++;

    if ( (Tabela[(pos+i)%M].GetChave() == Chave) &&
         !retirado[(pos+i)%M] ) {
       aux = Tabela[(pos+i)%M];
    }

    return aux;
}
```

Aplica a função hash na chave, para saber a sua posição na tabela

Enquanto não percorreu toda a tabela e nem achou uma posição vazia

E enquanto não achou a chave

Se achou a chave e não tem o status de retirada

Aux retorna o item procurado

Aux retorna o item

Endereçamento Aberto: Implementação

```
void Hash_EA::Insere(TipoItem item){
    TipoItem aux; // construtor seta o item para -1;
    int pos, i;

    aux = Pesquisa(item.GetChave());
    if(!aux.Vazio())
        throw("Erro: Item já está presente");

    pos = Hash(item.GetChave());
    i = 0;
    while ( (i<M) && !vazio[(pos+i)%M] && !retirado[(pos+i)%M] )
           i++;

    if(i==M)
        throw("Erro: Tabela está cheia");
    else {
        Tabela[(pos+i)%M] = item;
        vazio[(pos+i)%M] = false;
        retirado[(pos+i)%M] = false;
    }
}
```

Se o elemento já está na tabela, lança uma exceção

Aplica a função hash na chave, para saber a sua posição na tabela

Procura posição disponível para inserção

Se tabela cheia, lança uma exceção

Insere item na Tabela, na posição livre

Indica que a posição está ocupada

Endereçamento Aberto: Implementação

```
void Hash_EA::Remove(TipoChave Chave){
    int pos, i;

    pos = Hash(Chave);
    i = 0;
    while ( (i<M) && !vazio[(pos+i)%M] &&
            (Tabela[(pos+i)%M].GetChave() != Chave) )
        i++;

    if ( (Tabela[(pos+i)%M].GetChave() == Chave) &&
         !retirado[(pos+i)%M] )
        retirado[(pos+i)%M] = true;
    else
        throw("Erro: Item não está presente");
}
```

Procura posição do elemento na tabela

Se achou o elemento e ele não tem o status de retirado, retira-o.

Se o elemento não está na tabela, lança uma exceção

Endereçamento Aberto: Análise

- Tempo para Pesquisa, Inserção e Retirada
  - Melhor caso: O(1)
  - Pior caso: O (n)

- Seja α = N / M o fator de carga da tabela. Conforme demonstrado por Knuth (1973), o custo de uma pesquisa **com sucesso** é

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

C(n) é o numero de comparações

Endereçamento Aberto: Análise

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

α = N / M	C(n)
0.1000	1.0556
0.2000	1.1250
0.3000	1.2143
0.4000	1.3333
0.5000	1.5000
0.6000	1.7500
0.7000	2.1667
0.8000	3.0000
0.9000	5.5000
0.9500	10.5000
0.9800	25.5000
0.9900	50.5000

Endereçamento Aberto: Análise

- O hashing linear sofre de um mal chamado agrupamento.

- Mal do agrupamento
  - Ocorre na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contígua a outras posições já ocupadas, o que deteriora o tempo necessário para novas pesquisas.

- O caso médio é O(1).

Vantagens e Desvantagens do Hashing

- Vantagens:
  - Alta eficiência no custo de pesquisa, que é O(1) para o caso médio.
  - Simplicidade de implementação

- Desvantagens:
  - Custo para recuperar os registros na ordem lexicográfica das chaves é alto, sendo necessário ordenar o arquivo.
  - Pior caso é O(N)