

BCC202 - Estruturas de Dados I

Aula 22: Tabelas Hash

Pedro Silva

Universidade Federal de Ouro Preto, UFOP
Departamento de Computação, DECOM
Email: silvap@ufop.edu.br

2021



Abordagens de pesquisa em Memória Primária

- ▶ Pesquisa Sequencial.
- ▶ Pesquisa Binária.
- ▶ Árvores de Pesquisa:
 - ▶ Árvores Binárias de Pesquisa.
 - ▶ Árvores AVL.
- ▶ Transformação de Chave (*Hashing*):
 - ▶ Listas Encadeadas.
 - ▶ Endereçamento Aberto.
 - ▶ *Hashing* Perfeito.

Conteúdo

Introdução

Funções de Transformação

Listas Encadeadas

Análise

Endereçamento Aberto

Análise

Considerações Finais

Bibliografia

Conteúdo

Introdução

Funções de Transformação

Listas Encadeadas

Análise

Endereçamento Aberto

Análise

Considerações Finais

Bibliografia

Tabela Hash

Hash significa:

- ▶ Fazer picadinho de carne e vegetais para cozinhar.
- ▶ Fazer uma bagunça. (**Webster's New World Dictionary**)
- ▶ Espalhar x Transformar (informática x computação).

Em uma Tabela *Hash*, os registros armazenados em um vetor são diretamente endereçados a partir de uma **transformação aritmética sobre a chave de pesquisa**.

Transformação de Chave (*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.

Hashing x 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.

Colisões e o paradoxo do aniversário

- ▶ O paradoxo do aniversário (Feller, 1968, p. 33)
 - ▶ Em um grupo de 23 ou mais pessoas, existe uma chance maior do que **50%** de que 2 pessoas comemorem o aniversário no mesmo dia.
- ▶ Assim, se for utilizada uma função de transformação uniforme que enderece 23 chaves randômicas em uma tabela de tamanho 365, a probabilidade de que haja colisões é maior do que 50%.

Probabilidade de colisões

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

$$\begin{aligned} p &= \frac{M-1}{M} \times \frac{M-2}{M} \times \dots \times \frac{M-N+1}{M} \\ &= \prod_{i=1}^N \frac{M-i+1}{M} = \frac{M!}{(M-N)!M^N} \end{aligned} \quad (1)$$

Probabilidade de não ocorrer colisões

- Alguns valores da probabilidade p de não ocorrer colisões para valores de N , com $M = 365$.

N	p
10	0,883
22	0,524
23	0,493
30	0,303

- Para N pequeno a probabilidade p pode ser aproximada por $p \approx \frac{N(N-1)}{730}$.
- Por exemplo, para $N = 10$ então $p \approx 87,7\%$.

Conteúdo

Introdução

Funções de Transformação

Listas Encadeadas

Análise

Endereçamento Aberto

Análise

Considerações Finais

Bibliografia

Funções de Transformação

- ▶ Uma função de transformação deve mapear chaves em inteiros dentro do intervalo $[0, \dots, M - 1]$, 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.

Função de Transformação mais usada

- Usa o resto da divisão por M :

$$h(K) = K \% M \quad (\text{em linguagem C})$$

onde K é um inteiro correspondente à chave e M é o tamanho da tabela.

Função de Transformação mais usada - Cuidados

- Cuidado na escolha do valor de M . M deve preferencialmente ser um número primo, mas não qualquer primo: devem ser evitados os números primos obtidos a partir de

$$b * i \pm j$$

onde b é a base do conjunto de caracteres (geralmente $b = 64$ para BCD, 128 para ASCII, 256 para EBCDIC, ou 100 para alguns códigos decimais), e i e j são pequenos inteiros.

Transformações de Chaves não numéricas

- ▶ As chaves não numéricas devem ser transformadas em números:

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

onde:

- ▶ n é o número de caracteres da chave.
- ▶ $Chave[i]$ corresponde à representação ASCII do i -ésimo caractere da chave.
- ▶ $p[i]$ é um inteiro de um conjunto de pesos gerados randomicamente para $1 \leq i \leq n$.

Transformações de Chaves não numéricas - Vantagens

- ▶ Vantagem de se usar pesos:
 - ▶ Dois conjuntos diferentes de pesos $p_1[i]$ e $p_2[i]$, $1 \leq i \leq n$, levam a duas funções de transformação $h_1(K)$ e $h_2(K)$ diferentes.

Implementação da função para gerar os pesos

- Programa que gera um peso para cada caractere de uma chave constituída de n caracteres:

```
1 /* Gera valores randomicos entre 1 e 10.000 */
2 void GeraPesos(int p[], int n) {
3     int i;
4     // utilizando o tempo como semente de numeros aleatorios
5     srand(time(NULL));
6     for (i = 0; i < n; i++) {
7         p[i] = 1 + (int) (10000.0 * rand() / RAND_MAX);
8     }
9 }
```

Implementação da função de transformação

```
1  /* Funcao de hash que retorna o indice (numero inteiro)
2   * de uma chave (string) */
3  int h(char *chave, int p[], int m, int tam_p) {
4      int i;
5      unsigned int soma = 0;
6      int comp = strlen(chave);
7
8      for (i = 0; i < comp; i++) {
9          soma += (unsigned int) chave[i] * p[i % tam_p];
10     }
11
12     return (soma % m);
13 }
```

Resolvendo Colisões

- ▶ Listas Encadeadas.
- ▶ Endereçamento Aberto.

Conteúdo

Introdução

Funções de Transformação

Listas Encadeadas

Análise

Endereçamento Aberto

Análise

Considerações Finais

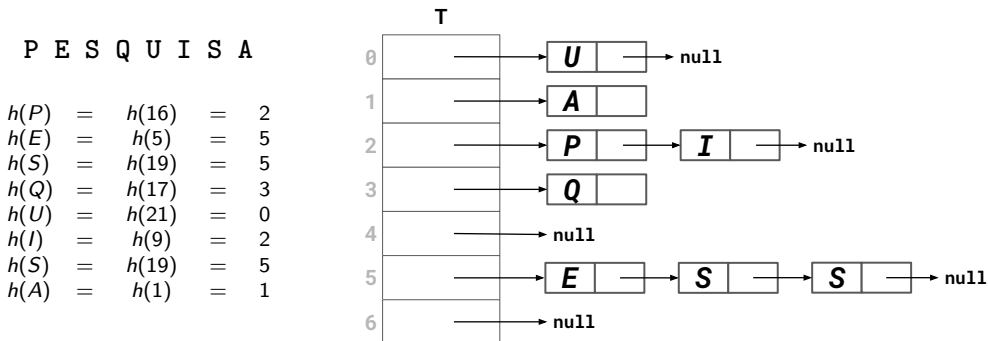
Bibliografia

Listas Encadeadas

- ▶ Uma das formas de resolver as **colisões** é simplesmente construir uma lista linear encadeada para cada endereço da tabela.
- ▶ Assim, todas as chaves com mesmo endereço são encadeadas em uma lista linear.

Listas Encadeadas - Exemplo

- Se a i -ésima letra do alfabeto é representada pelo número i e a função de transformação $h(chave) = chave \bmod M$ é utilizada para $M = 7$, o resultado da inserção das chaves abaixo na tabela é o seguinte:



TAD Lista Encadeada

```
1 #define N 16 // tamanho da chave (string)
2
3 typedef char TChave[N];
4
5 typedef struct {
6     /* outros componentes */
7     TChave chave;
8 } TItem;
9
10 typedef struct celula {
11     struct celula *pProx;
12     TItem item;
13 } TCelula;
14
15 typedef struct {
16     TCelula *pPrimeiro, *pUltimo;
17 } TLista;
```

TAD Hash

```
1 #include "lista.h"
2
3 typedef struct {
4     int n; // numero de itens na hash
5     int nro_listas; // tamanho do array de listas
6     int nro_pesos; // tamanho do array de pesos
7     int *p; // array de pesos
8     TLista *v; // array de listas
9 } THash;
10
11 void THash_Inicia(THash *hash, int nro_listas, int nro_pesos);
12 int THash_H(THash *hash, TChave chave);
13 int THash_Pesquisa(THash *hash, TChave chave, TItem *x);
14 TCelula *THash_PesquisaCelula(THash *hash, TChave chave);
15 int THash_Insere(THash *hash, TItem x);
16 int THash_Remove(THash *hash, TItem *x);
```


Função *Hash* para Transformação de uma chave

```
1 /* Função de hash que retorna o índice (número inteiro)
2  * de uma chave (string) */
3 int THash_H(THash *hash, TChave chave) {
4     int i;
5     unsigned int soma = 0;
6     int comp = strlen(chave);
7
8     for (i = 0; i < comp; i++)
9         soma += (unsigned int) chave[i] * hash->p[i % hash->nro_pesos];
10
11     return (soma % hash->nro_listas);
12 }
```

Pesquisa na Lista Encadeada de uma Hash

```
1 /* Retorno o ponteiro apontando para a celula anterior da lista */
2 T Celula *THash_PesquisaCelula(THash *hash, TChave chave) {
3     int i = THash_H(hash, chave);
4     T Celula *aux;
5
6     if (TLista_EhVazia(&hash->v[i]))
7         return NULL; // pesquisa sem sucesso
8
9     aux = hash->v[i].pPrimeiro;
10    while (aux->pProx->pProx != NULL &&
11           strcmp(chave, aux->pProx->item.chave) != 0)
12        aux = aux->pProx;
13
14    if (!strcmp(chave, aux->pProx->item.chave, sizeof(TChave)))
15        return aux;
16    else
17        return NULL; // pesquisa sem sucesso
18 }
```

Pesquisa em uma Hash

```
1 /* Retorna se a pesquisa foi bem sucedida e o item (x) por meio
2  * de passagem por referência */
3 int THash_Pesquisa(THash *hash, TChave chave, TItem *x) {
4     TCelula *aux = THash_PesquisaCelula(hash, chave);
5     if (aux == NULL) {
6         return 0;
7     }
8     *x = aux->pProx->item;
9     return 1;
10 }
```

Remoção e inserção de um item em uma Hash

```
1 int THash_Inserere(THash *hash, TItem x) {
2     if (THash_PesquisaCelula(hash, x.chave) == NULL) {
3         TLista_Inserere(&hash->v[THash_H(hash, x.chave)], x);
4         hash->n++;
5         return 1;
6     }
7     return 0;
8 }
9
10 int THash_Remove(THash *hash, TItem *x) {
11     TCelula *aux = THash_PesquisaCelula(hash, x->chave);
12
13     if (aux == NULL)
14         return 0;
15     TLista_Remove(&hash->v[THash_H(hash, x->chave)], aux, x);
16     hash->n--;
17     return 1;
18 }
```

Análise de *Hashs* implementadas com Listas Encadeadas

- ▶ Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada de T , então o comprimento esperado de cada lista encadeada é N/M , onde:
 - ▶ N representa o número de registros na tabela.
 - ▶ M representa o tamanho da tabela.
- ▶ Logo: as operações **Pesquisa**, **Inserir** e **Retira** custam $O(1 + N/M)$ operações em média, onde a constante 1 representa o tempo para encontrar a entrada na tabela e N/M o tempo para percorrer a lista.
- ▶ Se os valores forem bem distribuídos (poucas colisões) e N for igual a M , teremos que:
 - ▶ Pesquisa, inserção e remoção serão $O(1)$.

Conteúdo

Introdução

Funções de Transformação

Listas Encadeadas

Análise

Endereçamento Aberto

Análise

Considerações Finais

Bibliografia

Endereçamento Aberto

- ▶ Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, então não haverá necessidade de usar apontadores para armazenar os registros.
- ▶ Existem vários métodos para armazenar N registros em uma tabela de tamanho $M > N$, os quais utilizam os lugares vazios na própria tabela para resolver as colisões. (Knuth, 1973, p.518)

Endereçamento Aberto

- ▶ No **Endereçamento Aberto** todas as chaves são armazenadas na própria tabela, sem o uso de ponteiros explícitos.
- ▶ Existem várias propostas para a escolha de localizações alternativas. A mais simples é chamada de *hashing* linear, onde a posição h_j na tabela é dada por:

$$h_j = (h(x) + j) \text{ mod } M, \text{ para } 0 \leq j \leq M - 1.$$

Inserção em uma *Hash* de Endereçamento Aberto

- ▶ Se a i -ésima letra do alfabeto é representada pelo número i e a função de transformação

$$h(chave) = chave \bmod M$$

é utilizada para $M = 7$, teremos:

- ▶ $h(L) = h(12) = 5$
- ▶ $h(U) = h(21) = 0$
- ▶ $h(N) = h(14) = 0$
- ▶ $h(E) = h(5) = 5$
- ▶ $h(S) = h(19) = 5$

Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

T	
0	
1	
2	
3	
4	
5	
6	

Exemplo

Inserção de:

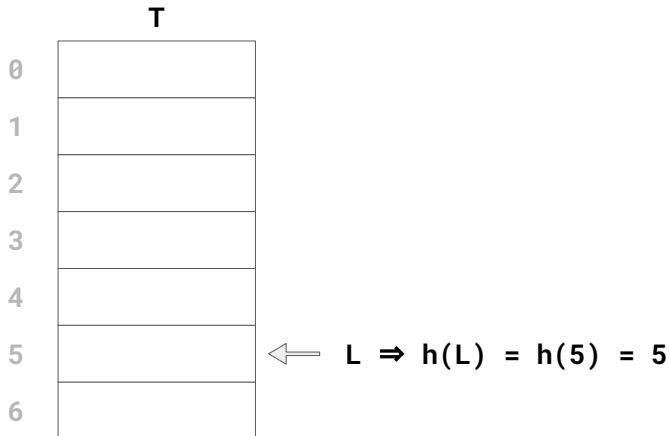
► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$



Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

T	
0	
1	
2	
3	
4	
5	L
6	

Exemplo

Inserção de:

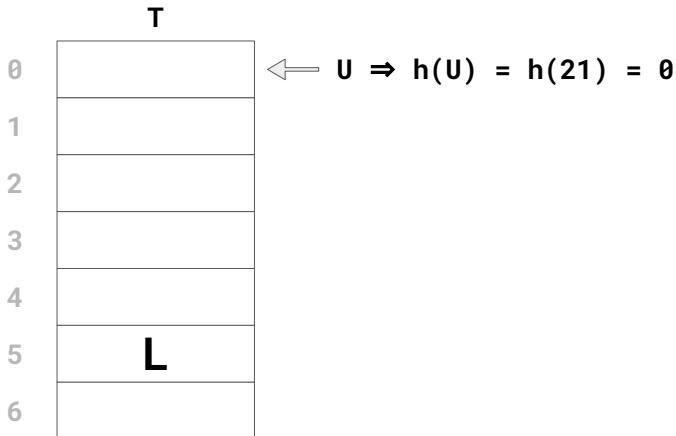
► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$



Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

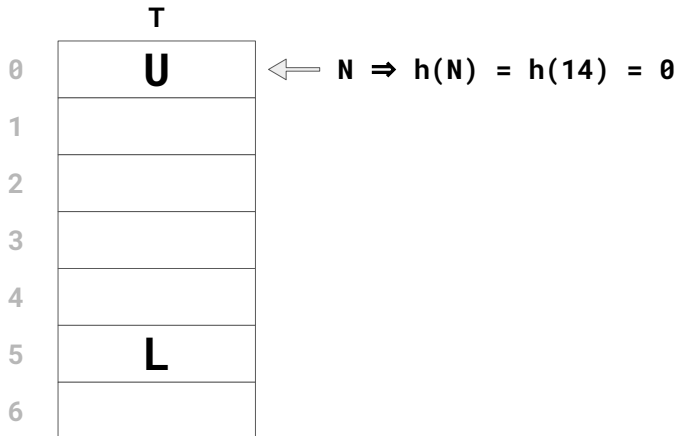
► $h(S) = h(19) = 5$

T	
0	U
1	
2	
3	
4	
5	L
6	

Exemplo

Inserção de:

- ▶ $h(L) = h(12) = 5$
- ▶ $h(U) = h(21) = 0$
- ▶ $h(N) = h(14) = 0$
- ▶ $h(E) = h(5) = 5$
- ▶ $h(S) = h(19) = 5$



Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

T	
0	U
1	
2	
3	
4	
5	L
6	

← $N \Rightarrow h(N) = h(14) = 0$

Exemplo

Inserção de:

- ▶ $h(L) = h(12) = 5$
- ▶ $h(U) = h(21) = 0$
- ▶ $h(N) = h(14) = 0$
- ▶ $h(E) = h(5) = 5$
- ▶ $h(S) = h(19) = 5$

T	
0	U
1	N
2	
3	
4	
5	L
6	

Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

T	
0	U
1	N
2	
3	
4	
5	L
6	

← $E \Rightarrow h(E) = h(5) = 5$

Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

T	
0	U
1	N
2	
3	
4	
5	L
6	



$E \Rightarrow h(E) = h(5) = 5$

Exemplo

Inserção de:

- ▶ $h(L) = h(12) = 5$
- ▶ $h(U) = h(21) = 0$
- ▶ $h(N) = h(14) = 0$
- ▶ $h(E) = h(5) = 5$
- ▶ $h(S) = h(19) = 5$

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

Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

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

← $S \Rightarrow h(S) = h(19) = 5$

Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

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

← $S \Rightarrow h(S) = h(19) = 5$

Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

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

← $S \Rightarrow h(S) = h(19) = 5$

Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

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

← S ⇒ $h(S) = h(19) = 5$

Exemplo

Inserção de:

► $h(L) = h(12) = 5$

► $h(U) = h(21) = 0$

► $h(N) = h(14) = 0$

► $h(E) = h(5) = 5$

► $h(S) = h(19) = 5$

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

← S ⇒ $h(S) = h(19) = 5$

Dicionário usando Endereçamento Aberto

```
1 | #define VAZIO "!!!!!!!!!!\0"
2 | #define N 10 // tamanho da chave (string)
3 | #define M 100 // tamanho da tabela
4 |
5 | typedef char TChave[N];
6 |
7 | typedef struct {
8 |     /* outros componentes */
9 |     TChave Chave;
10 | } TItem;
11 |
12 | typedef TItem TDicionario[M];
```

Iniciando um Dicionário de Endereçamento Aberto

```
1 void TDicionario_Inicia(TDicionario dic) {  
2     int i;  
3     for (i = 0; i < M; i++) {  
4         memcpy(dic[i].chave, VAZIO, N);  
5     }  
6 }
```

Pesquisando em um Dicionário de Endereçamento Aberto

```
1 int TDiccionario_Pesquisa(TDiccionario dic, TChave chave, int p) {
2     int j = 0;
3     int ini = h(chave, p);
4
5     while (strcmp(dic[(ini + j) % M].chave, VAZIO) != 0 &&
6            strcmp(dic[(ini + j) % M].chave, chave) != 0 &&
7            j < M) {
8         j++;
9     }
10
11     if (strcmp(dic[(ini + j) % M].chave, chave) == 0) {
12         return (ini + j) % M;
13     }
14     return -1; // pesquisa sem sucesso
15 }
```

Inserção de um item no Dicionário de Endereçamento Aberto

```
1 int TDiccionario_Inserere(TDiccionario dic, TItem x, int p) {
2     if (TDiccionario_Pesquisa(dic, x.chave, p) >= 0)
3         return 0; // chave já existe no dicionário
4
5     int j = 0;
6     int ini = h(x.chave, p);
7     while (strcmp(dic[(ini + j) % M].chave, VAZIO) != 0 && j < M) {
8         j++;
9     }
10
11     if (j < M) {
12         dic[(ini + j) % M] = x;
13         return 1;
14     }
15     return 0;
16 }
```

Remoção de um item do Dicionário de Endereçamento Aberto

```
1 int TDicionario_Retira(TDicionario dic, TItem *x, int p) {  
2     int i = TDicionario_Pesquisa(dic, x->chave, p);  
3     if (i == -1) {  
4         return 0; // chave não encontrada  
5     }  
6  
7     memcpy(dic[i].chave, VAZIO, N);  
8     return 1;  
9 }
```

Análise de *Hashs* implementadas com Endereçamento Aberto

- ▶ Seja $\alpha = N/M$ o fator de carga da tabela. Conforme demonstrado por Knuth (1973), o custo de uma pesquisa com sucesso é:

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

- ▶ O *hashing* linear sofre de um mal chamado agrupamento (*clustering*) (Knuth, 1973, pp.520–521).
 - ▶ Este fenômeno 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.

Análise de *Hashs* implementadas com Endereçamento Aberto

- ▶ Apesar do *hashing* linear ser um método relativamente pobre para resolver colisões os resultados apresentados são bons.
- ▶ O melhor caso, assim como o caso médio, é $O(1)$.
- ▶ Qual o pior caso?
 - ▶ Quando é necessário percorrer toda a tabela para encontrar uma chave (excesso de colisões).
 - ▶ Complexidade do pior caso: $O(n)$.

Vantagens e Desvantagens

Vantagens:

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

Desvantagens:

- ▶ Custo para recuperar os registros ordenados pela chave é alto, sendo necessário ordenar toda a tabela.
- ▶ Pior caso é $O(n)$.

Conteúdo

Introdução

Funções de Transformação

Listas Encadeadas

Análise

Endereçamento Aberto

Análise

Considerações Finais

Bibliografia

Conclusão

- ▶ Duas implementações para *Hash*: Lista encadeada e Endereçamento Aberto.
- ▶ São armazenados em Tabelas *Hash* uma coleção de registros com chaves.
- ▶ A posição de inserção de um novo registro depende do valor resultante da função *Hash*.
- ▶ A escolha do que fazer quando há uma colisão, depende da estratégia de implementação.
- ▶ Geralmente, procurar por uma chave é rápido.

Conteúdo

Introdução

Funções de Transformação

Listas Encadeadas

Análise

Endereçamento Aberto


Análise

Considerações Finais

Bibliografia

Bibliografia

Os conteúdos deste material, incluindo figuras, textos e códigos, foram extraídos ou adaptados de:

-  Cormen, Thomas H. and Leiserson, Charles E. and Rivest, Ronald L. and Stein, Clifford.
Introduction to Algorithms.
The MIT Press, 2011.

Exercício

Mostre o passo-a-passo de como seria a inserção da chave abaixo na tabela ao lado.

$$P = h(16)$$

Considere que a função *Hash*:

$$h(chave) = chave \bmod M$$

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

<