



Hash

Professor: Elton Sarmanho¹

E-mail: eltonss@ufpa.br



¹Faculdade de Sistemas de Informação - UFPA/CUTINS

7 de outubro de 2025

Roteiro

Hash

- Introdução

- Características

- Problemas

Funções de Dispersão

- Propriedades

- Tipos de Funções



Roteiro

Tratamento de Colisões

- Fator de Carga

- Encadeamento

- Endereçamento Aberto

- Hash Perfeito

Referências Bibliográficas



- ▶ Os métodos de busca linear ou binária baseiam-se na comparação da chave de busca com as chaves dos registros armazenados na tabela.



Busca Sequencial

Melhor Caso $O(1)$

Pior Caso $O(n)$

Case 1: Procura número 12

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

Case 2: Procura número 40

12	5	10	15	31	20	25	2	40
0	1	2	3	4	5	6	7	8

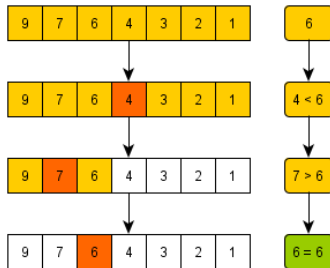


Busca binária

Chave é comparada com registro que se encontra no meio do array ordenado

Melhor Caso $O(1)$

Pior Caso $O(\log n)$



- ▶ A busca pode ser mais rápida se existir uma relação direta entre a posição em que o registro se encontra na tabela e a sua chave a ser localizada.
- ▶ Organizamos os dados em uma nova estrutura de dados chamada **Tabela Hash**
 - ▶ Média por operação de $\theta(1)$, sendo o pior caso, entretanto, $O(n)$



└ Hash

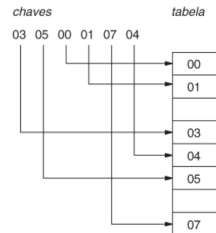
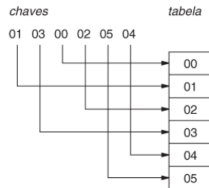
└ Características

- ▶ Os registros são armazenados em uma tabela T , sequencial e de dimensão m .
- ▶ As posições da tabela que se situam no intervalo $[0, m-1]$ são calculadas através de sua chave por meio de uma função de dispersão (*Hash Function*)
 - ▶ $h : U \rightarrow \{0, 1, \dots, m-1\}$
 - ▶ Universo das chaves U
 - ▶ Conjunto dos endereços $\{0, 1, \dots, m-1\}$
- ▶ h mapeia o universo U de chaves nos endereços de uma tabela T chamada de **tabela hash** (*hash table*)



Técnica de Acesso Direto

- ▶ número de chaves n
- ▶ número de endereços da tabela m
- ▶ $n \leq m$
- ▶ Todas operações $O(1)$

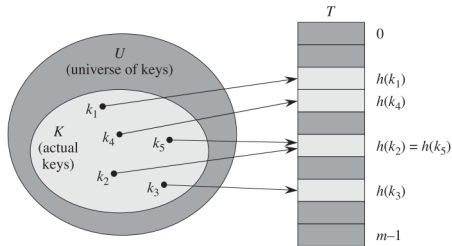


- ▶ Desperdício de espaço.
 - ▶ Exemplo: Se existem duas chaves com índices 0 e 999.999
 - ▶ A aplicação da técnica de acesso direto conduziria a uma tabela com 1.000.000 compartimentos, dos quais apenas dois ocupados
- ▶ Quando $n \geq m$
- ▶ **Solução:** aplicar $h(k) \forall k \in U$
 - ▶ A ideia é transformar cada chave k em um valor no intervalo $[0, m-1]$ usando $h(k)$
 - ▶ Dizemos que $h(k)$ é *hash value* da chave k

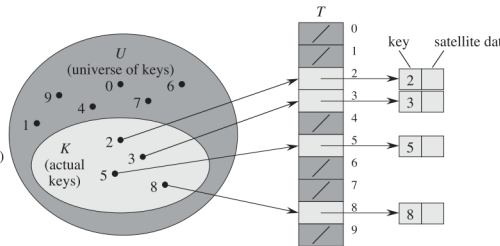


└ Hash

└ Problemas



Função Hash

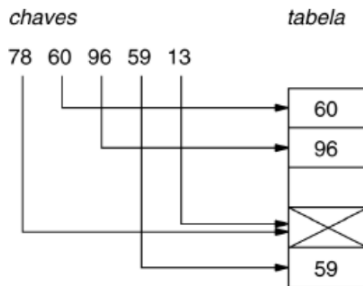


Acesso Direto



- ▶ Função de dispersão pode não garantir **injetividade**
 - ▶ É possível a existência de outra chave $y \neq x$
 - ▶ Duas chaves podem ter mesmo **valor hash** para o mesmo endereço
 - ▶ $h(x) = h(y)$
 - ▶ Fenômeno chamado de **Colisão**





Exemplo de Colisão usando $h(k) = k \bmod 5$



└ Funções de Dispersão

└ Propriedades

- ▶ Uma função de dispersão h transforma uma chave x em um endereço-base $h(x)$ em T
- ▶ Uma $h(x)$ deve satisfazer às seguintes condições:
 - ▶ produzir um número baixo de colisões;
 - ▶ Tratamento de colisões
 - ▶ ser facilmente computável;
 - ▶ ser uniforme.
 - ▶ A probabilidade de que $h(x)$ seja igual ao endereço k deve ser $\frac{1}{m}$ para todas as chaves x e todos os endereços $k \in [0, m - 1]$



Método da Divisão

- ▶ Este método é fácil e eficiente, sendo por isso muito empregado.
- ▶ A chave x é dividida pela dimensão m da tabela T , e o resto da divisão é usado como endereço.

$$h(x) = x \bmod m$$

- ▶ Resultando em endereços no intervalo $[0, m - 1]$

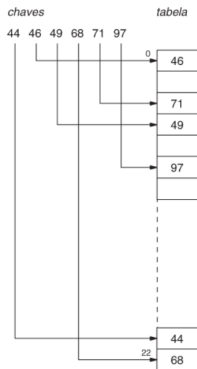


Método da Divisão

- ▶ Ao usar o método da divisão, geralmente evitamos certos valores de m . Existem critérios para escolher um melhor m .
 - ▶ Escolher m de modo que seja um número primo não próximo a uma potência de 2.



Método da Divisão



Método da Divisão.

$$m = 23$$

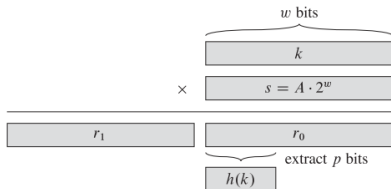
$$44 \bmod 23 = 21$$



Método da Multiplicação

- ▶ Endereço é calculado em duas etapas.
- ▶ A chave k é multiplicada por uma constante $A \in]0, 1[$. O resultado é armazenado em s . Extrai a parte fracionada de kA
- ▶ Multiplica parte fracionada por m e computa $\text{floor} \lfloor \cdot \rfloor$.

$$h(k) = \lfloor m (kA \bmod 1) \rfloor$$



Método da Multiplicação

- ▶ Vantagem desse método é que valor de m não é crítico
- ▶ Embora esse método funcione com qualquer valor da constante A , ele funciona melhor com alguns valores do que com outros.
 - ▶ A escolha ideal depende das características dos dados que estão sendo armazenados
 - ▶ Knuth sugere:

$$A \approx \frac{(\sqrt{5} - 1)}{2} = 0.6180\dots$$



Exercício A - Método de Divisão

- Considere uma Tabela hash de tamanho $m = 8$. Calcule as localizações para as quais são mapeadas as chaves $[16, 23, 41, 25]$.



Exercício A.1 - Método de Divisão

- ▶ Desenvolva a tabela hash com 9 slots inserindo [5, 28, 19, 15, 20, 33, 12, 17, 10].



Implementação em C - Método da Divisão

```
#include <stdio.h>
#define TABLE_SIZE 8
#include <math.h>
void initialize_table(int table[], int size) {
    for (int i = 0; i < size; i++) {
        table[i] = -1; // Valor inicial para indicar posicao vazia
    }
}

void insere(int table[], int key, char funcao) {
    if(funcao == 'D') { //Metodo da Divisao
        int hash = key % TABLE_SIZE;
        table[hash] = key;
    }
}
```



Implementação em C - Método da Divisão

```
//Continuacao
int main() {
    int hash_table[TABLE_SIZE];
    initialize_table(hash_table, TABLE_SIZE);
    int keys[] = {16,23,41,25};
    int num_keys = sizeof(keys) / sizeof(keys[0]);

    for (int i = 0; i < num_keys; i++) {
        int key = keys[i];
        insere(hash_table, key, 'D');
    }
    // Imprimir tabela hash
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("Index: %d, Key: %d\n", i, hash_table[i]);
    }
    return 0;
}
```



Implementação em Python - Método da Divisão

```
import math
import numpy as np

class Hashing:
    def __init__(self,funcao):
        self.funcao = funcao
        pass;
```



Implementação em Python - Método da Divisão

```
def hashingDivisao(self, chave, hashTable):  
    m = len(hashTable)  
    return chave % m;  
  
# Funcao add valor na Tabela hash  
def insere(self, hashtable, valor_chave, valor):  
    if(self.funcao == "Divisao"):  
        endereco = self.hashingDivisao(valor_chave, hashtable)  
  
        hashtable[endereco]= valor
```



└ Funções de Dispersão

└ Tipos de Funções

```
if __name__ == '__main__':  
    HashTable = np.array([None]*8)  
    hashing = Hashing(funcao="Divisao")  
    hashing.insere(HashTable, 16, '16')  
    hashing.insere(HashTable, 23, '23')  
    hashing.insere(HashTable, 41, '41')  
    hashing.insere(HashTable, 25, '25')  
    hashing.display_hash(HashTable)
```

Resultado

```
0 -- > 16  
1 -- > 25  
2 -- > None  
3 -- > None  
4 -- > None  
5 -- > None  
6 -- > None  
7 -- > 23
```



Exercício B - Método da Multiplicação

- Considere uma Tabela hash de tamanho $m = 1000$ e uma função hash correspondente $h(k)$ igual a $m (kA \bmod 1)$ para $A \approx \frac{(\sqrt{5} - 1)}{2}$. Calcule as localizações para as quais são mapeadas as chaves 61, 62, 63, 64 e 65.



Exercício B - Método da Multiplicação

▶ $m = 1000$

▶ $k = [61]$

▶ $A = \frac{(\sqrt{5} - 1)}{2}$

$$h(k) = \lfloor m (kA \bmod 1) \rfloor$$



Exercício B - Método da Multiplicação

▶ $m = 1000$

▶ $k = [61]$

▶ $A = \frac{(\sqrt{5} - 1)}{2}$

$$h(k) = \lfloor m \left(k \frac{(\sqrt{5} - 1)}{2} \bmod 1 \right) \rfloor$$



Exercício B - Método da Multiplicação

▶ $m = 1000$

▶ $k = [61]$

▶ $A = \frac{(\sqrt{5} - 1)}{2}$

$$h(61) = \lfloor m \left(61 \frac{(\sqrt{5} - 1)}{2} \bmod 1 \right) \rfloor$$



Exercício B - Método da Multiplicação

▶ $m = 1000$

▶ $k = [61]$

▶ $A = \frac{(\sqrt{5} - 1)}{2}$

$$h(61) = \lfloor m (37,700073314 \bmod 1) \rfloor$$



Exercício B - Método da Multiplicação

▶ $m = 1000$

▶ $k = [61]$

▶ $A = \frac{(\sqrt{5} - 1)}{2}$

$$h(61) = \lfloor m (0,700073314) \rfloor$$



Exercício B - Método da Multiplicação

▶ $m = 1000$

▶ $k = [61]$

▶ $A = \frac{(\sqrt{5} - 1)}{2}$

$$h(61) = \lfloor 1000 (0,700073314) \rfloor$$



Exercício B - Método da Multiplicação

▶ $m = 1000$

▶ $k = [61]$

▶ $A = \frac{(\sqrt{5} - 1)}{2}$

$$h(61) = \lfloor 700,073314 \rfloor$$



Exercício B - Método da Multiplicação

▶ $m = 1000$

▶ $k = [61]$

▶ $A = \frac{(\sqrt{5} - 1)}{2}$

$$h(61) = \lfloor 700,073314 \rfloor$$



Exercício B - Método da Multiplicação

▶ $m = 1000$

▶ $k = [61]$

▶ $A = \frac{(\sqrt{5} - 1)}{2}$

$$h(61) = 700$$



Implementação em C - Método da Multiplicação

```
//Alteramos somente metodo insere
void insere(int table[], int key, char funcao) {
    if(funcao == 'D'){
        int hash = key % TABLE_SIZE;
        table[hash] = key;
    }
    else if(funcao == 'M')
    {
        float A = (sqrt(5)-1)/2;
        float m = TABLE_SIZE;

        int hash = floor(m*(fmod(key*A , 1)));
        table[hash] = key;
    }
}
```



Implementação em C - Método da Multiplicação

```
//Caso esteja usando linux sera preciso implementar  
//metodo floor e fmod (biblioteca math.h)
```

```
double floor(double x) {  
    int int_part = (int)x;  
    return (double)int_part;  
}
```

```
double fmod(double x, double y) {  
    if (y == 0.0) {  
        // Tratar divisao por zero  
        return 0.0;  
    }
```

```
  
    double quotient = x / y;  
    double whole_part = (double)((int)quotient);  
    double remainder = x - (whole_part * y);  
    return remainder;  
}
```



Implementação em C - Método da Multiplicação

```
#define TABLE_SIZE 1000
.
int main() {
    int hash_table[TABLE_SIZE];
    initialize_table(hash_table, TABLE_SIZE);
    int keys[] = {61,62,63,64,65};
    int num_keys = sizeof(keys) / sizeof(keys[0]);
    for (int i = 0; i < num_keys; i++) {
        int key = keys[i];
        insere(hash_table, key, 'M');
    }
    for (int i = 0; i < TABLE_SIZE; i++) { // Imprimir tabela
        printf("Index: %d, Key: %d\n", i, hash_table[i]);
    }
    printf("Index: %d, Key: %d\n", 700, hash_table[700]);
    return 0;
}
```



Implementação em Python - Método da Multiplicação

```
def hashingMultiplicacao(self, chave, hashTable):  
    A = (math.sqrt(5)-1)/2  
    m = len(hashTable)  
    h = math.floor(m*(chave*A % 1))  
    return h;  
  
def insere(self, hashtable, valor_chave, valor):  
    if(self.funcao == "Divisao"):  
        endereco = self.hashingDivisao(valor_chave, hashtable)  
    elif (self.funcao == "Multiplicacao"):  
        endereco = self.hashingMultiplicacao(valor_chave,  
                                                hashtable)  
    hashtable[endereco]=valor
```



Implementação em Python - Método da Multiplicação

```
if __name__ == '__main__':  
    HashTable = np.array([None] * 1000)  
    hashing = Hashing(funcao="Multiplicacao")  
    # Add os elementos na Tabela  
    hashing.insere(HashTable, 61, '61',)  
    hashing.insere(HashTable, 62, '62',)  
    hashing.insere(HashTable, 63, '63', )  
    hashing.insere(HashTable, 64, '64')  
    hashing.insere(HashTable, 65, '65')  
    hashing.display_hash(HashTable)
```

Resultado

700 -- > 61



└ Tratamento de Colisões

└ Fator de Carga

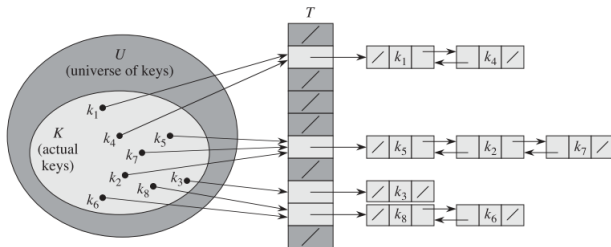
- ▶ Já foi observado que o mesmo $h(x) = h(y)$ endereço pode ser encontrado para chaves diferentes $x \neq y$
 - ▶ Colisão
- ▶ Dado uma tabela **T** com **m** endereços e que armazena **n** elementos definimos **fator de carga** (*load factor*):
 - ▶ *número médio de elementos armazenados em uma entrada*
 - ▶ $\alpha = \frac{n}{m}$, onde $\alpha < 1$, $\alpha = 1$ e $\alpha > 1$
 - ▶ $\uparrow \alpha \sim \uparrow$ Colisões
- ▶ **Encadeamento**



└ Tratamento de Colisões

└ Encadeamento

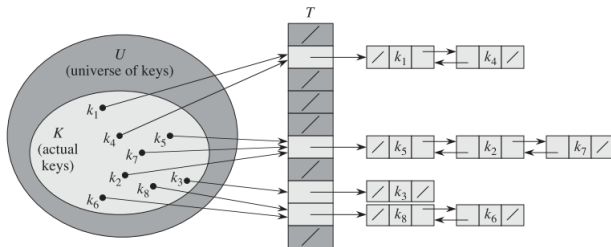
- Colocamos todos os elementos que efetuam hash para o mesmo endereço em uma **lista ligada**.



└ Tratamento de Colisões

└ Encadeamento

- ▶ Endereço j contém um ponteiro para início da lista.
- ▶ Se caso estiver vazio, j conterá nullo.



Inserção

Pior Caso $O(1)$

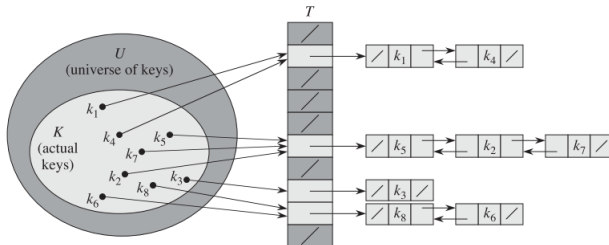
Remoção

Pior Caso $O(1)$

- ▶ Lista Duplamente Encadeada
- ▶ Método recebe como entrada um elemento X e não sua chave k

$X.ant.prox = X.prox$

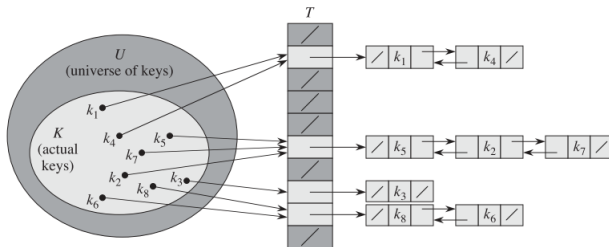
$X.prox.ant = X.ant$



Pesquisa

Pior Caso $\Theta(n)$

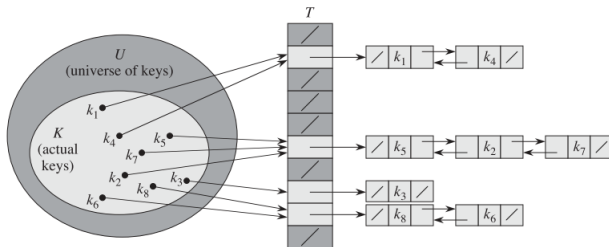
- As n chaves no mesmo endereço, criando uma lista de dimensão d



Pesquisa

Sucesso $\Theta(1 + \alpha)$ Sem Sucesso $\Theta(1 + \alpha)$

► Hash uniforme



Exercício C - Encadeamento

- ▶ Demostre a inserção das chaves **5,28,19,15,20,33,12,17,10** em uma tabela hash com colisões por encadeamento. Seja tabela com 9 posições e com função $h(k) = k \bmod 9$



Exercício C - Encadeamento

► **5,28,19,15,20,33,12,17,10**

► $h(k) = k \bmod 9$

$h(k)$	k



Exercício C - Encadeamento

▶ **28,19,15,20,33,12,17,10**

▶ $h(k) = 5 \bmod 9$

$h(k)$	k
5	5



Exercício C - Encadeamento

▶ **19,15,20,33,12,17,10**

▶ $h(k) = 28 \bmod 9$

$h(k)$	k
1	28
5	5



Exercício C - Encadeamento

▶ **15,20,33,12,17,10**

▶ **$h(k) = 19 \bmod 9$**

$h(k)$	k
1	19 → 28
5	5



Exercício C - Encadeamento

▶ **20,33,12,17,10**

▶ $h(k) = 15 \bmod 9$

$h(k)$	k
1	19 → 28
5	5
6	15



Exercício C - Encadeamento

▶ **33,12,17,10**

▶ $h(k) = 20 \bmod 9$

$h(k)$	k
1	19 → 28
2	20
5	5
6	15



Exercício C - Encadeamento

▶ **12,17,10**

▶ $h(k) = 33 \bmod 9$

$h(k)$	k
1	19 → 28
2	20
5	5
6	33 → 15



Exercício C - Encadeamento

▶ **17,10**

▶ $h(k) = 12 \bmod 9$

h(k)	k
1	19 → 28
2	20
3	12
5	5
6	33 → 15



Exercício C - Encadeamento

▶ 10

▶ $h(k) = 17 \bmod 9$

$h(k)$	k
1	19 → 28
2	20
3	12
5	5
6	33 → 15
8	17



Exercício C - Encadeamento

▶ \emptyset

▶ $h(k) = 10 \bmod 9$

$h(k)$	k
1	10 \rightarrow 19 \rightarrow 28
2	20
3	12
5	5
6	33 \rightarrow 15
8	17



Exercício C - Aluno - Encadeamento

- ▶ Forneça o conteúdo da *hash table* resultante quando você insere itens com as chaves ATLAS nessa ordem em uma tabela inicialmente vazia de $M = 5$, usando encadeamento com listas não ordenadas. Use a função hash $11k \bmod M$ para transformar a k -ésima letra do alfabeto em um índice de tabela.
- ▶ $A = 1, B = 2, \dots$



Implementação em C - Encadeamento

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

// Estrutura para o Nodo ou Elemento da lista encadeada
typedef struct Element {
    int key;
    struct Element* next;
} Element;

// Estrutura para a tabela de hash
typedef struct HashTable {
    Element** table; //Ponteiro para Ponteiro
} HashTable;
```



Implementação em C - Encadeamento

```
// Funcao para criar um novo elemento
Element* createElement(int key) {
    Element* newElement = (Element*)malloc(sizeof(Element));
    newElement->key = key;
    newElement->next = NULL;
    return newElement;
}

// Funcao para criar a tabela de hash
HashTable* createHashTable() {
    HashTable* hashTable = (HashTable*)malloc(sizeof(HashTable));
    hashTable->table = (Element**)malloc(SIZE * sizeof(Element*));
    // Inicializar cada posicao da tabela com NULL
    for (int i = 0; i < SIZE; i++) {
        hashTable->table[i] = NULL;
    }
    return hashTable;
}
```



Implementação em C - Encadeamento

```
// Funcao hash  
int hashFunction(int key) {  
    return key % SIZE;  
}
```



Implementação em C - Encadeamento

```
void insert(HashTable* hashTable, int key) {  
    int index = hashFunction(key);  
    // Criar um novo elemento ou nodo  
    Element* newElement = createElement(key);  
    // Inserir o elemento na posicao correspondente da tabela  
    if (hashTable->table[index] == NULL) {  
        // Caso a posicao esteja vazia, o novo elemento se torna  
        // a cabeca da lista  
        hashTable->table[index] = newElement;  
    } else { // Senao, add o novo elemento ao final da lista  
        Element* currentElement = hashTable->table[index];  
        while (currentElement->next != NULL) {  
            currentElement = currentElement->next;  
        }  
        currentElement->next = newElement;  
    }  
}
```



Implementação em C - Encadeamento

```
// Funcao para imprimir a tabela de hash
void printHashTable(HashTable* hashTable) {
    for (int i = 0; i < SIZE; i++) {
        printf("Posicao %d: ", i);
        Element* currentElement = hashTable->table[i];
        while (currentElement != NULL) {
            printf("%d ", currentElement->key);
            currentElement = currentElement->next;
        }
        printf("\n");
    }
}
```



Implementação em C - Encadeamento

```
int main() {  
    // Criar a tabela de hash  
    HashTable* hashTable = createHashTable();  
  
    // Inserir elementos na tabela de hash  
    insert(hashTable, 7);  
    insert(hashTable, 12);  
    insert(hashTable, 23);  
    insert(hashTable, 35);  
    insert(hashTable, 14);  
    insert(hashTable, 13);  
    insert(hashTable, 33);  
    insert(hashTable, 40);  
    insert(hashTable, 12);  
  
    // Imprimir a tabela de hash  
    printHashTable(hashTable);  
}
```



Implementação em C - Encadeamento

```
Posicao 0: 40
Posicao 1:
Posicao 2: 12 12
Posicao 3: 23 13 33
Posicao 4: 14
Posicao 5: 35
Posicao 6:
Posicao 7: 7
Posicao 8:
Posicao 9:

Process returned 0 (0x0)   execution time : 0.001 s
Press ENTER to continue.
```



Implementação em Python

```
class HashingChain:

    # Funcao add valor na Tabela hash
    def insere(self, hashtable, valor_chave, valor):
        if(self.funcao == "Divisao"):
            endereco = self.hashingDivisao(valor_chave, hashtable)
        elif (self.funcao == "Multiplicacao"):
            endereco = self.hashingMultiplicacao(valor_chave,
                                                    hashtable)
        hashtable[endereco].insert(0, valor)
```



Implementação em Python

```
class HashingChain:
    def display_hash(self, hashTable):
        for i in range(len(hashTable)):
            print(i, end=" ")
            for j in hashTable[i]:
                print("-->", end=" ")
                print(j, end=" ")
            print()
```



Implementação em Python

```
if __name__ == '__main__':  
    HashTable = [[] for x in range(9)]  
    hashing = HashingChain(funcao="Divisao")  
    for valor in [5,28,19,15,20,33,12,17,10]:  
        hashing.insere(HashTable, valor, str(valor))  
    hashing.display_hash(HashTable)
```

Resultado

```
0  
1 -- > 10 -- > 19 -- > 28  
2 -- > 20  
3 -- > 12  
4  
5 -- > 5  
6 -- > 33 -- > 15  
7  
8 -- > 17
```



- └ Tratamento de Colisões
 - └ Endereçamento Aberto

- ▶ Os elementos estão armazenados na própria tabela hash.
- ▶ Não existem listas ou estruturas externas.
- ▶ Tabela pode ficar “cheia”
- ▶ $\alpha \leq 1$
- ▶ Não usa ponteiros
 - ▶ Calculamos a sequência de endereços
 - ▶ Memória extra liberada
 - ▶ Menor número de colisões
 - ▶ Recuperação mais rápida



- └ Tratamento de Colisões
 - └ Endereçamento Aberto

- ▶ *Possível fornecer vários endereços a partir de uma única chave*
- ▶ Processa uma análise sequencial ou sondagem nos endereços
 - ▶ $h : U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$
 - ▶ $h(k, x)$
- ▶ Pesquisar uma chave k
 - ▶ $h(k, 0) \rightarrow h(k, 1) \rightarrow \dots \rightarrow h(k, m - 1)$
 - ▶ Permutação de endereços para cada chave
 $\langle h(k, 0), h(k, 1), \dots, h(k, m - 1) \rangle$
 - ▶ $x = 0, \dots, m - 1$
 - ▶ sequência de tentativas ou sondagem
- ▶ sequência uniforme
 - ▶ qualquer $m!$ dos endereços, têm igual probabilidade de ser produzida por h



- └ Tratamento de Colisões
- └ Endereçamento Aberto

- ▶ Operação de remoção
 - ▶ Romperia a sequência de tentativas
 - ▶ Tornaria impossível recuperar qualquer chave **k**
 - ▶ Encontraria **null** no endereço
 - ▶ Assinalar a posição do armazenamento:
 - Livre (L) Quando posição não foi utilizada
 - Ocupado (O) Quando uma chave está armazenada
 - Removido (R) Quando armazena uma chave que já foi removida - uma nova chave poderá ocupar a posição marcada como removida



- └ Tratamento de Colisões
- └ Endereçamento Aberto

- ▶ Existem três métodos para a determinação da sequência de tentativas ou sondagem $h(x, k) \forall k = 0, 1, \dots, m - 1$.
 - ▶ tentativa linear
 - ▶ tentativa quadrática
 - ▶ hash duplo



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Tentativa linear

- ▶ Dada função hash auxiliar $h' : U \rightarrow 0, 1, \dots, m - 1$, método de tentativa linear usa a função hash:

$$h(k, i) = (h'(k) + i) \bmod m$$

- ▶ $i = 0, 1, \dots, m - 1$
- ▶ A ideia consiste em tentar armazenar chave k em $h'(k)$, se este já está ocupado, tentar no endereço consecutivo $h'(k) + 1$, até encontrar posição vazia.
- ▶ Problema: **Agrupamento primário**
 - ▶ Longas sequência de posições ocupadas são construídas, aumentando o tempo médio de pesquisa.
 - ▶ Causa: Uma posição vazia precedida por i posições completas é preenchida em seguida com probabilidade $\frac{i+1}{m}$



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Tentativa linear

- ▶ $m = 11$ e $h'(x) = x \bmod 11$
- ▶ $h(k, i) = (h'(k) + i) \bmod m$
- ▶ $[20, 30, 2, 13, 25, 24, 10, 9]$
 - ▶ $h(20, 0) = h'(20) \bmod 11 = \mathbf{9}$
 - ▶ $h(30, 0) = h'(30) \bmod 11 = \mathbf{8}$
 - ▶ $h(2, 0) = h'(2) \bmod 11 = \mathbf{2}$
 - ▶ $h(13, 0) = h'(13) \bmod 11 = 2$
 - ▶ $h(13, 1) = (h'(13) + 1) \bmod 11 = 3$
 - ▶ $25 \bmod 11 = 3 \rightarrow 3 + 1 = \mathbf{4}$
 - ▶ $24 \bmod 11 = 2 \rightarrow 2 + 1 \dots \rightarrow 2 + 3 = \mathbf{5}$
 - ▶ $10 \bmod 11 = \mathbf{10}$
 - ▶ $9 \bmod 11 = 9 \rightarrow 9 + 1 \rightarrow 9 + 2 = 0$

0	9
1	
2	2
3	13
4	25
5	24
6	
7	
8	30
9	20
10	10

- └ Tratamento de Colisões
- └ Endereçamento Aberto

Implementação em C

```
int tentativaLinear(int table[],int hash)
{
    // Tratamento de colisao por sondagem linear
    while (table[hash] != -1) {
        hash = (hash + 1) % TABLE_SIZE;
    }
    return hash;
}
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Implementação em C

```
void insere(int table[], int key, char funcao) {  
    if(funcao == 'D'){  
        int hash = key % TABLE_SIZE;  
        hash = tentativaLinear(table, hash)  
        table[hash] = key;  
    }  
    else if(funcao == 'M')  
    {  
        float A = (sqrt(5)-1)/2;  
        float m = TABLE_SIZE;  
        int hash = floor(m*(fmod(key*A , 1)));  
        table[hash] = key;  
    }  
}
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Implementação em C

```
#define TABLE_SIZE 11

int main() {
    int hash_table[TABLE_SIZE];
    initialize_table(hash_table, TABLE_SIZE);
    int keys[] = {20,30,2,13,25,24,10,9};
    int num_keys = sizeof(keys) / sizeof(keys[0]);
    for (int i = 0; i < num_keys; i++) {
        int key = keys[i];
        insere(hash_table, key, 'D');
    }
    // Imprimir tabela hash
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("Index: %d, Key: %d\n", i, hash_table[i]);
    }
    return 0;
}
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Implementação em Python

```
class HashingOpenAddressing:
    def __init__(self, metodo):
        self.metodo = metodo;

    def linear(self, chave, hashTable):
        #HashTable = np.array([[None] * 11, ['L'] * 11])
        m = len(hashTable[0])
        h1 = chave % m
        for i in range(m):
            h = (h1+i) % m
            if(hashTable[1,h] == 'L'):
                break;
        return h;
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Implementação em Python

```
# Funcao add valor na Tabela hash
def insere(self, hashtable, valor_chave, valor):
    if(self.metodo == "linear"):
        endereco = self.linear(valor_chave, hashtable)
    elif(self.metodo == "quadratica"):
        endereco = self.quadratica(valor_chave, hashtable)
    hashtable[0, endereco] = valor
    hashtable[1, endereco] = '0'

def display_hash(self, hashTable):
    for i in range(len(hashTable[0])):
        print(str(i) + " --> " + str(hashTable[0,i]) + " (" + str(
            hashTable[1,i]) + ")", end="\n")
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

implementação em Python

```
HashTable = np.array([[None] * 11, ['L'] * 11])
hashing = HashingOpenAddressing(metodo="linear")
for valor in [20, 30, 2, 13, 25, 24, 10, 9]:
    hashing.insere(HashTable, valor, str(valor))
hashing.display_hash(HashTable)
```

Resultado

```
0 -- > 9(O)
1 -- > None(L)
2 -- > 2(O)
3 -- > 13(O)
4 -- > 25(O)
5 -- > 24(O)
6 -- > None(L)
7 -- > None(L)
8 -- > 30(O)
9 -- > 20(O)
10 -- > 10(O)
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Tentativa Quadrática

- ▶ **Tentativa linear** as chaves tendem a se concentrar, criando **agrupamentos primários**, que aumentam muito o tempo de busca
 - ▶ A ideia é obter sequências de endereços diversos para endereços próximos.

$$h(k, i) = (h'(k) + c_1i + c_2i^2) \bmod m$$

- ▶ c_1 e $c_2 \neq 0$
- ▶ $i = 0, 1, \dots, m - 1$



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Tentativa Quadrática

- ▶ Esse método consegue evitar os agrupamentos primários da tentativa linear.
- ▶ **Agrupamento Secundário**
 - ▶ $h(k_1, 0) = h(k_2, 0) \rightarrow h(k_1, i) = h(k_2, i)$
 - ▶ Degradações são reduzidas se comparadas com a Tentativa linear.



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Tentativa Quadrática

- ▶ Os valores m , c_1 e c_2 devem ser escolhidos de tal forma que os endereços $h(x, k)$ correspondam a varrer toda a tabela para $k = 0, 1, \dots, m - 1$
- ▶ As equações recorrentes fornecem uma maneira de calcular, diretamente esses endereços:

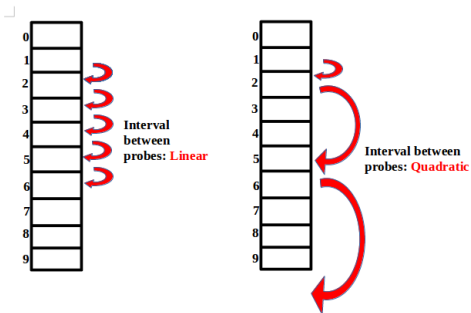
$$h(x, 0) = h'(x)$$

$$h(x, k) = (h(x, k - 1) + k) \bmod m, 0 < k < m$$



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Tentativa Quadrática



Comparação de Comportamento



Tentativa Quadrática

- ▶ $m = 11$ e $h'(x) = x \bmod 11$
- ▶ $[20, 30, 2, 13, 25, 24, 10, 9]$
 - ▶ $h(20, 0) = h'(20) \bmod 11 = 9$
 - ▶ $h(30, 0) = h'(30) \bmod 11 = 8$
 - ▶ $h(2, 0) = h'(2) \bmod 11 = 2$
 - ▶ $h(13, 0) = h'(13) \bmod 11 = 2$
 - ▶ $h(13, 1) = (h'(13) + 1^2) \bmod 11 = 3$
 - ▶ $25 \bmod 11 = 3 \rightarrow 3 + 1 = 4$
 - ▶ $24 \bmod 11 = 2 \rightarrow 2 + 1^2 \rightarrow 2 + 2^2 = 6$
 - ▶ $10 \bmod 11 = 10$
 - ▶ $9 \bmod 11 = 9 \rightarrow 9 + 1^2 \rightarrow 9 + 2^2 \rightarrow 9 + 3^2 = 7$

0	
1	
2	2
3	13
4	25
5	
6	24
7	9
8	30
9	20
10	10



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Implementação em C

```
int tentativaQuadratica(int table[],int hash)
{
    // Tratamento de colisao por sondagem quadratica
    for(int i = 0 ;i<TABLE_SIZE; i++)
    {
        int h = (hash + i*i) % TABLE_SIZE;
        if(table[h] == -1)
            return h;
    }
}
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Implementação em C

```
void insere(int table[], int key, char funcao) {  
    if(funcao == 'D'){  
        int hash = key % TABLE_SIZE;  
        hash = tentativaQuadratica(table, hash);  
        table[hash] = key;  
    }  
    else if(funcao == 'M')  
    {  
        float A = (sqrt(5)-1)/2;  
        float m = TABLE_SIZE;  
        int hash = floor(m*(fmod(key*A , 1)));  
        table[hash] = key;  
    }  
}
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Implementação em C

```
#define TABLE_SIZE 11
int main() {
    int hash_table[TABLE_SIZE];
    initialize_table(hash_table, TABLE_SIZE);
    int keys[] = {20,30,2,13,25,24,10,9};
    int num_keys = sizeof(keys) / sizeof(keys[0]);

    for (int i = 0; i < num_keys; i++) {
        int key = keys[i];
        insere(hash_table, key, 'D');
    }
    for (int i = 0; i < TABLE_SIZE; i++) {
        printf("Index: %d, Key: %d\n", i, hash_table[i]);
    }

    return 0;
}
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Implementação em Python

```
class HashingOpenAddressing:
    def __init__(self, metodo):
        self.metodo = metodo;

    def quadratica(self, chave, hashTable):
        m = len(hashTable[0])
        h1 = chave % m
        for i in range(m):
            h = (h1+i*i) % m
            if hashTable[1,h] == 'L':
                break;
        return h;
```



- └ Tratamento de Colisões
- └ Endereçamento Aberto

implementação em Python

```
HashTable = np.array([[None] * 11, ['L'] * 11])  
hashing = HashingOpenAddressing(metodo="quadratica")  
for valor in [20, 30, 2, 13, 25, 24, 10, 9]:  
    hashing.insere(HashTable, valor, str(valor))  
hashing.display_hash(HashTable)
```

Resultado

```
0 -- > None(L)  
1 -- > None(L)  
2 -- > 2(O)  
3 -- > 13(O)  
4 -- > 25(O)  
5 -- > None(L)  
6 -- > 24(O)  
7 -- > 9(O)  
8 -- > 30(O)  
9 -- > 20(O)  
10 -- > 10(O)
```



Hash Duplo

- ▶ Um dos melhores métodos para endereçamento aberto
- ▶ Por que as permutações produzidas possuem características de permutações aleatórias

$$h(k, i) = (h_1(k) + i \times h_2(k)) \bmod m$$

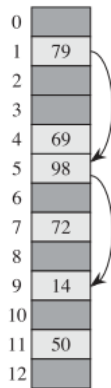
- ▶ Endereço inicial $T[h_1(k)]$
- ▶ Endereços sucessivos dependem da quantidade $h_2(k)$
- ▶ O deslocamento depende da chave k de duas maneiras



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Hash Duplo

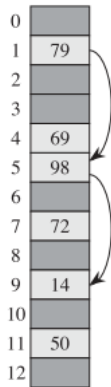
- ▶ $m = 13$
- ▶ $h_1(k) = k \bmod 13$
 $h_2(k) = 1 + (k \bmod 11)$
- ▶ $h(k, i) = (h_1(k) + i \times h_2(k)) \bmod 13$
- ▶ Inserir **k=14**



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Hash Duplo

- ▶ $m = 13$
- ▶ $h_1(k) = k \bmod 13$
 $h_2(k) = 1 + (k \bmod 11)$
- ▶ $h(k, i) = (h_1(k) + i \times h_2(k)) \bmod 13$
- ▶ Inserir **k=14**
- ▶ $((14 \bmod 13) + 0(1 + (14 \bmod 11))) \bmod 13 = 1$



- └ Tratamento de Colisões
- └ Endereçamento Aberto

Hash Duplo

- ▶ Análise do hash do endereço aberto

- ▶ *Hash uniforme*

- ▶ Em termos de $\alpha \leq 1$ e $n \leq m$

- ▶ Pesquisa

- ▶ malsucedida é executada $\frac{1}{1 - \alpha}$

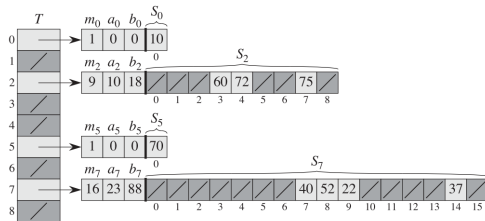
- ▶ bem sucedida é executada $\frac{1}{\alpha} \ln \frac{1}{1 - \alpha}$






└ Tratamento de Colisões

└ Hash Perfeito

- ▶ Chaves estáticas
- ▶ Se o número de acessos de memória exigidos no pior caso para executar pesquisa é $O(1)$
- ▶ Esquema de criar hash em dois níveis
 - ▶ Primeiro nível é hash com encadeamento.
 - ▶ Segundo nível ao invés de usar uma lista usamos tabela hash secundária






Referências I

-  Lee K.D., Hubbard S. (2015) Trees. In: Data Structures and Algorithms with Python. Undergraduate Topics in Computer Science. Springer, Cham. Retrieved from https://doi.org/10.1007/978-3-319-13072-9_6
-  Hubbard, J. (2007). Schaum's Outline of Data Structures with Java. Retrieved from <http://www.amazon.com/Schaums-Outline-Data-Structures-Java/dp/0071476989>
-  Cormen, T. H., Leiserson, C. E., & Stein, R. L. R. E. C. (2012). Algoritmos: teoria e prática. Retrieved from <https://books.google.com.br/books?id=6iA4LgEACAAJ>.



Referências II

-  Ascenio, Ana Fernanda Gomes. Estrutura de dados: Algoritmos, análise da complexidade e implementações em Java e C++. São Paulo: Pearson Prentice Hall, 2010.
-  Donald E. Knuth. Sorting and Searching, volume 3 of The Art of Computer Programming. Addison-Wesley, 1973. Second edition, 1998
-  Szwarcfiter, Jayme Luiz. Estruturas de dados e seus algoritmos / Jayme Luiz Szwarcfiter, Lilian Markenzon. 3.ed. [Reimpr.]. - Rio de Janeiro : LTC, 2015.





Hash

Professor: Elton Sarmanho¹

E-mail: eltonss@ufpa.br



¹Faculdade de Sistemas de Informação - UFPA/CUTINS

7 de outubro de 2025