UNIVERSIDADE FEDERAL RURAL DO SEMI-ÁRIDO

COMPONENTE CURRICULAR: ALGORITMOS E ESTRUTURA DE DADOS II

DOCENTE: KENNEDY REURISON LOPES

DISCENTE(S): ISABEL DE PAIVA FREIRE 2024010417

LISTA II UNIDADE

PAU DOS FERROS

JUNHO DE 2025

COMPRESSÃO DE DADOS

01. Calcule a taxa de compressão de: “ABABABACBABABABA”.

Tamanho da mensagem original = 16 x 8 = 128

Frequência dos caracteres:

A = 9

B = 6

C = 1

Total 16

Juntando os elementos

C + B = 7

CB + A = 16

Codificando a string:

Letra	**Código**	**Bits usados**
A	1	1
B	01	2
A	1	1
B	01	2
A	1	1
B	01	2
A	1	1
C	00	2
B	01	2
A	1	1
B	01	2
A	1	1
B	01	2

A	1	1
B	01	2
A	1	1

Somando os bits:

A (9 vezes × 1 bit) → 9 bits

B (6 vezes × 2 bits) → 12 bits

C (1 vez × 2 bits) → 2 bits

Total = 23 bits

128 - 23 / 128 (x100) = 82%

02. Utilize o resultado da compressão dos dados para representar: “ACCCCCCCCC”.

Caractere	**Código**
A	1
B	01
C	00

A → 1 → 1

B → 9 → 01

Calculando os bits

A = 1x1 = 1

B = 9 x 2 = 18

Total = 19 bits

03. Dado um texto qualquer, a compressão é única? Explique o porquê.

A compressão de um texto não é única porque depende do algoritmo escolhido, das estratégias de implementação, e das especificidades do conteúdo a ser comprimido.

04. A Lei de Benford é amplamente reconhecida como uma ferramenta eficaz para analisar dados e determinar sua autenticidade ou possíveis indícios de manipulação. Esta abordagem é

frequentemente utilizada em diversos contextos, como em pesquisas, para avaliar a validade dos dados coletados. A Lei de Benford sugere que, em um conjunto de números autênticos, os demais significativos aparecem com uma probabilidade específica, indicando se os números refletem ou não a realidade.

Dígito	1	2	3	4	5	6	7	8	9
Prob(%)	30.1	17.6	12.5	9.7	7.9	6.7	5.8	5.1	4.6

Posição	Cidade	Pop
1	São Paulo, SP	12.33 M
2	Rio de Janeiro, RJ	6.75 M
3	Brasília, DF	3.11 M
4	Salvador, BA	2.88 M
5	Fortaleza, CE	2.69 M
6	Belo Horizonte, MG	2.52 M
7	Manaus, AM	2.22 M
8	Curitiba, PR	1.95 M
9	Recife, PE	1.65 M
10	Porto Alegre, RS	1.48 M
11	Belém, PA	1.49 M
12	Goiânia, GO	1.53 M
13	Guarulhos, SP	1.39 M
14	Campinas, SP	1.2 M
15	São Luís, MA	1.1 M
16	São Gonçalo, RJ	1.05 M
17	Maceió, AL	1.02 M
18	Duque de Caxias, RJ	932 k
19	Natal, RN	890 k
20	Teresina, PI	868 k
21	S. B. do Campo, SP	837 k
22	Nova Iguaçu, RJ	818 k
23	Campo Grande, MS	906 k
24	Osasco, SP	696 k
25	Santo André, SP	721 k

Posição	Cidade	Pop
26	João Pessoa, PB	809 k
27	Jaboatão, PE	698 k
28	S. J. dos Campos, SP	729 k
29	Ribeirão Preto, SP	711 k
30	Uberlândia, MG	699 k
31	Contagem, MG	668 k
32	Sorocaba, SP	666 k
33	Aracaju, SE	664 k
34	Feira de Santana, BA	627 k
35	Cuiabá, MT	618 k
36	Joinville, SC	597 k
37	Juiz de Fora, MG	563 k
38	Londrina, PR	575 k
39	Niterói, RJ	515 k
40	Ap. de GO, GO	590 k
41	Ananindeua, PA	525 k
42	Belford Roxo, RJ	502 k
43	São João de Meriti, RJ	473 k
44	C. dos Goy., RJ	507 k
45	Caxias do Sul, RS	516 k
46	Santos, SP	434 k
47	Betim, MG	425 k
48	Olinda, PE	390 k
49	S. J. do R. Preto, SP	461 k
50	Diadema, SP	416 k

Com base na Lei de Benford, determine computacionalmente se os dados sobre a população das 50 maiores cidades do Brasil (a) obedecem a essa lei e (b) construa uma árvore de Huffman para comprimir dados que seguem esta lei.

a) Código:

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

#define NUM_CIDADES 50


float populacoes[NUM_CIDADES] = {
    12.33, 6.75, 3.11, 2.88, 2.69, 2.52, 2.22, 1.95, 1.65, 1.48,
    1.49, 1.53, 1.39, 1.2, 1.1, 1.05, 1.02, 0.932, 0.89, 0.868,
```

```
    0.837, 0.818, 0.906, 0.696, 0.721, 0.809, 0.698, 0.729, 0.711,
0.699,
    0.668, 0.666, 0.664, 0.627, 0.618, 0.597, 0.563, 0.575, 0.515,
0.59,
    0.525, 0.502, 0.473, 0.507, 0.516, 0.434, 0.425, 0.39, 0.461, 0.416
};


float benford_probs[9] = {30.1, 17.6, 12.5, 9.7, 7.9, 6.7, 5.8, 5.1,
4.6};


int primeiro_digito(float num) {
    if (num == 0) return 0;


    while (num < 1.0) {
        num *= 10.0;
    }


    return (int)num;
}

int main() { // função principal
    int freq[9] = {0}; // frequência para dígitos 1 a 9
    int i;
    int d;


    for (i = 0; i < NUM_CIDADES; i++) { // função para contar as
frequências
        d = primeiro_digito(populacoes[i]);
        if (d >= 1 && d <= 9) {
            freq[d - 1]++;
        }
    }

    printf("Digito | Frequencia (%%) | Benford (%%)\n");
    printf("----------------------------------------\n");
    for (i = 0; i < 9; i++) {
        float freq_percent = (freq[i] / (float)NUM_CIDADES) * 100.0;
```

```
        printf("    %d     |      %6.2f      |     %5.2f\n", i + 1,
freq_percent, benford_probs[i]);
    }


    printf("\nDiferencas em pontos percentuais:\n"); // mostrar as
diferenças
    for (i = 0; i < 9; i++) {
        float freq_percent = (freq[i] / (float)NUM_CIDADES) * 100.0;
        float diff = freq_percent - benford_probs[i];
        if(diff < 0) diff = -diff;
        printf("Digito %d: %.2f%%\n", i + 1, diff);
    }

    return 0;
}
```

Saída do código:

```
Digito | Frequencia (%) | Benford (%)
----------------------------------------
   1   |      20.00     |    30.10
   2   |       8.00     |    17.60
   3   |       4.00     |    12.50
   4   |      10.00     |     9.70
   5   |      18.00     |     7.90
   6   |      18.00     |     6.70
   7   |       6.00     |     5.80
   8   |      10.00     |     5.10
   9   |       4.00     |     4.60

Diferencas em pontos percentuais:
Digito 1: 10.10%
Digito 2: 9.60%
Digito 3: 8.50%
Digito 4: 0.30%
Digito 5: 10.10%
Digito 6: 11.30%
Digito 7: 0.20%
Digito 8: 4.90%
Digito 9: 0.60%
```

b) Código:

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

#define MAX_NODES 2*9-1  // número máximo de nós na árvore
```

```c
typedef struct Node {
    int symbol;        // dígito (1-9) ou -1 para nó interno
    float freq;        // frequência (probabilidade)
    struct Node *left, *right;
} Node;


Node* criar_no(int symbol, float freq) { //função para criar um novo nó
    Node* n = (Node*) malloc(sizeof(Node));
    n->symbol = symbol;
    n->freq = freq;
    n->left = n->right = NULL;
    return n;
}


typedef struct { // estrutura para fila de prioridade mínima simples
    Node* nodes[MAX_NODES];
    int size;
} MinHeap;

void swap(Node** a, Node** b) {
    Node* temp = *a;
    *a = *b;
    *b = temp;
}

void minHeapify(MinHeap* heap, int idx) {
    int smallest = idx;
    int left = 2*idx + 1;
    int right = 2*idx + 2;

    if (left < heap->size && heap->nodes[left]->freq <
heap->nodes[smallest]->freq)
        smallest = left;
    if (right < heap->size && heap->nodes[right]->freq <
heap->nodes[smallest]->freq)
        smallest = right;
    if (smallest != idx) {
        swap(&heap->nodes[smallest], &heap->nodes[idx]);
        minHeapify(heap, smallest);
    }
```

```
}

Node* extractMin(MinHeap* heap) {
    Node* temp = heap->nodes[0];
    heap->nodes[0] = heap->nodes[heap->size -1];
    heap->size--;
    minHeapify(heap, 0);
    return temp;
}

void insertHeap(MinHeap* heap, Node* node) {
    heap->size++;
    int i = heap->size -1;
    heap->nodes[i] = node;
    while (i != 0 && heap->nodes[(i-1)/2]->freq > heap->nodes[i]->freq)
{
        swap(&heap->nodes[i], &heap->nodes[(i-1)/2]);
        i = (i-1)/2;
    }
}

void printCodes(Node* root, int arr[], int top) {
    if (root->left) {
        arr[top] = 0;
        printCodes(root->left, arr, top+1);
    }
    if (root->right) {
        arr[top] = 1;
        printCodes(root->right, arr, top+1);
    }
    if (!root->left && !root->right) {
        printf("Digito %d: codigo ", root->symbol);
        for (int i=0; i<top; i++)
            printf("%d", arr[i]);
        printf("\n");
    }
}

int main() { //função principal
    int n = 9;
    int digits[9] = {1,2,3,4,5,6,7,8,9};
    float freqs[9] = {30.1, 17.6, 12.5, 9.7, 7.9, 6.7, 5.8, 5.1, 4.6};
```

```
    MinHeap heap;
    heap.size = 0;


    for (int i=0; i<n; i++) { // função para criar um nó folha para cada dígito e inserir na heap
        Node* no = criar_no(digits[i], freqs[i]);
        insertHeap(&heap, no);
    }


    while (heap.size > 1) { // função para construir árvore de Huffman
        Node* left = extractMin(&heap);
        Node* right = extractMin(&heap);

        Node* interno = criar_no(-1, left->freq + right->freq);
        interno->left = left;
        interno->right = right;

        insertHeap(&heap, interno);
    }


    int arr[20], top=0;
    printf("Codigos de Huffman para digitos segundo Lei de Benford:\n");
    printCodes(heap.nodes[0], arr, top);

    return 0;
}
```

Saída do código:

```
Codigos de Huffman para digitos segundo Lei de Benford:
Digito 2: codigo 00
Digito 9: codigo 0100
Digito 8: codigo 0101
Digito 3: codigo 011
Digito 1: codigo 10
Digito 7: codigo 1100
Digito 6: codigo 1101
Digito 5: codigo 1110
Digito 4: codigo 1111
PS C:\Users\Isa\Desktop\Isabel\UFERSA\III Período\Algoritmos & estrutura de
```

ÁRVORES AVL

05. Considere uma árvore AVL iniciada apenas com o valor de 50. Realize o processo de inserção dos elementos na ordem indicada nesta árvore AVL.

    X = {35, 85, 48, 47, 24, 40, 69, 93, 31, 11, 77, 30, 74, 67, 87, 98, 40, 83, 18, 35}

    Após a inserção de cada número, realize o processo de balanceamento, indicando qual rotação foi necessária para balancear.

1. Inicializar a árvore com o valor 50

```
      50
```

2. Inserção do 35

```
      50
     /
   35
```

3. Inserção do 85

```
      50
     /  \
   35    85
```

4. Inserção do 48

```
   50
  / \
 35   85
  \
   48
```

5. Inserção do 47

```
   50
  / \
 35   85
  \
   48
  /
 47
```

6. A árvore ficou desbalanceada com a inserção do 47, então é necessário fazer uma rotação esquerda-direita para balancear

```
   50
  / \
 47   85
 / \
35  48
```

7. Inserção do 40 - já com o elemento rotação direita esquerda

```
      50
     /  \
   40    85
  /  \
 35  47
/
24
```

8. Inserção do 69 - 93 e 31 já com as devidas rotações aplicadas

```
        50
     /       \
   35        85
  /  \      /  \
24   40   69   93
   /
  31
```

9. Inserção do 11 - 77 - 30 - 74 - 67 - 87 - 98 - 40 - 83 - 18 - 35 já com as devidas rotações

```
                [50]
              /       \
          [35]       [69]
         /   \        /   \
      [24]  [40] [67]  [85]
      /     \    \         /    \
   [11] [31]   [48]   [74]    [93]
      \      /               \       /  \
   [18]  [30]         [77]    [87][98]
                         /
                      [83]
```

06. Julgue a seguinte afirmação: "Toda árvore binária de busca cheia é necessariamente uma árvore AVL." Caso seja falso, mostre uma árvore binária de busca completa que não é AVL. Caso seja verdadeiro, explique em termos dos fatores de balanceamento.

A afirmação é falsa. Árvore AVL é uma árvore binária balanceada, como ela é balanceada, não significa que seja necessariamente cheia.

Exemplo de árvore binária de busca completa que não é AVL:

```
  10
 /  \
```

```
   5    15
  / \
 2   7
/
1
```

07. Julgue a seguinte afirmação: “Toda árvore AVL é necessariamente uma árvore binária de busca completa” Caso seja falso, mostre uma árvore AVL que não é Binária de busca completa. Caso seja verdadeiro, explique em termos dos fatores de balanceamento.

Essa afirmação também é falsa, uma vez que nem toda árvore AVL é necessariamente uma árvore binária de busca completa. O que justifica é que a árvore AVL garante o balanceamento dos nós, mas não exige que os níveis estejam totalmente preenchidos (que é o que exige a árvore binária de busca completa).

Exemplo de árvore AVL que não é binária de busca completa:

```
    10
   /  \
  5    15
 /
2
```

08. Utilizando a árvore construída na questão 5, realize o processo de remoção dos nós um a um, na mesma ordem que foi inserido. Realize o processo de rotação a cada momento que for necessário.

1. Remove o 35 - o 40 sobe e o 35 é eliminado
2. Remover o 85 - rotação a direita, no nó 93
3. Remover o 48 - remoção simples
4. Remover o 47 - remoção simples
5. Remover o 24 - rebalancear o 31
6. Remover o 48 - remoção simples
7. Remover o 69- rebalancear o 74
8. Remover o 93- remoção simples
9. Remover o 31- remoção simples
10. Remover o 11- remoção simples
11. Remover o 77- remoção simples
12. Remover o 30- remoção simples
13. Remover o 74- rebalancear o 69
14. Remover o 67- remoção simples
15. Remover o 87- rebalancear o 83

16. Remover o 98 - remoção simples
17. Remover o 83- remoção simples
18. Remover o 18 - remoção simples
19. Remover o 50- remoção simples

ÁRVORE 2-3

09. Considere uma árvore 2-3 iniciada apenas com o valor de 50. Realize o processo de inserção dos elementos na ordem indicada nesta árvore AVL.

    X = {12, 86, 68, 99, 82, 59, 65, 70, 16, 58, 40, 67, 22, 48, 59, 11, 52, 91, 65, 73}

    Após a inserção de cada número, realize o processo de balanceamento, garantindo que os nós folhas estejam sempre no último nível da árvore.

```
                      [50]
                  /           \
               [16]          [65]
              /    \        /       \
      [11,12]  [22,40]  [52]      [68, 91]
                            \    /  |        |    \
                        [59] [67] [70,73] [82] [99]
```

10. Julgue a seguinte afirmação, provando se é verdadeiro ou falso: "Uma ´árvore 23 com N chaves possui a altura maior com exatamente log3(N) níveis".

A afirmação é falsa porque a altura máxima ocorre quando há menos ramificação, ou seja, quando os nós são do tipo 2, e não do tipo 3. Logo, a altura mínima de uma árvore 2-3 é aproximadamente log3(N), e a altura máxima é aproximadamente log2(N). Portanto, log3(N) representa a menor altura possível, não a maior.

11. Calcule:

    a) A maior e menor quantidade de chaves que uma árvore 2-3 com 10 níveis possui?

Menor quantidade

2^10-1 = 2^9= 512

Total de nós 2^10=1024 (-1) = 1023 (quantidade mínima)

Maior quantidade

3^9=19683

19683-1=19682

19682 / 2 = 9841

9841 + 19682 = 29523

29523 x 2 = 59046 (quantidade máxima)

    b) A maior e menor quantidade de nós que uma árvore 2-3 com 10 níveis possui?

Menor quantidade

2^9 = 512

2^9-1=511

512 + 511 = 1023

Maior quantidade

3^9 = 19683

3^9-1 / 2 = 9841

19683 + 9841 = 29524

c) A altura de uma árvore 2-3 com 105 chaves?

Altura mínima = log3(n)

log10(105) / log10 (3) = 2021 / 0.477 = 4.23

altura mínima = 5

HEAP

12. Considere uma HEAP como a mostrada logo abaixo para realizar as operações na ordem que são solicitadas:

    HEAP = {97, 88, 84, 72, 55, 44, 37, 30, 26, 12, 18, 20, 25, 14, 8, 10, 6, 15, 5, 9}

    a) Modifique a prioridade de 20 para 40;

[97, 88, 84, 72, 55, 44, 37, 30, 26, 12, 18, 40, 25, 14, 8, 10, 6, 15, 5, 9]

Ainda é uma heap

    b) Modifique a prioridade de 9 para 99;

O valor 9 foi substituído por 99, e ele subiu para o topo

[99, 97, 84, 72, 88, 44, 37, 30, 26, 55, 18, 40, 25, 14, 8, 10, 6, 15, 5, 12]

Foi preciso reorganizar as prioridades

Continua sendo uma heap agora

    c) Modifique a prioridade e 97 para 11;

O valor 97 (anteriormente o segundo maior) foi reduzido para 11:

[99, 88, 84, 72, 55, 44, 37, 30, 26, 12, 18, 40, 25, 14, 8, 10, 6, 15, 5, 11]

O valor 11 foi preciso descer para obedecer a lista de prioridades

Continua sendo uma heap

    d) Remova o elemento com maior prioridade;

Basta remover o 99, que é a raíz do heap

[88, 84, 44, 72, 55, 40, 37, 30, 26, 12, 18, 5, 25, 14, 8, 10, 6, 15, 11]

Continua sendo uma heap

    e) Remova o elemento com maior prioridade;

Basta remover o 88, que agora é a nova raíz

[84, 72, 44, 30, 55, 40, 37, 11, 26, 12, 18, 5, 25, 14, 8, 10, 6, 15]

Continua sendo uma heap.

Após cada operação, certifique-se que a estrutura continua sendo uma HEAP.

13. Julgue as afirmações seguintes:

    a) Toda HEAP-MAX é uma lista em ordem decrescente.

Falso, uma heap-max garante apenas que cada nó é maior ou igual aos seus filhos diretos. Não significa que os elementos estão em ordem decrescente total na estrutura ou no vetor.

    b) Toda lista em ordem decrescente é uma HEAP-MAX.

Também é falso. Mesmo que uma lista esteja em ordem decrescente, isso não garante a propriedade do heap quando representada como árvore. A posição dos elementos pode quebrar a regra de pai ≥ filhos.

    c) O menor elemento é o último elemento da lista.

Este item também é falso. Em um HEAP-MAX, não há garantia sobre a posição do menor elemento. O menor pode estar em qualquer folha, e o vetor que representa o heap não está ordenado.

d) Um elemento de um nível menor tem prioridade menor do que todos os de níveis acima.

Verdadeiro. Em uma HEAP-MAX, cada elemento no nível abaixo é filho de algum elemento do nível acima, então sua prioridade (valor) é menor ou igual.

e) A HEAP-MIN pode ser construída a partir de uma HEAP-MAX invertendo o vetor de prioridades.

Este item é falso. Inverter o vetor não transforma uma HEAP-MAX em uma HEAP-MIN. As estruturas têm regras diferentes de organização:

HEAP-MAX: pai ≥ filhos

HEAP-MIN: pai ≤ filhos

14. Construa uma HEAP-MIN com os elementos inseridos na HEAP na ordem indicada:
    X = {92, 24, 67, 30, 61, 58, 36, 33, 14, 81, 55, 16, 26, 51, 39, 15, 82, 49, 90, 84}

1. Passo: reorganizar o vetor em ordem crescente
   [14, 15, 16, 24, 55, 26, 39, 30, 33, 81, 61, 67, 36, 58, 51, 92, 82, 49, 90, 84]
2. Passo: montar a árvore binária

```
              14
           /       \
          15        16
        /   \      /   \
      24    55    26    39
     / \   / \   / \   / \
    30 33 81 61 67 36 58 51
   / \
  92  82
 / \
49  90
/
84
```

TABELA HASH

15. Apresente o objetivo (pretensão) da tabela Hash e apresente o porquê as colisões dificultam essa estrutura alcançar este objetivo.

O principal objetivo de uma tabela hash é permitir acesso, inserção e remoção de dados em tempo constante — ou seja, O(1) na média. A ideia é que, ao aplicar uma função hash sobre uma chave, seja possível localizar rapidamente onde o dado está armazenado, sem precisar percorrer a estrutura inteira (como em listas ou árvores).

Uma colisão acontece quando duas ou mais chaves diferentes produzem o mesmo índice na tabela hash. Em vez de acessar o valor diretamente no índice desejado, o algoritmo precisa usar um método extra para resolver a colisão: procurar outro espaço (endereçamento aberto) ou manter uma lista no índice (encadeamento). Isso significa mais comparações, mais passos, logo o acesso pode deixar de ser O(1) e se aproximar de O(n) no pior caso.

16. Considere uma tabela Hash com 16 elementos e tratamento de colisão no formato de endereçamento aberto no formato com sondagem linear. Apresente a configuração final da tabela ao tentar inserir os elementos na ordem apresentada abaixo:
    X = {13, 12, 27, 77, 32, 16, 49}

Cálculo do mod do número de elementos:

Elemento	**h(x) = x mod 16**	**Inserido em**	**Justificativa**
13	13	13	vaga
12	12	12	vaga
27	11	11	vaga
77	13	14	13 ocupado → próximo livre
32	0	0	vaga
16	0	1	0 ocupado → vai para 1
49	1	2	1 ocupado → vai para 2

Configuração da tabela:

Índice	**Valor**
0	32
1	16
2	49
3	—
4	—
5	—
6	—
7	—
8	—
9	—
10	—
11	27
12	12
13	13
14	77
15	—

17. Realize o mapeamento da chave 18 utilizando o método da multiplicação em uma tabela com apenas 8 posições. Em qual dessas 8 posições estará o elemento 18?

É necessário, utilizar a fórmula:

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

Assumindo que A = 0,6180339887

Temos que:

$k \cdot A = 18 \times 0{,}6180339887 = 11{,}124611797$

$k \cdot A \bmod 1 \approx 0{,}124611797$

$8 \cdot 0{,}124611797 \approx 0{,}996894376$

Arredonda para 0

A chave 18 será mapeada para a posição 0 da tabela hash com 8 posições, usando o método da multiplicação.

18. Calcule mapeamentos dos valores:

{61, 58, 36, 33, 14, 81, 55, 16, 26, 51, 39}

a) Em uma tabela com 4 posições utilizando método da divisão e tratamento de colisão por endereçamento encadeado exterior;

Método da divisão, fórmula:

h(x) = x mod 4.

61: 61 mod 4= 1

58: 58 mod 4=2

36: 36 mod 4=0

33: 33 mod 4=1

14: 14 mod 4=2

81: 81 mod 4=1

55: 55 mod 4=3

16: 16 mod 4=0

26: 26 mod 4=2

51: 51 mod 4=3

39: 39 mod 4=3

Índice 0: {36, 16}

Índice 1: {61, 33, 81}

Índice 2: {58, 14, 26}

Índice 3: {55, 51, 39}

b) Em uma tabela com 20 posições, utilizando o método da dobra e tratamento de colisão por encadeamento exterior com 8 valores primários e o restante para extensão.

Primeiro, temos que dividir a tabela em duas partes:

Área Primária: 8 posições (índices 0 a 7)

Área de Extensão: as outras 12 posições (índices 8 a 19), onde serão armazenados os elementos que causarem colisão na área primária.

61: $6 + 1 = 7 \rightarrow 7 \mod 8 = 7$

58: $5 + 8 = 13 \rightarrow 13 \mod 8 = 5$

36: $3 + 6 = 9 \rightarrow 9 \mod 8 = 1$

33: $3 + 3 = 6 \rightarrow 6 \mod 8 = 6$

14: $1 + 4 = 5 \rightarrow 5 \mod 8 = 5$

81: $8 + 1 = 9 \rightarrow 9 \mod 8 = 1$

55: $5 + 5 = 10 \rightarrow 10 \mod 8 = 2$

16: $1 + 6 = 7 \rightarrow 7 \mod 8 = 7$

26: $2 + 6 = 8 \rightarrow 8 \mod 8 = 0$

51: $5 + 1 = 6 \rightarrow 6 \mod 8 = 6$

39: $3 + 9 = 12 \rightarrow 12 \mod 8 = 4$

Distribuição dos elementos:

Índice 0 (primário): Recebe: 26

Índice 1 (primário):Recebe: 36. Colisão: 81 → vai para área de extensão (primeiro slot disponível, por exemplo, índice 8)

Índice 2 (primário): Recebe: 55

Índice 3 (primário): (vazio)

Índice 4 (primário): Recebe: 39

Índice 5 (primário): Recebe: 58. Colisão: 14 → vai para extensão (próximo slot disponível, por exemplo, índice 9)

Índice 6 (primário): Recebe: 33. Colisão: 51 → vai para extensão (por exemplo, índice 10)

Índice 7 (primário): Recebe: 61. Colisão: 16 → vai para extensão (por exemplo, índice 11)

Área primária:

0: 26

1: 36

2: 55

3: [vazio]

4: 39

5: 58

6: 33

7: 61

Área de extensão:

8: 81 (extensão da posição 1)

9: 14 (extensão da posição 5)

10: 51 (extensão da posição 6)

11: 16 (extensão da posição 7)

12 a 19: [vazias]

c) Em uma tabela com 16 posições pelo método da multiplicação sem tratamento de colisão.

Utiliza-se a mesma fórmula anterior:

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

61:

- $61\times0{,}618\approx37{,}69861$
- Parte fracionária: 0,6980
- $0{,}698\times16\approx11{,}168$
- Índice = 11

58:

- $58\times0{,}618\approx35{,}84458$
- Parte fracionária: 0,8440
- $0{,}844\times16\approx13{,}504$
- Índice = 13

36:

- $36\times0{,}618\approx22{,}24836$
- Parte fracionária: 0,2480
- $0{,}248\times16\approx3{,}9680$
- Índice = 3

33:

- $33\times0{,}618\approx20{,}39433$
- Parte fracionária: 0,3940
- $0{,}394\times16\approx6{,}304$
- Índice = 6

14:

- $14\times0{,}618\approx8{,}65214$
- Parte fracionária: 0,6520
- $0{,}652\times16\approx10{,}4320{,}652$
- Índice = 10

81:

- 81×0,618≈50,05881
- Parte fracionária: 0,0580
- 0,058×16≈0,9280
- Índice = 0

55:

- 55×0,618≈33,9955
- Parte fracionária: 0,990
- 0,99×16≈15,84
- Índice = 15

16:

- 16×0,618≈9,88816
- Parte fracionária: 0,8880
- 0,888×16≈14,208
- Índice = 14

26:

- 26×0,618≈16,06826
- Parte fracionária: 0,0680
- 0,068×16≈1,0880
- Índice = 1

51:

- 51×0,618≈31,51851
- Parte fracionária: 0,5180
- 0,518×16≈8,288
- Índice = 8

39:

- 39×0,618≈24,10239
- Parte fracionária: 0,1020
- 0,102×16≈1,632
- Índice = 1

Mapeamento final:

Índice 0: 81

Índice 1: 26 → e depois 39 também mapeia para 1 (conflito)

Índice 3: 36

Índice 6: 33

Índice 8: 51

Índice 10: 14

Índice 11: 61

Índice 13: 58

Índice 14: 16

Índice 15: 55

Os índices 2, 4, 5, 7, 9, 12 ficam vazios.

Observação: Em particular, o índice 1 recebeu dois mapeamentos (26 e 39) e, sem tratamento de colisão, apenas um ficará (ou ocorreria conflito).