Universidade Eduardo Mondlane
Faculdade de Engenharia
Departamento de Engenharia Eletrotécnica
Engenharia Informática

Estrutura de Dados e Algoritmos – EDA

Por:

- **Dr. Alfredo Covele**
- **Eng. Cristiliano Maculuve**

Janeiro 2022

Tópicos

No.	Designação
1	**Hashing** ❑ Tabelas de Comprovação Complementar ❑ Técnica de Transformação de Chaves

Recapitulação

- Até agora vimos basicamente tipos de estruturas de dados para o armazenamento flexível de dados: **Listas e Árvores**.
  - Cada um desses grupos possui muitas variantes.
- As **Listas** são simples de se implementar. Mas, com um tempo médio de acesso T = n/2, são impraticáveis para grandes quantidades de dados.
  - Em uma lista com 100.000 dados, para recuperar 3 elementos em sequência, faremos um número esperado de 150.000 acessos a nodos;
  - listas são óptimas para pequenas quantidades de dados.
- As **Árvores** são estruturas mais complexas, mas que possuem um tempo médio de acesso T= $\log_G n$, onde G = grau da árvore.
  - A organização de uma árvore depende da ordem de entrada dos dados;
  - para evitar a deterioração, há modelos de árvores que se reorganizam sozinhas. As principais são **AVL** e **Árvore B**.

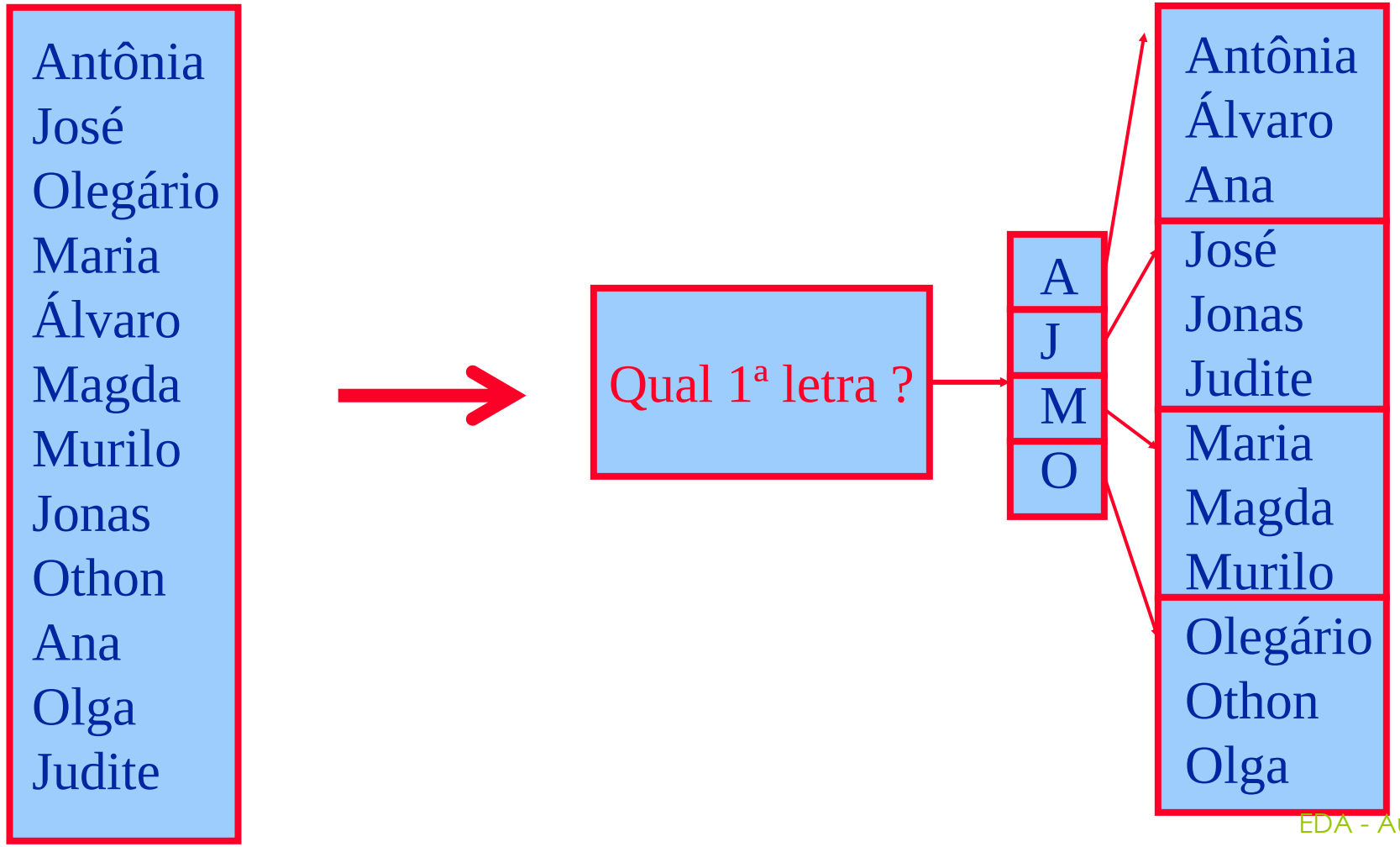
Hashing: visão geral

- É uma forma extremamente simples, fácil de se implementar e intuitiva de se organizar grandes quantidades de dados.
  - Permite armazenar e encontrar rapidamente dados por chave.
- Possui como idéia central a divisão de um universo de dados a ser organizado em subconjuntos mais gerenciáveis.
- Possui dois conceitos centrais:
  - **Tabela de Hashing**: **estrutura que permite o acesso aos subconjuntos**;
  - **Função de Hashing: função que realiza um mapeamento entre valores de chaves e entradas na tabela**.
- Possui uma série de limitações em relação às árvores:
  - não permite recuperar/imprimir todos os elementos em ordem de chave nem outras operações que exijam sequência dos dados;
  - não permite operações do tipo recuperar o elemento com a maior ou a menor chave.

Hashing: introdução

- Idéia geral: se eu possuo um universo de dados classificáveis por chave, posso:
  - criar um critério simples para dividir esse universo em subconjuntos com base em alguma qualidade do domínio das chaves;
  - saber em qual subconjunto procurar e colocar uma chave;
  - gerenciar esses subconjuntos bem menores por algum método simples.
- Para isso eu preciso:
  - saber quantos subconjuntos eu quero e criar uma regra de cálculo que me diga, dada uma chave, em qual subconjunto devo procurar pelos dados com essa chave ou colocar esse dado, caso seja um novo elemento. Isto é chamado de **função de hashing;**
  - possuir um índice que me permita encontrar o início do subconjunto certo, depois de calcular o hashing. Isto é a **tabela de hashing;**
  - possuir uma ou um conjunto de estruturas de dados para os subconjuntos. Existem duas filosofias: **hashing fechado** ou de ***endereçamento aberto*** ou o **hashing aberto** ou ***encadeado***.

Exemplo


Antônia
José
Olegário
Maria
Álvaro
Magda
Murilo
Jonas
Othon
Ana
Olga
Judite
Qual 1ª letra ?
A
J
M
O
Antônia
Álvaro
Ana
José
Jonas
Judite
Maria
Magda
Murilo
Olegário
Othon
Olga

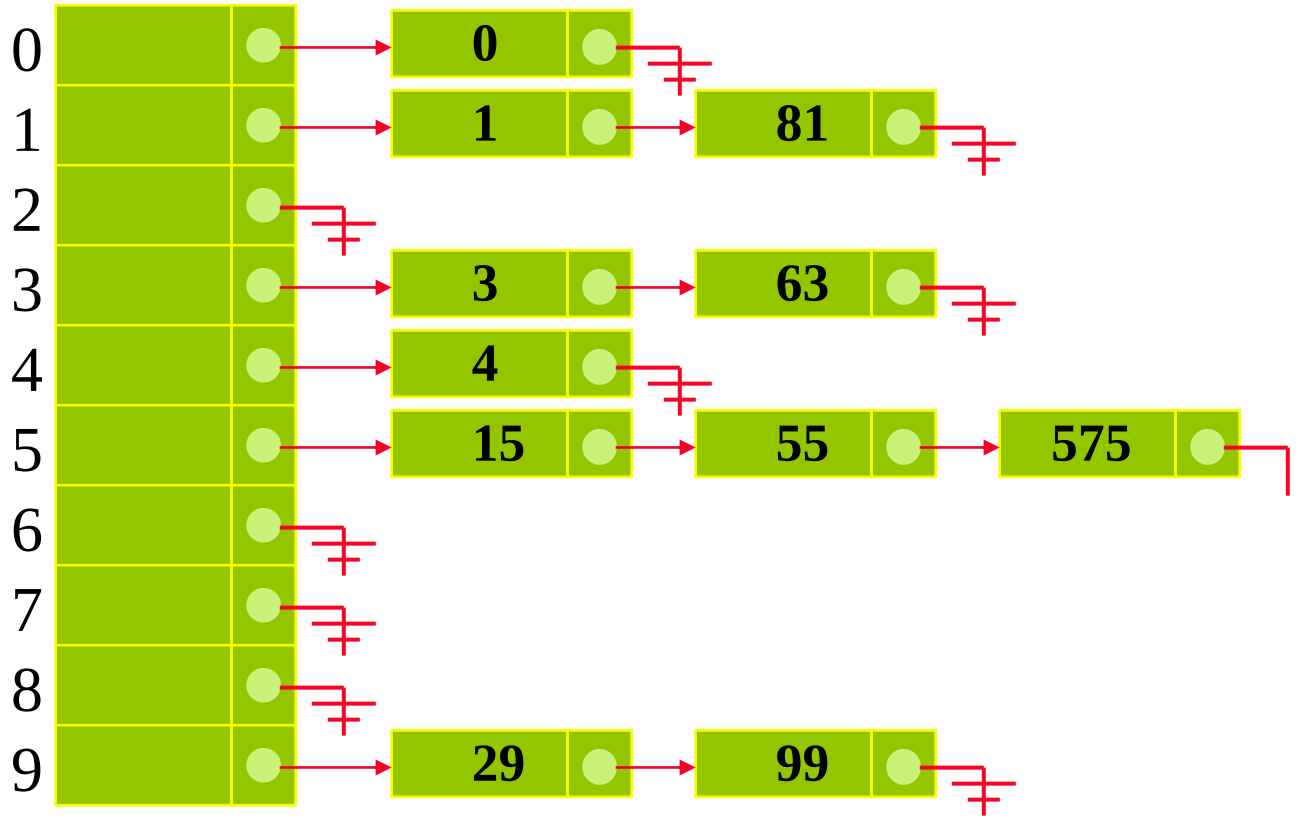
Nomenclatura

- ***n***: tamanho do universo de dados;
- ***b***: número de subconjuntos em que dividimos os dados – ***depósitos***;
- ***s***: capacidade de cada depósito (quando for limitada. Aplica-se somente a hashing fechado ou endereçamento aberto);
- ***T***: cardinalidade do domínio das chaves. Quantas chaves podem existir?
- ***n/T***: densidade identificadora;
- $\alpha = n/(b.s)$: densidade de carga. ***b.s*** fornece a capacidade máxima (quando existir explicitamente). ***n/(b.s)*** indica o factor de preenchimento. Aplica-se somente a hashing fechado (endereçamento aberto).

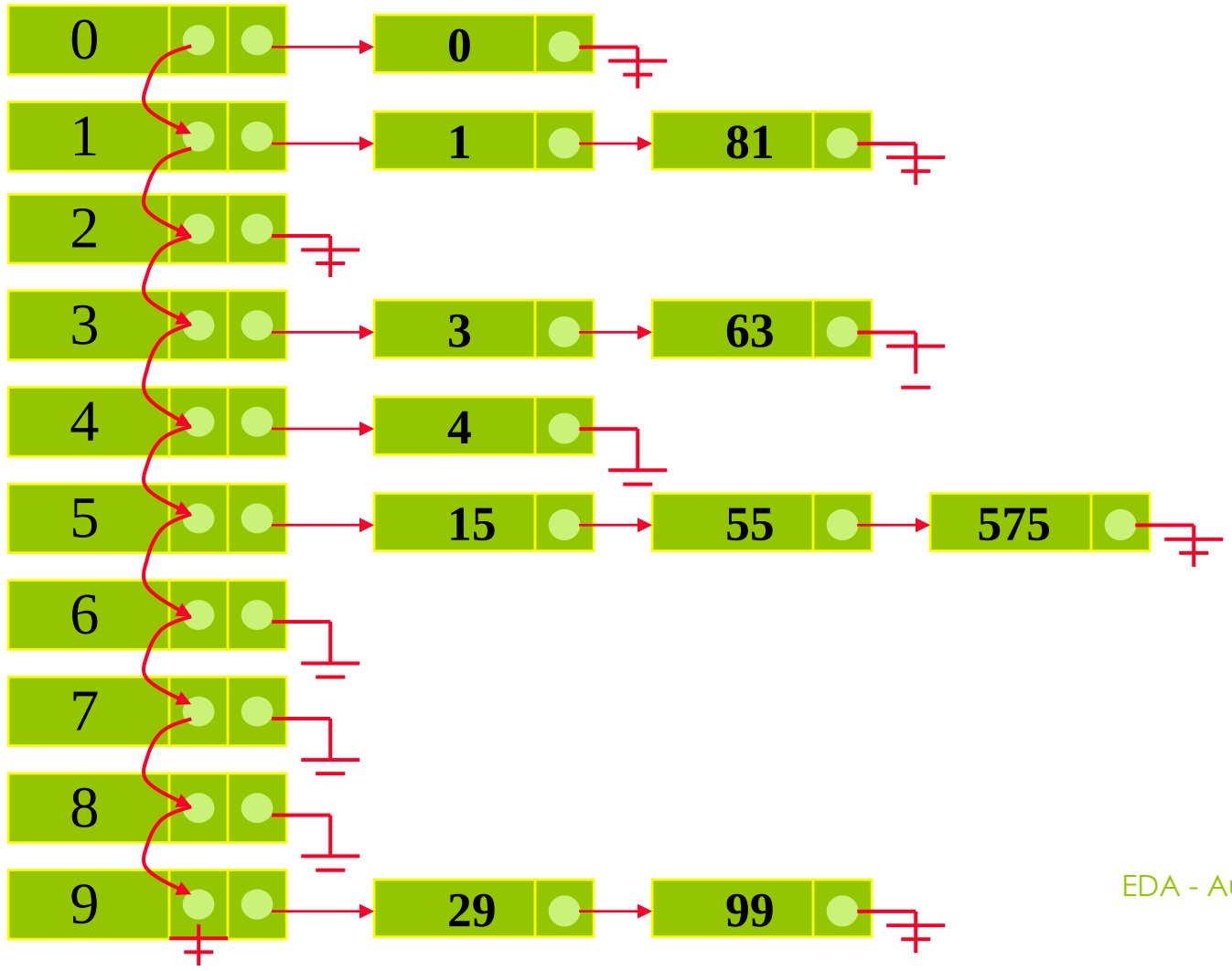
Hashing Aberto ou de Encadeamento Separado (*Separate Chaining Hashing*)

- Forma mais intuitiva de se implementar o conceito de Hashing:
  - intuitiva para nós, que usamos linguagens que lidam com ponteiros.
- Utiliza a idéia de termos uma tabela com ***b*** entradas, cada uma como cabeça de lista para uma lista representando o conjunto b_i.
  - Calculamos a partir da chave qual entrada da tabela é a cabeça da lista que queremos;
  - utilizamos uma técnica qualquer para pesquisa dentro de b_i. Tipicamente será a técnica de pesquisa sequencial em lista encadeada;
  - podemos utilizar qualquer outra coisa para representar os b_i. Uma árvore poderia ser uma opção.

Hashing Aberto ou de Encadeamento Separado (Implementação com a tabela como vector)



Hashing Aberto ou de Encadeamento Separado (Implementação como multilista)

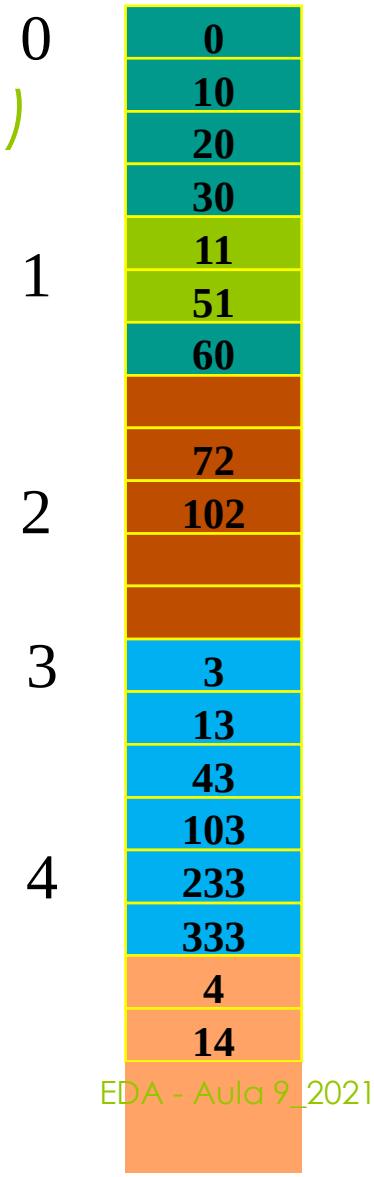


Hashing Fechado ou de Endereçamento Aberto (*Open Addressing Hashing*)

- Utiliza somente uma estrutura de dados de tamanho fixo para implementar o hashing:
  - não necessita de ponteiros para a sua implementação;
  - forma muito utilizada "antigamente";
  - adequada para implementação em disco (*ISAM*).
- Somente uma tabela dividida em partes iguais:
  - áreas de tamanho fixo para cada b_i : ***repositórios***;
  - cada repositório é examinado sequencialmente na busca por uma chave;
  - quando um repositório está cheio, há ***estouro***.
- Filosofias para tratamento de estouros:
  - utilização do primeiro espaço vazio;
  - busca quadrática.

Hashing Fechado ou de Endereçamento Aberto - Implementação com busca sequencial após estouro (Suponha *h(k)* como *mod(k,10)*)

- Manipulação de estouro:
  - usamos o primeiro espaço livre;
  - no caso do 60, isto foi logo após duas entradas em "1";
  - no caso dos valores com chave apontando para "3", foram incluídos dois valores antes mesmo de ser incluído algo com chave "hasheada" para "4".
- Critério de parada de busca:
  - consideramos o arquivo inteiro como uma lista circular e paramos ao chegar de novo ao ponto de partida;
  - somente aí consideramos uma chave como não existente.



Complexidade

- A ordem de complexidade de tempo de uma implementação de hashing é linear ***O(n)*** e ***T(n)*** compõe-se de três termos:
  1. o tempo para calcular a função de hashing;
  2. o tempo para encontrar o início do depósito indicado pela função de hashing através da tabela de hashing;
  3. o tempo para encontrar a chave procurada dentro do seu depósito.
- Essas complexidades são diferentes para as duas filosofias de implementação de hashing: aberto ou fechado.

  T(n) vai tender a ser algo em torno de ***n/b*** no caso ideal;
  - variam muito nos casos menos ideais.

Vantagens

- Simplicidade
  - É muito fácil de imaginar um algoritmo para implementar hashing.
- Escalabilidade
  - Podemos adequar o tamanho da tabela de hashing ao ***n*** esperado em nossa aplicação.
- Eficiência para ***n*** grandes
  - Para trabalharmos com problemas envolvendo ***n*** = 1.000.000 de dados, podemos imaginar uma tabela de hashing com 2.000 entradas, onde temos uma divisão do espaço de busca da ordem de ***n***/2.000 de imediato.
- Aplicação imediata a arquivos
  - Os métodos de hashing, tanto de endereçamento aberto quanto fechado, podem ser utilizados praticamente sem nenhuma alteração em um ambiente de dados persistentes utilizando arquivos em disco.

Desvantagens

- Dependência da escolha de função de hashing
  - Para que o tempo de acesso médio ideal $T(n) = c1 . (1/b).n + c2$ seja mantido, é necessário que a função de hashing divida o universo dos dados de entrada em ***b*** conjuntos de tamanho aproximadamente igual.
- Tempo médio de acesso é ótimo somente em uma faixa
  - A complexidade linear implica em um crescimento mais rápido em relação a ***n*** do que as árvores, por exemplo.
  - Existe uma faixa de valores de ***n***, determinada por ***b***, onde o hashing será muito melhor do que uma árvore.
  - Fora dessa faixa é pior.

Função de Hashing

- Possui o objetivo de transformar o valor de chave de um elemento de dados em uma posição para esse elemento em um dos ***b*** subconjuntos definidos.
  - É um ***mapeamento*** de ***K* -> {1,..,*b*}**, onde ***K* = {k_0,...,k_m}** é o conjunto de todos os valores de chave possíveis no universo de dados em questão.
- Deve dividir o universo de chaves ***K* = {k_0,...,k_m}** em ***b*** subconjuntos de mesmo tamanho.
  - A probabilidade de uma chave $k_j \in K$ aleatória qualquer cair em um dos subconjuntos $b_i : i \in [1,b]$ deve ser **uniforme**;
  - se a função de Hashing não dividir ***K*** uniformemente entre os b_i, a tabela de hashing pode degenerar:
    - o pior caso de degeneração é aquele onde todas as chaves caem em um único conjunto b_i.
  - A função "primeira letra" do exemplo anterior é um exemplo de uma função ruim:
    - a letra do alfabeto com a qual um nome inicia não é distribuída uniformememente. Quantos nomes começam com "X"?

Funções de Hashing

- Para garantir a distribuição uniforme de um universo de chaves entre ***b*** conjuntos, foram desenvolvidas 4 técnicas principais.
  - Lembre-se: a função de hashing $h(k_j) \rightarrow [1,b]$ toma uma chave $k_j \in \{k_0,\ldots,k_m\}$ e devolve um número i, que é o índice do subconjunto $b_i : i \in [1,b]$ onde o elemento possuidor dessa chave vai ser colocado;
  - as funções de hashing abordadas adiante supõem sempre uma chave simples, um único dado, seja string ou número, sobre o qual será efetuado o cálculo.
    - Hashing sobre mais de uma chave, por exemplo "Nome" E "CPF" também é possível, mas implica em funções mais complexas.
- Principais Funções de Hashing:
  - Divisão
  - Meio do Quadrado
  - Folding ou Desdobramento
  - Análise de Dígitos

Divisão

- Forma mais simples e mais utilizada para função de hashing;
- o endereço de um elemento na tabela de hashing é dado simplesmente pelo resto da divisão da sua chave por ***b***: $h(k_j) = mod(k_j, b) + 1$.
  - O termo "+ 1" é para numeração de $b_i : i \in [1, b]$ a partir de 1.
- Funciona se a distribuição de valores de chave k_j for uniforme em seu domínio ***K***:
  - se as chaves, mesmo sendo numéricas, possuem uma regra de formação que faz com que estejam irregularmente distribuídas em seu domínio, o resto da divisão também não gera distribuição uniforme.
  - uma ajuda sugerida é a utilização somente de números primos de valores altos (acima de 20) como valores para ***b***.
- Para chaves alfanuméricas pode-se utilizar a soma de todos os valores numéricos dos caracteres da chave como base:
  - antigamente utilizava-se simplesmente a representação binária do string como "numero". É mau porque para valores de ***b*** pequenos, toda a parte "mais alta" do string é ignorada.

Divisão

- Exemplo:

Suponha b=1.000 e a sequinte sequência de chaves [*Villas et.al.*]:

1.030, 839, 10.054 e 2030.

Teremos a seguinte distribuição:

Chave	Endereço
1.030	30
10.054	54
839	839
2.030	30

- Observe que 1.030 e 2.030 geram o mesmo endereço.
  - A isto chama-se **colisão**.

Meio do quadrado

- Calculada em dois passos:
  - eleva-se a chave ao quadrado;
  - utiliza-se um determinado número de dígitos ou bits do meio do resultado.
- Idéia geral:
  - A parte central de um número elevado ao quadrado depende dele como um todo.
- Quando utilizamos diretamente bits:
  - se utilizarmos ***r*** bits do meio do quadrado, podemos utilizar o seu valor para acessar diretamente uma tabela de 2^r entradas.

Folding ou desdobramento

- **Método para cadeias de caracteres:**
  - inspirado na idéia de se ter uma tira de papel e de se dobrar essa tira formando um "bolinho" ou "sanfona";
  - baseia-se em uma representação numérica de uma cadeia de caracteres.
    - Pode ser binária ou ASCII.
- **Dois tipos:**
  - *Shift Folding;*
  - *Limit Folding*.

Shif Folding

- Divido um string em pedaços, onde cada pedaço é representado numericamente e soma as partes.
  - Exemplo mais simples: somar o valor ASCII de todos os caracteres;
  - o resultado uso diretamente ou como chave para uma ***h´(k).***
- **Exemplo:**
  - Suponha que os valores ASCII de uma string sejam os seguintes: 123, 203, 241, 112 e 20.
  - O *folding* será:

```
  123
  203
  241
  112
   20
 ----
  699
```

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

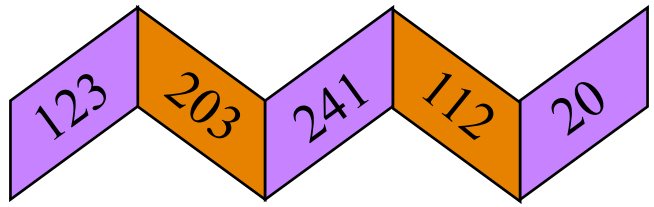
Limit Folding

- Uso a idéia da tira de papel como sanfona:
  - exemplo mais simples: somar o valor ASCII de todos os caracteres, invertendo os dígitos a cada segundo caracter.

Exemplo:

- Suponha que os valores ASCII de um string sejam os seguintes: 123, 203, 241, 112 e 20
- O *folding* será:

```
  123
  302
  241
  211
   20
  ---
  897
```



Análise de dígitos

- Pode ser utilizada quando eu conheço as distribuições dos dígitos ou caracteres das chaves:
  - é útil quando temos um domínio com poucas chaves ou com chaves com regra de formação bem conhecida.
- Escolhemos uma parte da chave para cálculo do Hashing:
  - essa parte deverá ter uma distribuição conhecida;
  - essa distribuição deverá ser a mais uniforme possível.
- Exemplo:
  - sabemos que o DDD e os 4 primeiros dígitos de um número telefônico não são distribuídos de forma uniforme: *não servem*.
    - Diferentes estados possuem número diferente de telefones e a maioria das cidades começa seus números com X222.
  - Podemos supor que os 4 últimos dígitos de um número telefônico são distribuídos mais ou menos uniformemente: *boa opção*.

Exercício: implementação da classe *Dictionary*

- A classe Dicionário em Smalltalk comporta-se como uma lista indexada:
  - na realidade esta é a forma como uma instância de *Dictionary* se mostra para o usuário;
  - um *Dictionary* é implementado internamente como uma tabela de Hashing, para melhor tempo de acesso.
- Você vai implementar uma ***Classe Dicionário*** em JAVA que:
  - utilize hashing encadeado e um vetor como tabela de hashing;
  - utilize Strings como chaves;
  - possua uma função de hashing que utilize ***shift folding*** sobre o string-chave e ***mod(k,17)*** sobre o resultado do folding;
  - seja um dicionário genérico, sendo capaz, em teoria, de armazenar objetos de qualquer tipo, através de ponteiros **`void`**.
    Na prática você vai implementá-lo para inteiros, floats e strings.

Modelagem do *Dictionary*

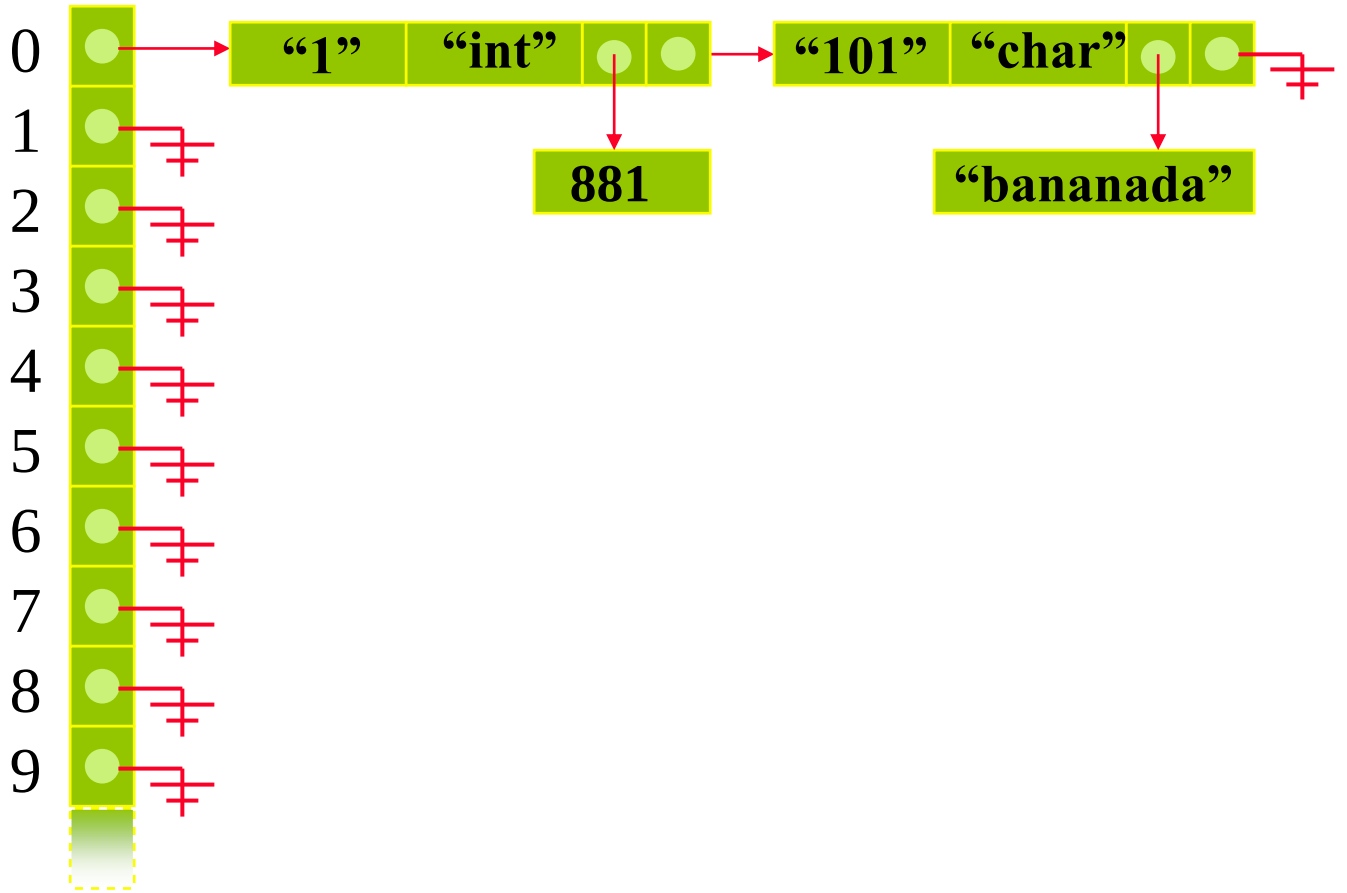
- Você vai precisar de 2 classes:
  - o Dicionário em si, implementado como um vetor de ponteiros para Associações;
  - a Associação, que associa uma chave a um objeto.
    - A Associação conterá um campo para a chave (ponteiro para char), um campo para o objeto associado (ponteiro void, para objetos genéricos), um campo tipo (definido por nós como inteiro) e um ponteiro para a próxima associação;
    - o campo tipo é necessário para que saibamos qual é o tipo de objeto que é apontado pela associação, já que C++ não associa informações sobre pertinência de classe a objetos.

Modelagem das classes Dicionário e Associação

```
Classe Dicionário {
    Associação dados[17];
}

Classe Associação {
    caracter *chave;
    inteiro tipo;
    void *valor;
    Associação *prox;
}
```

Modelagem da classe *Dictionary*



Implementação da Classe *Dictionary*

- O Dicionário deverá ser capaz de:
  - adicionar um novo objeto, lendo a chave, o tipo e o objeto e adicionando-o;
  - recuperar e imprimir um objeto a partir da chave. Lembre-se que em C++ para isto é necessário que, ao imprimir, primeiro seja verificado qual o tipo para que seja chamada a impressão correta;
  - imprimir todas as chaves. Essa impressão deverá ser da forma mais simples possível, sem se preocupar com a ordem;
  - lembre-se de fazer um ***typecasting*** ao imprimir, depois que descobrir qual é o tipo do objeto apontado pelo ponteiro ***valor***.

Por hoje é tudo