



SCC-0224 - Capítulo 3

Métodos de Busca

João Luís Garcia Rosa¹

¹Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
<http://www.icmc.usp.br/~joaoluis>

2018

Sumário

- 1 Busca
 - Introdução
 - Busca Sequencial
 - Busca Binária
- 2 Outros tipos de Busca
 - Busca por interpolação
 - Busca em árvores
- 3 *Hashing*
 - Conceitos
 - Funções *hash*
 - Tipos de *hashing*

Sumário

- 1 Busca
 - Introdução
 - Busca Sequencial
 - Busca Binária
- 2 Outros tipos de Busca
 - Busca por interpolação
 - Busca em árvores
- 3 Hashing
 - Conceitos
 - Funções *hash*
 - Tipos de *hashing*

A importância em estudar busca

- Busca é uma tarefa muito comum em computação?
- Vários métodos e estruturas de dados podem ser empregados para se fazer busca:
 - Quais estruturas de dados?
- Certos métodos de organização/ordenação de dados podem tornar o processo de busca mais eficiente.

A importância em estudar busca

O problema da busca (ou pesquisa):

Dado um conjunto de elementos, onde cada um é identificado por uma chave, o objetivo da busca é localizar, nesse conjunto, o elemento que corresponde a uma chave específica.

Terminologia básica

- **Tabela** ou **Arquivo**: termos genéricos, pode ser qualquer estrutura de dados usada para armazenamento interno e organização dos dados:
 - Uma tabela é um conjunto de elementos, chamados **registros**,
 - Existe uma **chave** associada a cada registro, usada para diferenciar os registros entre si:
 - **Chave interna**: contida dentro do registro, em uma localização específica,
 - **Chave externa**: contida em uma tabela de chaves separada que inclui ponteiros para os registros,
 - **Chave primária**: para todo arquivo existe pelo menos um conjunto exclusivo de chaves - dois registros não podem ter o mesmo valor de chave,
 - **Chave secundária**: chaves não primárias, que não precisam ter seus valores exclusivos. Para que servem?

Terminologia básica

- **Algoritmo de busca:** formalmente, é o algoritmo que aceita um argumento a e tenta encontrar o registro cuja chave seja a ,
- **Operações:**
 - **Inserção:** adicionar um novo elemento à tabela,
 - **Algoritmo de busca e inserção:** se não encontra o registro, insere um novo,
 - **Remoção:** retirar um elemento da tabela,
 - **Recuperação:** procurar um elemento na tabela e, se achá-lo, torná-lo disponível.

Terminologia básica

- A tabela pode ser:
 - Um vetor de registros,
 - Uma lista encadeada,
 - Uma árvore,
 - Etc.
- A tabela pode ficar:
 - Totalmente na memória (busca interna),
 - Totalmente no armazenamento auxiliar (busca externa),
 - Dividida entre ambos.

Tipos de busca

- As técnicas de busca em memória interna que estudaremos serão:
 - Busca Sequencial,
 - Busca Binária,
 - Busca por Interpolação,
 - Busca em Árvores,
 - *Hashing*.
- O objetivo é encontrar um dado registro com o **menor custo**,
- Cada técnica possui **vantagens** e **desvantagens**.

Sumário

- 1 Busca
 - Introdução
 - Busca Sequencial
 - Busca Binária
- 2 Outros tipos de Busca
 - Busca por interpolação
 - Busca em árvores
- 3 Hashing
 - Conceitos
 - Funções *hash*
 - Tipos de *hashing*

Busca sequencial

- A busca sequencial é a forma mais simples de busca,
- É aplicável a uma tabela organizada como um vetor ou como uma lista encadeada,
- Busca mais simples que há,
- Percorre-se registro por registro em busca da chave:

1							N=8
12	25	33	37	48	57	86	92

Busca sequencial

Procure por 48

1							N=8
12	25	33	37	48	57	86	92

Busca sequencial

Procure por 48

1							N=8
12	25	33	37	48	57	86	92
↑							

Busca sequencial

Procure por 48

1							N=8
12	25	33	37	48	57	86	92
	↑						

Busca sequencial

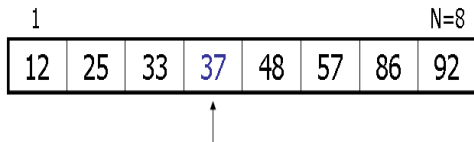
Procure por 48

1							N=8
12	25	33	37	48	57	86	92
		↑					

Busca sequencial

Procure por 48

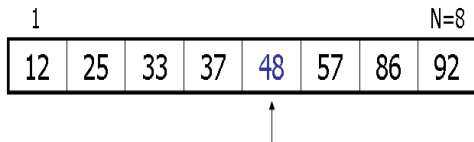
1							N=8
12	25	33	37	48	57	86	92



Busca sequencial

Procure por 48

1							N=8
12	25	33	37	48	57	86	92



Busca sequencial

- Algoritmo de busca sequencial em um vetor A , com n posições (0 até $n - 1$), sendo x a chave procurada:

```
for (i=0; i<n; i++)  
    if (A[i]==x)  
        return(i); /*chave encontrada*/  
return(-1); /*chave não encontrada*/
```

Busca sequencial

- Uma maneira de tornar o algoritmo mais eficiente é usar um **sentinela**:
 - Sentinela: consiste em adicionar um elemento de valor x no final da tabela.
- O sentinela garante que o elemento procurado será encontrado, o que elimina uma expressão condicional, melhorando a performance do algoritmo:

```
A[n]=x;  
for (i=0; x!=A[i]; i++);  
if (i < n) return(i); /*chave encontrada*/  
else return(-1); /*chave não encontrada*/
```

Busca sequencial

- Limitações do vetor:
 - Tamanho fixo:
 - Pode desperdiçar ou faltar espaço.
- Alternativa:
 - Lista encadeada:
 - O que muda na busca sequencial?
- Complexidade:
 - Se o registro for o primeiro: 1 comparação,
 - Se o registro procurado for o último: N comparações,
 - Se for igualmente provável que o argumento apareça em qualquer posição da tabela, em média: $\frac{(n+1)}{2}$ comparações,
 - Se a busca for mal sucedida: N comparações.
 - Logo, a busca sequencial, no pior caso, é $\mathcal{O}(n)$.

Busca sequencial

- Arranjo não ordenado:
 - Inserção no final do arranjo,
 - Remoção:
 - Realocação dos registros acima do registro removido.
- Para aumentar a eficiência:
 - Reordenar continuamente a tabela de modo que os registros mais acessados sejam deslocados para o início:
 - 1 **Método mover-para-frente**: sempre que uma pesquisa obtiver êxito, o registro recuperado é colocado no início da lista,
 - 2 **Método da transposição**: um registro recuperado com sucesso é trocado com o registro imediatamente anterior.
 - Ambos se baseiam no fenômeno da **recuperação recorrente de registros**.

Busca sequencial

- Desvantagens do método mover-para-frente:
 - Uma única recuperação não implica que o registro será frequentemente recuperado:
 - Perda de eficiência para os outros registros.
 - O método é mais “caro” em vetores do que em listas.
- Vantagens do método mover-para frente:
 - Possui resultados melhores para quantidades pequena e média de buscas.

Busca sequencial

- Busca sequencial em tabela ordenada:
 - A eficiência da operação de busca melhora se as chaves dos registros estiverem ordenadas:
 - No pior caso (caso em que a chave não é encontrada), são necessárias N comparações quando as chaves estão desordenadas,
 - No caso médio, $\frac{N}{2}$ comparações se as chaves estiverem ordenadas, pois se para a busca assim que uma chave maior do que a procurada é encontrada.
 - Dificuldade do método?

Busca sequencial

- Busca sequencial indexada:
 - Existe uma tabela auxiliar, chamada **tabela de índices**, além do próprio arquivo ordenado,
 - Cada elemento na tabela de índices contém uma chave (*index*) e um indicador do registro no arquivo que corresponde a *index*:
 - Faz-se a busca a partir do ponto indicado na tabela, sendo que a busca não precisa ser feita desde o começo.
 - Pode ser implementada como um vetor ou como uma lista encadeada:
 - O indicador da posição na tabela pode ser um ponteiro ou uma variável inteira.

Busca sequencial

Tabela de índices

index

321	
592	
876	
...	

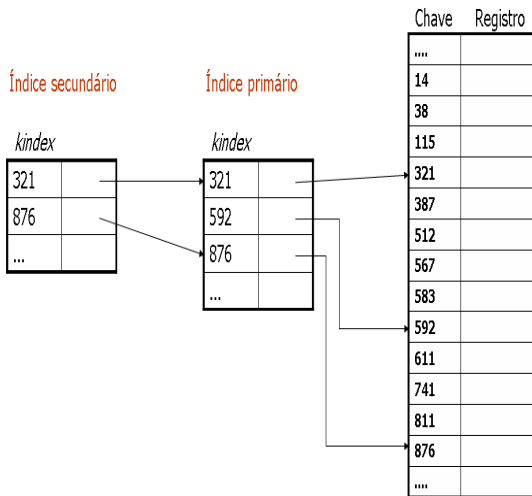
Chave Registro

....	
14	
38	
115	
321	
387	
512	
567	
583	
592	
611	
741	
811	
876	
....	

Busca sequencial

- Busca sequencial indexada:
 - Se a tabela for muito grande, pode-se ainda usar a **tabela de índices secundária**:
 - O índice secundário é um índice para o índice primário.

Busca sequencial



Busca sequencial

- Vantagem:
 - Os itens na tabela poderão ser examinados sequencialmente sem que todos os registros precisem ser acessados:
 - O tempo de busca diminui consideravelmente.
- Desvantagens:
 - A tabela tem que estar ordenada,
 - Exige espaço adicional para armazenar a(s) tabela(s) de índices.
- Algo mais?

Busca sequencial

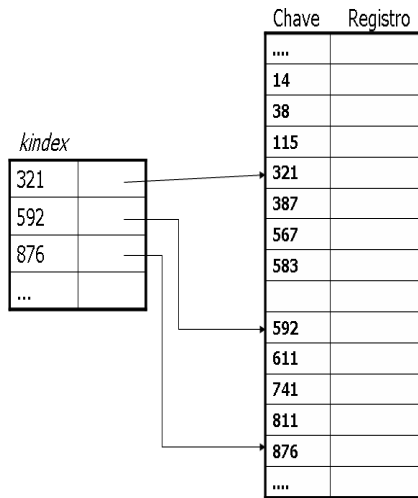
- Remoção:
 - Remove-se o elemento e rearranja-se a tabela inteira e $O(s)$ índice(s),
 - Marca-se a posição do elemento removido, indicando que ela pode ser ocupada por um outro elemento futuramente:
 - A posição da tabela fica vazia.

Busca sequencial

- Inserção:
 - Se houver espaço vago na tabela, rearranjam-se os elementos localmente,
 - Se não houver espaço vago:
 - Rearranjar a tabela a partir do ponto apropriado e reconstruir o(s) índice(s).

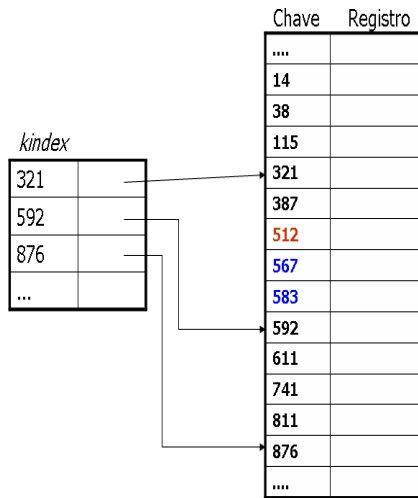
Busca sequencial

- Inserção do elemento 512 **com** espaço vago:
 - 567 e 583 descem,
 - 512 é inserido.



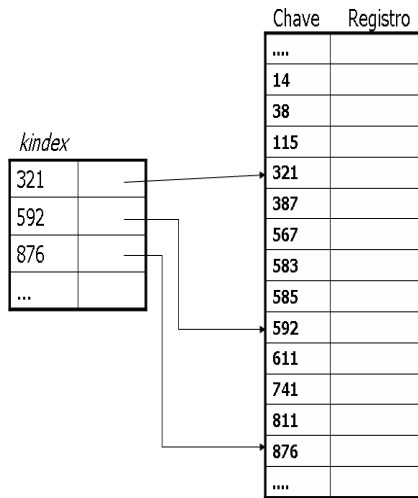
Busca sequencial

- Inserção do elemento 512 **com** espaço vago:
 - 567 e 583 descem,
 - 512 é inserido.



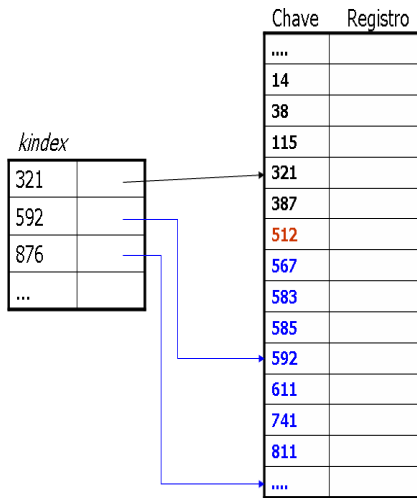
Busca sequencial

- Inserção do elemento 512 **sem** espaço vago:
 - Elementos a partir de 567 descem,
 - 512 é inserido,
 - Índice é reconstruído.



Busca sequencial

- Inserção do elemento 512 **sem** espaço vago:
 - Elementos a partir de 567 descem,
 - 512 é inserido,
 - Índice é reconstruído.



Busca sequencial

- Como montar o índice primário:
 - Se a tabela não estiver ordenada, ordene-a,
 - Divide-se o número de elementos da tabela pelo tamanho do índice desejado: $n/\text{tamanho-índice}$,
 - Para montar o índice, recuperam-se da tabela os elementos 0, $0+n/\text{tamanho-índice}$, $0+2*n/\text{tamanho-índice}$, etc.,
 - Cada elemento do índice representa $n/\text{tamanho-índice}$ elementos da tabela.

Busca sequencial

- Exemplo:
 - Divide-se o número de elementos da tabela pelo tamanho do índice desejado:
 - Se a tabela tem 1.000 elementos e deseja-se um índice primário de 10 elementos, faz-se $1.000/10=100$.
 - Para montar o índice, recuperam-se da tabela os elementos 0, $0+n/\text{tamanho-índice}$, $0+2*n/\text{tamanho-índice}$, etc.:
 - O índice primário é montado com os elementos das posições 0, 100, 200, etc. da tabela.
 - Cada elemento do índice representa $n/\text{tamanho-índice}$ elementos da tabela:
 - Cada elemento do índice primário aponta para o começo de um grupo de 100 elementos da tabela.

Busca sequencial

- Para montar um índice secundário, aplica-se raciocínio similar sobre o índice primário,
- Em geral, não são necessários mais do que 2 índices.

Sumário

- 1 Busca
 - Introdução
 - Busca Sequencial
 - Busca Binária
- 2 Outros tipos de Busca
 - Busca por interpolação
 - Busca em árvores
- 3 Hashing
 - Conceitos
 - Funções *hash*
 - Tipos de *hashing*

Busca binária

- Se os dados estiverem ordenados em um arranjo, pode-se tirar vantagens dessa ordenação:
 - Busca binária: $A[i] \leq A[i + 1]$, se ordem crescente;
 $A[i] \geq A[i + 1]$, se ordem decrescente.
- O elemento buscado é comparado ao elemento do meio do arranjo:
 - Se igual, busca bem-sucedida,
 - Se menor, busca-se na metade inferior do arranjo,
 - Se maior, busca-se na metade superior do arranjo.

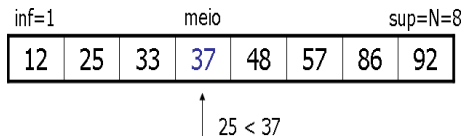
Busca binária

- Busca-se por 25:

inf=1				sup=N=8			
12	25	33	37	48	57	86	92

Busca binária

- Busca-se por 25:



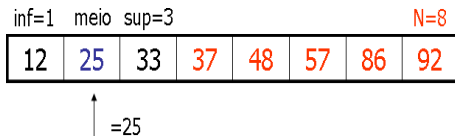
Busca binária

- Busca-se por 25:

inf=1			sup=3				N=8
12	25	33	37	48	57	86	92

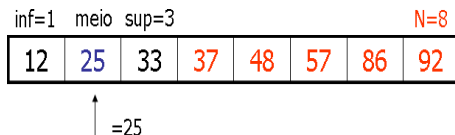
Busca binária

- Busca-se por 25:



Busca binária

- Busca-se por 25:



- Em cada passo, o tamanho do arranjo em que se busca é dividido por 2.

Busca binária

- **Complexidade:**

- $\mathcal{O}(\log n)$, pois cada comparação reduz o número de possíveis candidatos por um fator de 2.

- **Vantagens:**

- Eficiência da busca,
- Simplicidade da implementação.

- **Desvantagens:**

- Nem todo arranjo está ordenado,
- Exige o uso de um arranjo para armazenar os dados:
 - Faz uso do fato de que os índices do vetor são inteiros consecutivos.
- Inserção e remoção de elementos são ineficientes:
 - Realocação de elementos.

Busca binária

- A busca binária pode ser usada com a organização de tabela sequencial indexada:
 - Em vez de pesquisar o índice sequencialmente, pode-se usar uma busca binária.

Sumário

- 1 Busca
 - Introdução
 - Busca Sequencial
 - Busca Binária
- 2 Outros tipos de Busca
 - Busca por interpolação
 - Busca em árvores
- 3 Hashing
 - Conceitos
 - Funções *hash*
 - Tipos de *hashing*

Busca por interpolação

- Se as chaves estiverem uniformemente distribuídas, esse método pode ser ainda mais eficiente do que a busca binária,
- Com chaves uniformemente distribuídas, pode-se esperar que x esteja aproximadamente na posição:

$$meio = inf + (sup - inf) * \left(\frac{x - A[inf]}{A[sup] - A[inf]} \right)$$

sendo que inf e sup são redefinidos iterativamente como na busca binária.

Busca por interpolação

- **Complexidade:**

- $\mathcal{O}(\log(\log(n)))$ se as chaves estiverem uniformemente distribuídas:
 - Raramente precisará de mais comparações.
- Se as chaves não estiverem uniformemente distribuídas, a busca por interpolação pode ser tão ruim quanto uma busca sequencial.

- **Desvantagem:**

- Em situações práticas, as chaves tendem a se aglomerar em torno de determinados valores e não são uniformemente distribuídas:
 - Exemplo: há uma quantidade maior de nomes começando com “S” do que com “Q”.

Sumário

- 1 Busca
 - Introdução
 - Busca Sequencial
 - Busca Binária
- 2 Outros tipos de Busca
 - Busca por interpolação
 - Busca em árvores
- 3 Hashing
 - Conceitos
 - Funções *hash*
 - Tipos de *hashing*

Busca em árvores

- Árvores:
 - Árvores binárias:
 - Árvores binárias de busca,
 - AVLs: árvores binárias de busca balanceadas.
 - Árvores multidirecionais:
 - Árvores B.
- Complexidade de tempo no pior caso?
- Complexidade de tempo no melhor caso?

Árvore binária de busca

- Uma **árvore binária** é um conjunto finito de nós que ou está vazio ou consiste de um nó raiz e duas árvores binárias disjuntas chamadas de sub-árvores *esquerda* e *direita*,
- Uma **árvore binária de busca** é uma árvore binária que exhibe a propriedade de ordenação. Satisfaz as seguintes propriedades:
 - 1 Todo elemento tem uma chave e não há dois elementos com a mesma chave (i.e., as chaves são distintas),
 - 2 As chaves (se houver) na sub-árvore esquerda são menores do que a chave na raiz,
 - 3 As chaves (se houver) na sub-árvore direita são maiores do que a chave na raiz,
 - 4 As sub-árvores esquerda e direita são também árvores binárias.

Árvore binária de busca

- Uma árvore binária de busca pode suportar as operações de localização, inserção e exclusão,
- Um **algoritmo de busca** é aquele que aceita um argumento a e tenta encontrar o registro cuja chave é a ,
- O algoritmo pode retornar o registro inteiro ou, mais comum, um ponteiro para o registro,
- Uma tabela de registros na qual uma chave é usada para recuperação é chamada de **tabela de busca** ou **dicionário**.

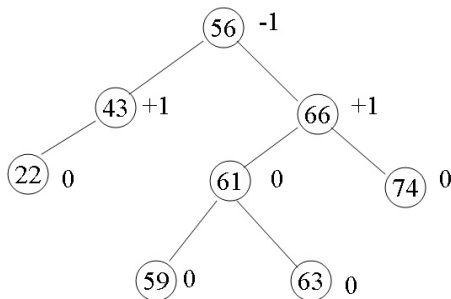
Árvores AVL

- **Árvores AVL**¹ são árvores binárias de busca balanceadas e ordenadas:
 - A árvore é balanceada se a altura da sub-árvore esquerda de qualquer nó não difere de mais de uma unidade da altura da sub-árvore direita,
 - O relacionamento hierárquico prevê que o dado em cada nó da árvore é maior do que todos dados da sua sub-árvore esquerda e menor ou igual aos dados da sub-árvore direita.

¹O termo AVL é devido a proposta de Adelson-Velskii e Landis, de 1962.

Árvores AVL

- Seja a seguinte árvore AVL:



- Fator de equilíbrio = $H_e - H_d$
 - H_e = altura da sub-árvore esquerda,
 - H_d = altura da sub-árvore direita.

Operações com Árvores AVL

- As operações básicas com árvores AVL são as mesmas operações com as árvores binárias de busca:
 - inserir um nó na árvore,
 - localizar um nó na árvore (maior prioridade),
 - remover um nó da árvore.
- O problema é que com a inserção e/ou remoção, a árvore pode deixar de ser AVL,
- Neste caso, deve-se transformá-la.

Árvores AVL

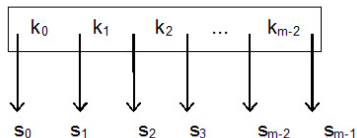
- **Vantagem:** a árvore AVL é uma árvore balanceada, ou seja, a distância média dos nós até a raiz é mínima. Isto contribui para que, na média, os nós sejam localizados mais rapidamente,
- **Aplicação:** tabelas dinâmicas, onde elementos podem ser inseridos e removidos freqüentemente.

Árvore de Busca Multidirecional

- As árvores de busca não binárias dividem-se em:
 - árvore de busca multidirecionais (que inclui árvores-B),
 - árvores de busca digitais.

Árvore de Busca Multidirecional

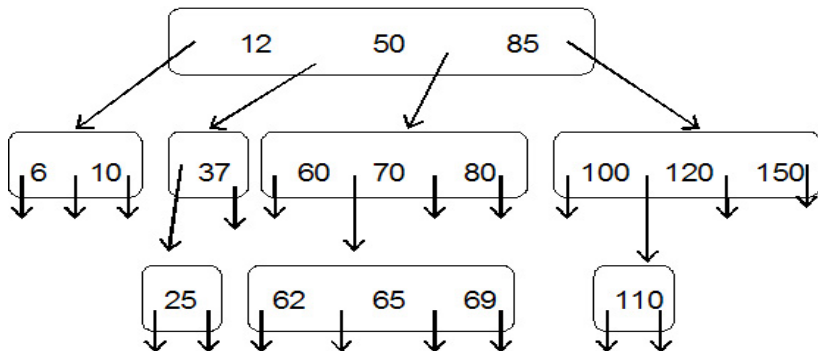
- **Árvore de busca multidirecional de ordem n :**
 - cada nó possui n ou menos sub-árvores e contém uma chave a menos que a quantidade de sub-árvores,
 - Se $s_0, s_1, s_2, \dots, s_{m-1}$ são m sub-árvores de um nó contendo as chaves k_0, k_1, \dots, k_{m-2} , em ordem crescente,



- todas as chaves em s_0 serão menores ou iguais à k_0 , todas as chaves em s_1 serão menores ou iguais à k_1 e maiores que k_0 e assim por diante,
- uma ou mais sub-árvores de um nó podem estar vazias.

Árvore de Busca Multidirecional

- Exemplo de árvore de ordem 4: 4 sub-árvores no máximo \Rightarrow máximo 3 chaves por nó:



Árvores-B

- Uma **árvore-B** de ordem m é uma **árvore de busca multidirecional balanceada** que satisfaz as seguintes condições:
 - todo nó possui m ou menos sub-árvores (máximo m),
 - todo nó, exceto o raiz e os folhas, possui no mínimo $m/2$ sub-árvores (maior inteiro),
 - o raiz possui no mínimo duas sub-árvores não vazias,
 - todas as folhas estão no mesmo nível,
 - um nó não folha com k sub-árvores armazena $k - 1$ registros,
 - um nó folha armazena no máximo $m - 1$ e no mínimo $m/2$ registros,
 - todos os nós pai (de derivação) possuem exclusivamente sub-árvores não vazias.

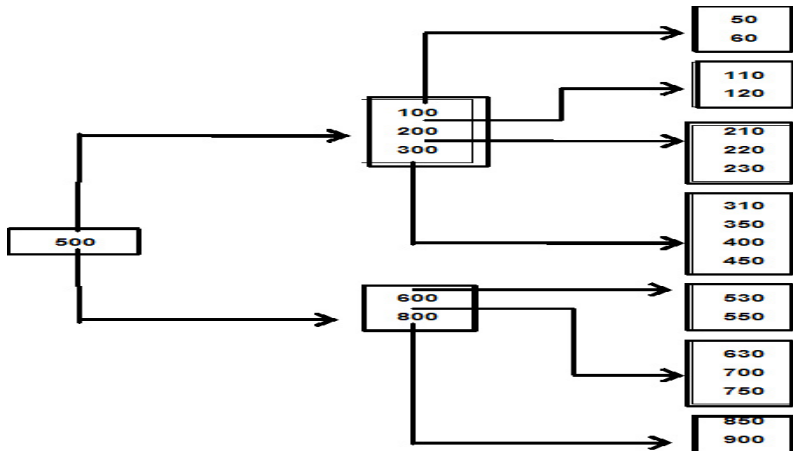
Árvores-B

- Um nó com j registros e $j + 1$ filhos pode ser representado por:

E_0	C_1	E_1	C_2	...	E_{j-1}	C_j	E_j
-------	-------	-------	-------	-----	-----------	-------	-------

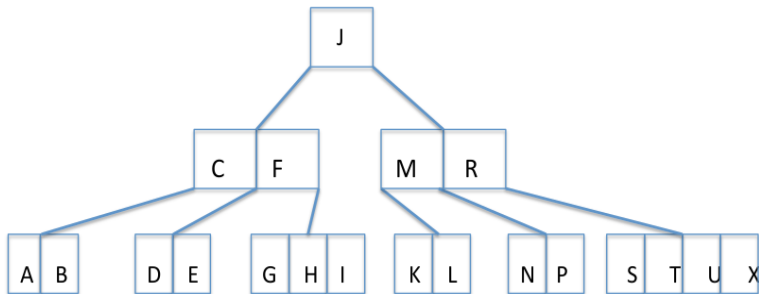
- onde:
 - $C_1 < C_2 < \dots < C_j$ são chaves dos registros,
 - E_i é o endereço da sub-árvore que contém os registros com chaves compreendidas entre C_i e C_{i+1} .
- Exemplo 1 de Árvore-B de ordem 5: próximo slide.

Árvores-B



Árvores-B

- Exemplo 2 de Árvore-B de ordem 5:



Eficiência

- Acesso sequencial = $\mathcal{O}(n)$:
 - Quanto mais as estruturas (tabelas, arquivos, etc.) crescem, mais acessos há,
 - Quando armazenamento é em disco, reduzir acessos é essencial.
- Busca binária = $\mathcal{O}(\log(n))$:
 - Restrita a arranjos.
- Árvores AVL (no melhor caso) = $\mathcal{O}(\log(n))$:
 - Não importa o tamanho da tabela.

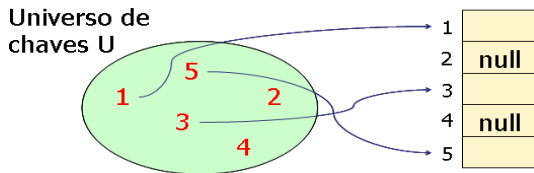
Eficiência

<i>Estrutura de Dados</i>	<i>localização</i>	<i>inserção</i>	<i>remoção</i>
árvore binária de busca	$\mathcal{O}(n)$ (pc) $\mathcal{O}(\log n)$ (cm)	$\mathcal{O}(n)$ (pc) $\mathcal{O}(\log n)$ (cm)	$\mathcal{O}(n)$ (pc) $\mathcal{O}(\log n)$ (cm)
árvore AVL	$\mathcal{O}(\log n)$ (pc)	$\mathcal{O}(\log n)$ (pc)	$\mathcal{O}(\log n)$ (pc)
árvore-B	$\mathcal{O}(\log_{m/2} n)$ (pc)	$\mathcal{O}(\log_{m/2} n)$ (pc)	$\mathcal{O}(\log_{m/2} n)$ (pc)

Resumo de algumas implementações de dicionários. Legenda: pc = pior caso, cm = caso médio, m = ordem da árvore-B [3, 2].

Reflexão

- Acesso em tempo constante:
 - Tradicionalmente, endereçamento direto em um arranjo: Cada chave k é mapeada na posição k do arranjo:
 - Função de mapeamento $f(k) = k$.



Reflexão

- Endereçamento direto:
 - Vantagem: Acesso direto e, portanto, rápido:
 - Via indexação do arranjo.
 - Desvantagem: Uso ineficiente do espaço de armazenamento:
 - Declara-se um arranjo do tamanho da maior chave?
 - E se as chaves não forem contínuas? Por exemplo, {1 e 100},
 - Pode sobrar espaço? Pode faltar?

Reflexão

- *Hashing*:
 - **Acesso direto**, mas **endereçoamento indireto**:
 - Função de mapeamento $h(k) \neq k$, em geral,
 - Resolve uso ineficiente do espaço de armazenamento.
 - Ideal: $\mathcal{O}(1)$, em média, independente do tamanho do arranjo,
 - *Hash* significa (Webster's New World Dictionary):
 - 1 Fazer picadinho de carne e vegetais para cozinhar,
 - 2 Fazer uma bagunça.

Sumário

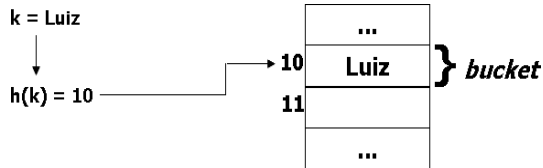
- 1 Busca
 - Introdução
 - Busca Sequencial
 - Busca Binária
- 2 Outros tipos de Busca
 - Busca por interpolação
 - Busca em árvores
- 3 **Hashing**
 - **Conceitos**
 - Funções *hash*
 - Tipos de *hashing*

Hashing: Conceitos e definições

- Também conhecido como tabela de **espalhamento** ou de **dispersão**,
- *Hashing* é uma técnica que utiliza uma função h para transformar uma chave k em um endereço:
 - O endereço é usado para armazenar e recuperar registros.
- Ideia: particionar um conjunto de elementos (possivelmente infinito) em um número finito de classes:
 - B classes, de 0 a $B - 1$,
 - Classes são chamadas de *buckets*.

Hashing: Conceitos e definições

- Conceitos relacionados:
 - A função h é chamada de função *hash*,
 - $h(k)$ retorna o valor *hash* de k :
 - Usado como endereço para armazenar a informação cuja chave é k .
 - k pertence ao *bucket* $h(k)$.

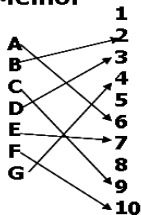


- A função *hash* é utilizada para inserir, remover ou buscar um elemento:
 - Deve ser determinística, ou seja, resultar sempre no mesmo valor para uma determinada chave.

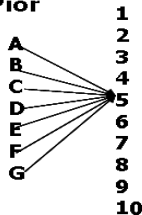
Hashing: Conceitos e definições

- Colisão: ocorre quando a função *hash* produz o mesmo endereço para chaves diferentes:
 - As chaves com mesmo endereço são ditas “sinônimos”.

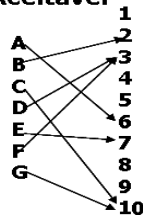
Melhor



Pior



Aceitável



Hashing: Conceitos e definições

- Distribuição uniforme é muito difícil:
 - Dependente de cálculos matemáticos e estatísticos complexos.
- Função que aparente gerar endereços aleatórios:
 - Existe chance de alguns endereços serem gerados mais de uma vez e de alguns nunca serem gerados.
- Existem alternativas melhores que a puramente aleatória,
- **Segredos para um bom *hashing*:**
 - Escolher uma boa função *hash* (em função dos dados):
 - Distribui uniformemente os dados, na medida do possível:
Hash uniforme.
 - Evita colisões,
 - É fácil/rápida de computar.
 - Estabelecer uma boa estratégia para tratamento de colisões.

Exemplo de função *hash*

- Técnica simples e muito utilizada que produz bons resultados:
 - Para chaves inteiras, calcular o resto da divisão k/B ($k\%B$), sendo que o resto indica a posição de armazenamento:
 - k = valor da chave, B = tamanho do espaço de endereçamento.
 - Para chaves tipo string, tratar cada caractere como um valor inteiro (ASCII), somá-los e pegar o resto da divisão por B ,
 - B deve ser primo, preferencialmente.

Sumário

- 1 Busca
 - Introdução
 - Busca Sequencial
 - Busca Binária
- 2 Outros tipos de Busca
 - Busca por interpolação
 - Busca em árvores
- 3 **Hashing**
 - Conceitos
 - **Funções *hash***
 - Tipos de *hashing*

Exemplo de função *hash*

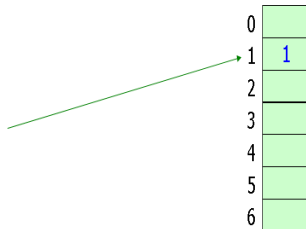
- Exemplo:
 - Seja B um arranjo de 7 elementos:
 - Inserção dos números 1, 5, 10, 20, 25, 24.

0	
1	
2	
3	
4	
5	
6	

Exemplo de função *hash*

- Exemplo:
 - Seja B um arranjo de 7 elementos:
 - Inserção dos números 1, 5, 10, 20, 25, 24.

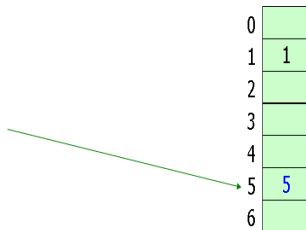
$$1 \% 7 = 1$$



Exemplo de função *hash*

- Exemplo:
 - Seja B um arranjo de 7 elementos:
 - Inserção dos números 1, 5, 10, 20, 25, 24.

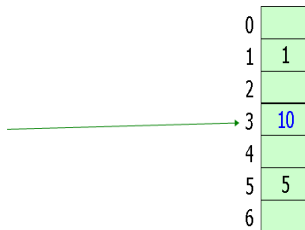
$$5 \% 7 = 5$$



Exemplo de função *hash*

- Exemplo:
 - Seja B um arranjo de 7 elementos:
 - Inserção dos números 1, 5, 10, 20, 25, 24.

$$10 \% 7 = 3$$




0	
1	1
2	
3	10
4	
5	5
6	

Exemplo de função *hash*

- Exemplo:
 - Seja B um arranjo de 7 elementos:
 - Inserção dos números 1, 5, 10, **20**, 25, 24.

$$20 \% 7 = 6$$

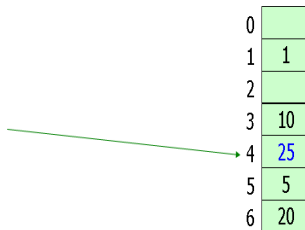


0	
1	1
2	
3	10
4	
5	5
6	20

Exemplo de função *hash*

- Exemplo:
 - Seja B um arranjo de 7 elementos:
 - Inserção dos números 1, 5, 10, 20, **25**, 24.

$$25 \% 7 = 4$$

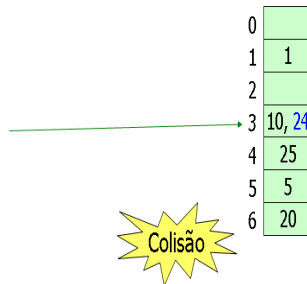


0	
1	1
2	
3	10
4	25
5	5
6	20

Exemplo de função *hash*

- Exemplo:
 - Seja B um arranjo de 7 elementos:
 - Inserção dos números 1, 5, 10, 20, 25, **24**.

$$24 \% 7 = 3$$



Exemplo de função *hash*

- Exemplo com string: mesmo raciocínio:
 - Seja B um arranjo de 13 elementos:
 - $LOWEL = 76\ 79\ 87\ 69\ 76$,
 - $L+O+W+E+L = 387$,
 - $h(LOWEL) = 387 \% 13 = 10$.
 - Qual a idéia por trás da função *hash* que usa o resto?
 - Os elementos sempre caem no intervalo entre 0 e $n - 1$
 - Outras funções *hash*?
 - Como você trataria colisões?

Funções *hash*

- Às vezes, deseja-se que chaves próximas sejam armazenadas em locais próximos:
 - Por exemplo, em um compilador, os identificadores de variáveis *pt* e *pts*.
- Normalmente, não se quer tal propriedade:
 - Questão da aleatoriedade aparente:
 - *Hash* uniforme, com menor chance de colisão.
- Função *hash* escolhida deve espelhar o que se deseja.

Funções *hash*

- **Pergunta:** supondo que se deseja armazenar n elementos em uma tabela de m posições, qual o número esperado de elementos por posição na tabela?
 - **Fator de carga** $\alpha = n/m$.

Sumário

- 1 Busca
 - Introdução
 - Busca Sequencial
 - Busca Binária
- 2 Outros tipos de Busca
 - Busca por interpolação
 - Busca em árvores
- 3 **Hashing**
 - Conceitos
 - Funções *hash*
 - Tipos de *hashing*

Tipos de *Hashing*

1 Estático:

- Espaço de endereçamento não muda,
- Fechado: Permite armazenar um conjunto de informações de tamanho limitado:
 - Técnicas de *rehash* para tratamento de colisões,
 - Overflow progressivo,
 - 2ª. função *hash*.
- Aberto: Permite armazenar um conjunto de informações de tamanho potencialmente ilimitado:
 - Encadeamento de elementos para tratamento de colisões.

2 Dinâmico:

- Espaço de endereçamento pode aumentar.

Hashing estático

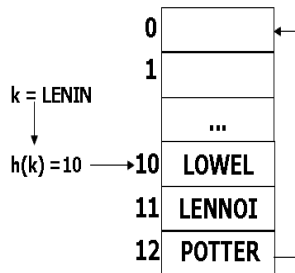
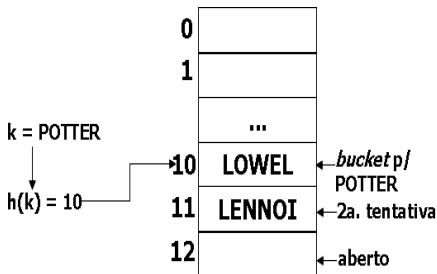
- *Hashing* fechado:
 - Uma tabela de *buckets* é utilizada para armazenar informações:
 - Os elementos são armazenados na própria tabela:
Normalmente conhecido como endereçamento aberto.
 - Colisões: aplicar técnicas de *rehash*:
 - *Overflow* progressivo,
 - 2ª. função *hash*.

Hashing estático

- Técnicas de *rehash*:
 - Se posição $h(k)$ está ocupada (lembre-se de que $h(k)$ é um índice da tabela), aplicar função de *rehash* sobre $h(k)$, que deve retornar o próximo *bucket* livre: $rh(h(k))$,
 - Características de uma boa função de *rehash*:
 - Cobrir o máximo de índices entre 0 e $B - 1$,
 - Evitar agrupamentos de dados.
 - Além de utilizar o índice resultante de $h(k)$ na função de *rehash*, pode-se usar a própria chave k e outras funções *hash*.

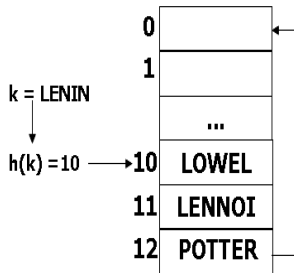
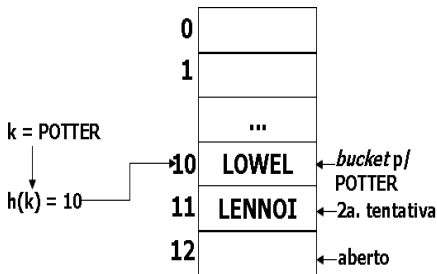
Hashing estático

- Overflow progressivo:
 - $rh(h(k)) = (h(k) + i) \% B$, com i variando de 1 a $B - 1$ (i é incrementado a cada tentativa).



Hashing estático

- Overflow progressivo:
 - $rh(h(k)) = (h(k) + i) \% B$, com i variando de 1 a $B - 1$ (i é incrementado a cada tentativa).



- Como saber que a informação procurada não está armazenada?

Hashing estático

- Exemplo de dificuldade: busca pelo nome “Smith”:

$h(\text{SMITH}) = 7$

	...
7	ADMS
8	JONES
9	MORRIS
10	SMITH

- Pode ter que percorrer muitos campos.

Hashing estático

- Exemplo de dificuldade: busca pelo nome “Smith”:

$h(\text{SMITH}) = 7$

	...
7	ADMS
8	JONES
9	
10	SMITH

- A remoção do elemento no índice 9 pode causar uma falha na busca.

Hashing estático

- Exemplo de dificuldade: busca pelo nome “Smith”:

$h(\text{SMITH}) = 7$

	...
7	ADMS
8	JONES
9	####
10	SMITH

- Solução para remoção de elementos: não eliminar elemento, mas indicar que a posição foi esvaziada e que a busca deve continuar.

Hashing estático

- *Overflow* progressivo:
 - Exemplo anterior: $rh(h(k)) = (h(k) + i) \% B$, com $i = 1 \dots B - 1$:
 - Chamada **sondagem linear**, pois todas as posições da tabela são checadas, no pior caso.
 - Outro exemplo: $rh(h(k)) = (h(k) + c_1 * i + c_2 * i^2) \% B$, com $i = 1 \dots B - 1$ e constantes c_1 e c_2 :
 - Chamada **sondagem quadrática**, considerada melhor do que a linear, pois evita “mais” o agrupamento de elementos.

Hashing estático

- *Overflow* progressivo:
 - Vantagem:
 - Simplicidade.
 - Desvantagens:
 - 1 Agrupamento de dados (causado por colisões),
 - 2 Com estrutura cheia a busca fica lenta,
 - 3 Dificulta inserções e remoções.
- } **overflow progressivo**
- } **hashing fechado**

Hashing estático

- 2ª. função *hash*, ou *hash* duplo:
 - Uso de 2 funções:
 - $h(k)$: função *hash* primária,
 - $haux(k)$: função *hash* secundária.
 - Exemplo: $rh(h(k)) = (h(k) + i * haux(k)) \% B$, com $i = 1 \dots B - 1$.
 - Algumas boas funções:
 - $h(k) = k \% B$,
 - $haux(k) = 1 + k \% (B - 1)$.

Hashing estático

- 2ª. função *hash*, ou *hash* duplo:
 - Vantagem:
 - Evita agrupamento de dados, em geral. Por quê?
 - Desvantagens:
 - Difícil achar funções *hash* que, ao mesmo tempo, satisfaçam os critérios de cobrir o máximo de índices da tabela e evitem agrupamento de dados,
 - Operações de buscas, inserções e remoções são mais difíceis.

Hashing estático

- Alternativamente, em vez de fazer o *hashing* utilizando uma função *hash* e uma função de *rehash*, podemos representar isso em uma única função dependente do número da tentativa (i),
- Por exemplo: $h(k, i) = (k + i) \% B$, com $i = 0 \dots B - 1$:
 - A função h depende agora de dois fatores: a chave k e a iteração i ,
 - Note que $i = 0$ na primeira execução, resultando na função *hash* tradicional de divisão que já conhecíamos,
 - Quando $i = 1 \dots B - 1$, já estamos aplicando a função de *rehash* de sondagem linear.

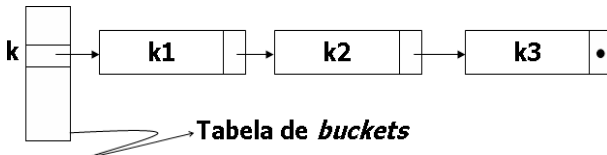
Hashing estático

- Exercício: implemente uma sub-rotina de inserção utilizando função *hash* anterior:

```
#define B 100
#define h(k,i) (k+i)%B
int inserir(int T[], int k)
{
    int i, j;
    for (i=0; i<B; i++)
    {
        j=h(k,i);
        if (T[j]==-1)
        {
            T[j]=k;
            return(j);
        }
    }
    return(-1) //tabela já está cheia
}
```

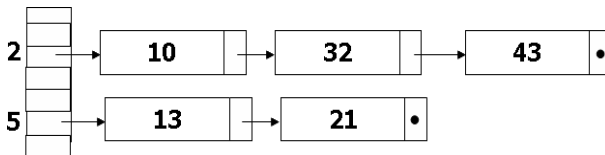
Hashing estático

- *Hashing* aberto:
 - A tabela de *buckets*, indo de 0 a $B - 1$, contém apenas **ponteiros para uma lista de elementos**,
 - Quando há colisão, o sinônimo é inserido no *bucket* como um novo nó da lista,
 - Busca deve percorrer a lista.



Hashing estático

- Se as listas estiverem ordenadas, reduz-se o tempo de busca:
 - Dificuldade deste método?



Hashing estático

- Vantagens:
 - A tabela pode receber mais itens mesmo quando um bucket já foi ocupado,
 - Permite percorrer a tabela por ordem de valor *hash*.
- Desvantagens:
 - Espaço extra para as listas,
 - Listas longas implicam em muito tempo gasto na busca:
 - Se as listas estiverem ordenadas, reduz-se o tempo de busca,
 - Custo extra com a ordenação.

Hashing estático: Eficiência

- Hashing fechado:
 - Depende da técnica de *rehash*:
 - Com *overflow* progressivo, após várias inserções e remoções, o número de acessos aumenta.
 - A tabela pode ficar cheia,
 - Pode haver mais espaço para a tabela, pois não são necessários ponteiros e campos extras como no *hashing* aberto.
- *Hashing* aberto:
 - Depende do tamanho das listas e da função *hash*:
 - Listas longas degradam desempenho,
 - Poucas colisões implicam em listas pequenas.

Algumas boas funções *hash*

- Divisão:
 - $h(k) = k \% m$, com m tendo um tamanho primo, de preferência.
- Multiplicação:
 - $h(k) = (k * A \% 1) * m$, com A sendo uma constante entre 0 e 1:
 - $(k * A \% 1)$ recupera a parte fracionária de $k * A$,
 - Knuth sugere $A = \frac{\sqrt{5}-1}{2} = 0,6180 \dots$

Algumas boas funções *hash*

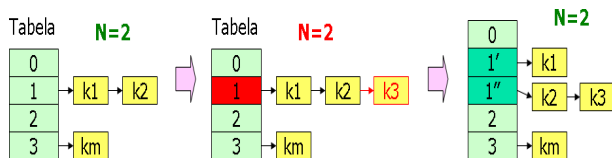
- *Hash* universal:
 - A função *hash* é escolhida aleatoriamente no início de cada execução, de forma que minimize/evite tendências das chaves,
 - Por exemplo, $h(k) = ((A * k + B) \% P) \% m$:
 - P é um número primo maior do que a maior chave k ,
 - A é uma constante escolhida aleatoriamente de um conjunto de constantes $\{0, 1, 2, \dots, P - 1\}$ no início da execução,
 - B é uma constante escolhida aleatoriamente de um conjunto de constantes $\{1, 2, \dots, P - 1\}$ no início da execução.
 - Diz-se que h representa uma coleção de funções universal.

Hashing

- *Hash* perfeito:
 - Quando não há colisão:
 - Aplicável em um cenário em que o conjunto de chaves é estático,
 - Exemplo de cenário deste tipo?
 - Exemplo de *hash* perfeito:
 - *Hashing* em 2 níveis,
 - Uma primeira função *hash* universal é utilizada para encontrar a posição na tabela, sendo que cada posição da tabela contém uma outra tabela (ou seja, outro arranjo),
 - Uma segunda função *hash* universal é utilizada para indicar a posição do elemento na nova tabela.

Hashing dinâmico

- O tamanho do espaço de endereçamento (número de *buckets*) pode aumentar,
- Exemplo de *hashing* dinâmico:
 - *Hashing* extensível: Conforme os elementos são inseridos na tabela, o tamanho aumenta se necessário:
 - Supondo que o número máximo de elementos por *bucket* é N , sempre que o elemento $N + 1$ surgir, o *bucket* é dividido juntamente com os elementos.

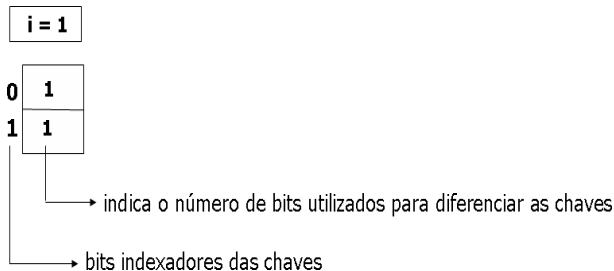


Hashing dinâmico

- *Hashing* extensível:
 - Em geral, trabalha-se com bits,
 - Após $h(k)$ ser computada, uma segunda função f transforma o índice $h(k)$ em uma sequência de bits:
 - Os bits são utilizados para indexar de fato a chave.
 - Alternativamente, h e f podem ser unificadas como uma única função *hash* final.
 - Função *hash* computa sequência de m bits para uma chave k , mas apenas os i primeiro bits ($i \leq m$) do início da sequência são usados como endereço:
 - Se i é o número de bits usados, a tabela de *buckets* terá 2^i entradas,
 - Portanto, tamanho da tabela de *buckets* cresce sempre como potência de 2.
 - N é o número de nós permitidos por *bucket*,
 - Tratamento de colisões: listas encadeadas, em geral.

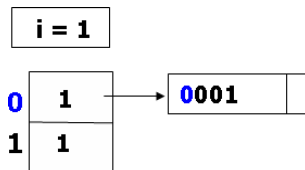
Hashing dinâmico

- Hashing extensível: inicialmente, tabela vazia:
 - $m = 4$ (bits), $N = 2$.



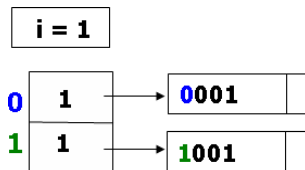
Hashing dinâmico

- Hashing extensível: **inserção do elemento 0001**:
 - $m = 4$ (bits), $N = 2$.



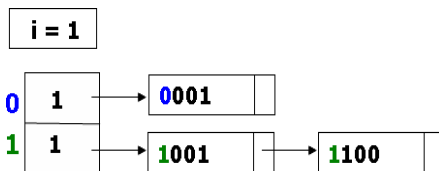
Hashing dinâmico

- Hashing extensível: **inserção do elemento 1001**:
 - $m = 4$ (bits), $N = 2$.



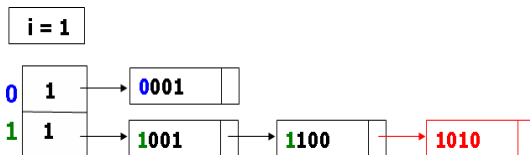
Hashing dinâmico

- Hashing extensível: **inserção do elemento 1100**:
 - $m = 4$ (bits), $N = 2$.



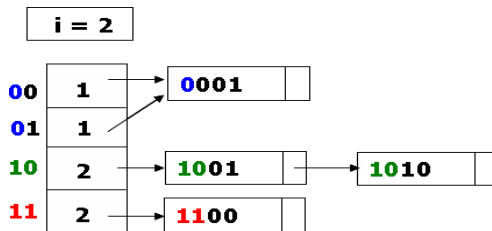
Hashing dinâmico

- Hashing extensível: **inserção do elemento 1010:**
 - $m = 4$ (bits), $N = 2$.
 - N é ultrapassado e a tabela precisa ser rearranjada, pois um único bit não é suficiente para diferenciar os elementos, sendo que o índice em que houve problema tem seu bit incrementado.



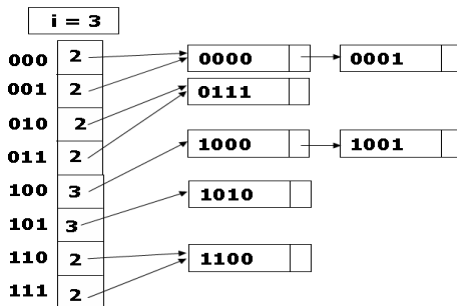
Hashing dinâmico

- Hashing extensível: **rearranjando tabela**:
 - $m = 4$ (bits), $N = 2$.
 - Número de posições (i) aumenta para observar a restrição de N e chaves são rearranjadas.



Hashing dinâmico

- Exemplo: Insira os elementos 0000, 0111 e 1000, nesta ordem:
 - Hashing* extensível: resultado das inserções:



Hashing dinâmico

- **Vantagens:**

- Custo de acesso constante, determinado pelo tamanho de N ,
- A tabela pode crescer.

- **Desvantagens:**

- Complexidade extra para gerenciar o aumento do arranjo e a divisão das listas,
- Podem existir sequências de inserções que façam a tabela crescer rapidamente, tendo, contudo, um número pequeno de registros.

- Quais são as principais desvantagens de *hashing*?

- Os elementos da tabela não são armazenados sequencialmente e nem sequer existe um método prático para percorrê-los em sequência.

Métodos de Busca: Resumo

- Busca sequencial,
- Busca sequencial indexada,
- Busca binária,
- Busca por interpolação,
- Busca em árvores:
 - AVLs.
- *Hashing*.

Métodos de Busca: Resumo

- Critérios para se eleger um (ou mais) método(s):
 - Eficiência da busca,
 - Eficiência de outras operações:
 - Inserção e remoção,
 - Listagem e ordenação de elementos,
 - Outras?
 - Frequência das operações realizadas,
 - Dificuldade de implementação,
 - Consumo de memória (interna),
 - Tempo de acesso a memória externa,
 - Outros?

Bibliografia I

- [1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.
Algoritmos - Teoria e Prática.
Ed. Campus, Rio de Janeiro, Segunda Edição, 2002.
- [2] Knuth, D. E.
The Art of Computer Programming.
Volume 3 - Sorting and Searching. Second Edition.
Addison-Wesley, 1998.
- [3] Horowitz, E., Sahni, S. Rajasekaran, S.
Computer Algorithms.
Computer Science Press, 1998.
- [4] Pardo, Thiago A. S.
Métodos de Busca. SCE-181 Introdução à Ciência da Computação II.
Slides. Ciência de Computação. ICMC/USP, 2008.

Bibliografia II

- [5] Rosa, João Luís G.
ICC II.
Slides. Ciências da Computação. ICMC-USP, 2009.
- [6] Tenenbaum, A. M., Langsam, Y., Augestein, M. J.
Estruturas de Dados Usando C.
Makron Books, 1995.
- [7] Wirth, N.
Algoritmos e Estruturas de Dados.
LTC, 1989.