Estrutura de Dados

Tabela Hash

Agenda

- Introdução
- Função hash
- Resolução de colisões
- Implementação
- Exercícios

Introdução

- Imagine que temos um banco armazendo dados dos usuários de uma aplicação:

ID	Nome
48371	João
19204	Maria
03811	José
…	…

- A aplicação realiza várias consultas para obter o nome do usuário por meio de seu ID
- Usando uma árvore AVL, poderíamos ter um desempenho O(logn) para essas consultas
- Mas e se quisermos ter um desempenho O(1)? Como poderíamos ter este desempenho?

Introdução

- E se quisermos ter um desempenho O(1)?
  - Deveríamos armazenar os dados em um vetor
  - O vetor teria 100.000 posições
  - Várias posições do vetor estariam sendo desperdiçadas
  - Se considerarmos que o campo ID tem 4 bytes e o campo Nome tem 50, gastaríamos ~5MB.
- Agora imagine que existem vários campos (~1MB para cada usuário) e que o ID pode assumir um valor no intervalo de 0 a 4.294.967.295
  - Que tamanho teria que ter o vetor para conseguir armazenar os dados dos usuários?
  - Quanto de espaço seria necessário?

ID	Nome	Login	Senha	…
19204	Maria	ma123	#dlfjnkljdf	…
…	…	…	…	…

Introdução

- Uma alternativa é aproveitar melhor os espaços do vetor
  - Definir faixas para as chaves

0 - 999	1000 - 1999	2000 - 2999	3000 - 3999	…	…	…	…	…	…

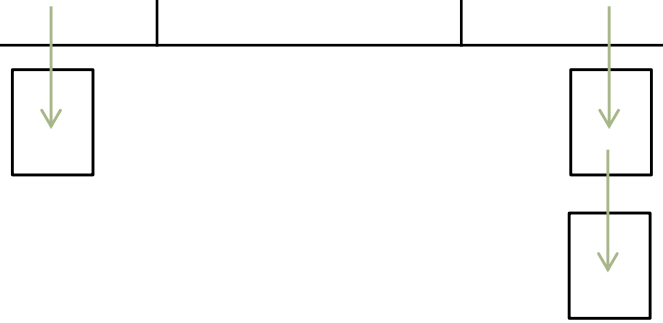
Introdução

- Uma alternativa é aproveitar melhor os espaços do vetor
  - Definir faixas para as chaves

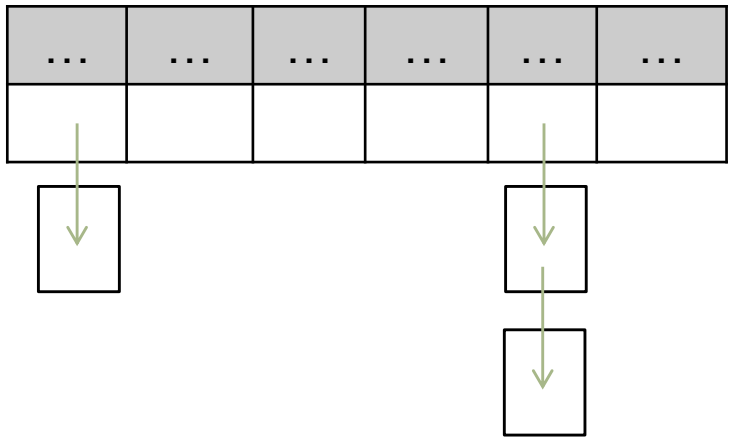
0 - 999	1000 - 1999	2000 - 2999	3000 - 3999	…	…	…	…	…	…

  - Mas como seriam armazenadas chaves dentro de uma mesma faixa?
    - Listas simplesmente encadeadas

0 - 999	1000 - 1999	2000 - 2999	3000 - 3999	…	…	…	…	…	…



Introdução



- Uma Tabela Hash (*hash table*) ou tabela de dispersão é um vetor em que cada uma de suas posições armazena zero, uma, ou mais chaves (e valores associados).

Introdução

- Definir faixas para uma chave sequencial não é a melhor ideia
  - Haveriam várias chaves nos primeiros índices e nenhuma nos últimos índices
- Além disso, há outras alternativas quando acontece o caso de duas ou mais chaves ocuparem um mesmo índice no vetor

- Para isso temos que estudar:
  - Função hash
  - Tratamento de colisões

Função hash

- Função para mapear dados de comprimento variável para dados de comprimento fixo
- Transforma cada chave em um índice da Tabela Hash
- Também chamada de função de espalhamento
  - espalha as chaves pela tabela hash
- Exemplo simples de função hash:

```
int hash (int chave) {
	return abs (chave % M); // M é o tamanho do vetor
}
```

Função hash

- Função hash para chaves não numéricas
  - Há a necessidade de transformar valores não numéricos em números
  - Há várias maneiras de realizar essa tarefa
  - Uma alternativa: uso de pesos previamente gerados para cada caractere da chave

$$\sum_{i=0}^{n-1} chave[i] \times p[i]$$

    - n: número de caracteres
    - chave[i]: um caractere da palavra que representa a chave
    - p[i]: um peso gerado aleatoriamente
  - O resultado do somatório pode ser usado com o operador % para obter o índice no qual a chave será armazenada

Função hash

- Características obrigatórias em uma função hash
  - Retornar um índice dentro do intervalo desejado
  - Gerar sempre o mesmo índice para chaves iguais
- Características desejadas em uma função hash
  - Simples de ser computada
  - Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer

Função hash

- Independente da função hash, há a possibilidade de ocorrer colisões
- As colisões devem ser resolvidas de alguma forma

Resolução de colisões

- Endereçamento aberto
- Listas encadeadas

Resolução de colisões

- Endereçamento aberto
  - Todas as chaves são armazenadas no próprio vetor
  - Caso haja colisão, utiliza uma localização alternativa no mesmo vetor
  - Exemplo: a posição consecutiva livre

Resolução de colisões

- Endereçamento aberto
  - Exemplo: Para uma função hash simples que utiliza apenas o resto como resultado (%) e M = 7, temos:
    - hash (12) = 5
    - hash (21) = 0
    - hash (14) = 0
    - hash (5) = 5
    - hash (19) = 5
  - Várias colisões
  - Para cada colisão, resolver com o endereçameno aberto

Resolução de colisões

- Endereçamento aberto
  - Exemplo:
    - **hash (12) = 5**
    - hash (21) = 0
    - hash (14) = 0
    - hash (5) = 5
    - hash (19) = 5

0	1	2	3	4	5	6
					12	

Resolução de colisões

- Endereçamento aberto
  - Exemplo:
    - hash (12) = 5
    - **hash (21) = 0**
    - hash (14) = 0
    - hash (5) = 5
    - hash (19) = 5

0	1	2	3	4	5	6
21					12	

Resolução de colisões

- Endereçamento aberto
  - Exemplo:
    - hash (12) = 5
    - hash (21) = 0
    - **hash (14) = 0**
    - hash (5) = 5
    - hash (19) = 5

0	1	2	3	4	5	6
21	**14**				12	

    - Colisão! Tentar inserir no próximo índice livre

Resolução de colisões

- Endereçamento aberto
  - Exemplo:
    - hash (12) = 5
    - hash (21) = 0
    - hash (14) = 0
    - **hash (5) = 5**
    - hash (19) = 5

0	1	2	3	4	5	6
21	14				12	**5**

    - Colisão! Tentar inserir no próximo índice livre

Resolução de colisões

- Endereçamento aberto
  - Exemplo:
    - hash (12) = 5
    - hash (21) = 0
    - hash (14) = 0
    - hash (5) = 5
    - **hash (19) = 5**

0	1	2	3	4	5	6
21	14	**19**			12	5

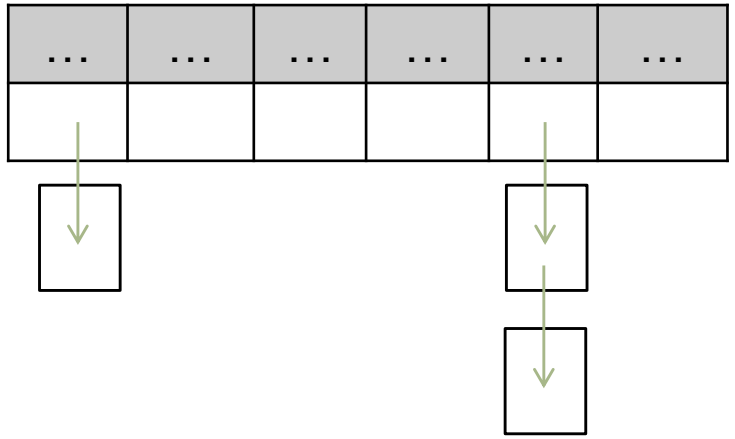
    - Colisão! Tentar inserir no próximo índice livre
    - Chegando ao final do vetor, tentar posição do início do mesmo
    - Procurar posição livre até encontrar um índice vazio

Resolução de colisões

- Endereçamento aberto
  - Desvantagens
    - É necessário prever a quantidade de dados total
      - Melhoria: Redimensionar o vetor quando necessário
        - Recalcular todas as chaves novamente
    - A medida que o vetor for sendo preenchido, os dados ficarão cada vez mais agrupados, diminuindo a eficiência da estrutura
      - Melhoria: ao invés de incrementar 1 na busca por um espaço livre, utilizar uma segunda função hash para incrementar a posição
  - Vantagens
    - Simplicidade
    - Estudos mostram que este método de tratamento de colisões ainda consegue obter bons resultados

Resolução de colisões

- Listas encadeadas
  - Construir uma lista simplesmente encadeada para cada índice do vetor
  - Todas as chaves com o mesmo índice são encadeadas em uma lista



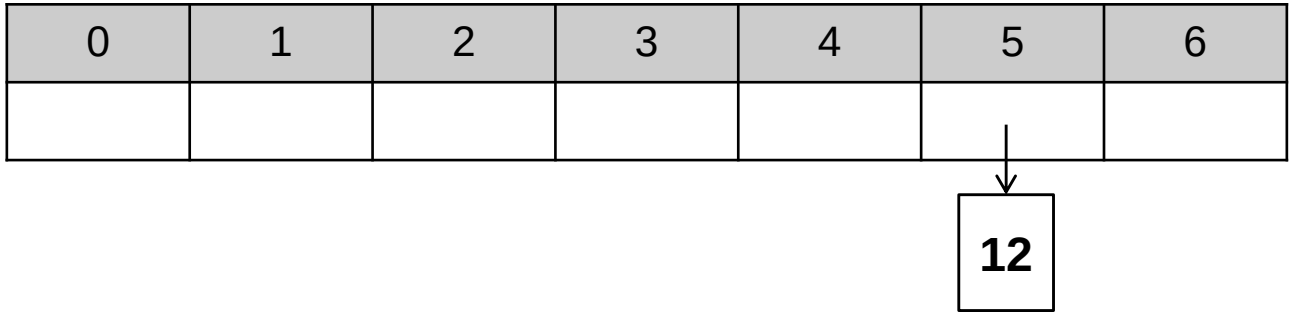
Resolução de colisões

- Listas encadeadas
  - Exemplo
    - hash (12) = 5
    - hash (21) = 0
    - hash (14) = 0
    - hash (5) = 5
    - hash (19) = 5

0	1	2	3	4	5	6

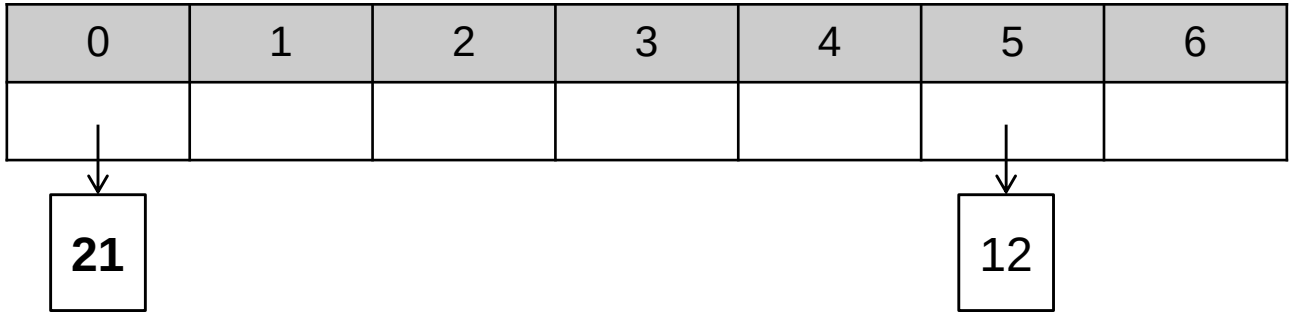
Resolução de colisões

- Listas encadeadas
  - Exemplo
    - **hash (12) = 5**
    - hash (21) = 0
    - hash (14) = 0
    - hash (5) = 5
    - hash (19) = 5



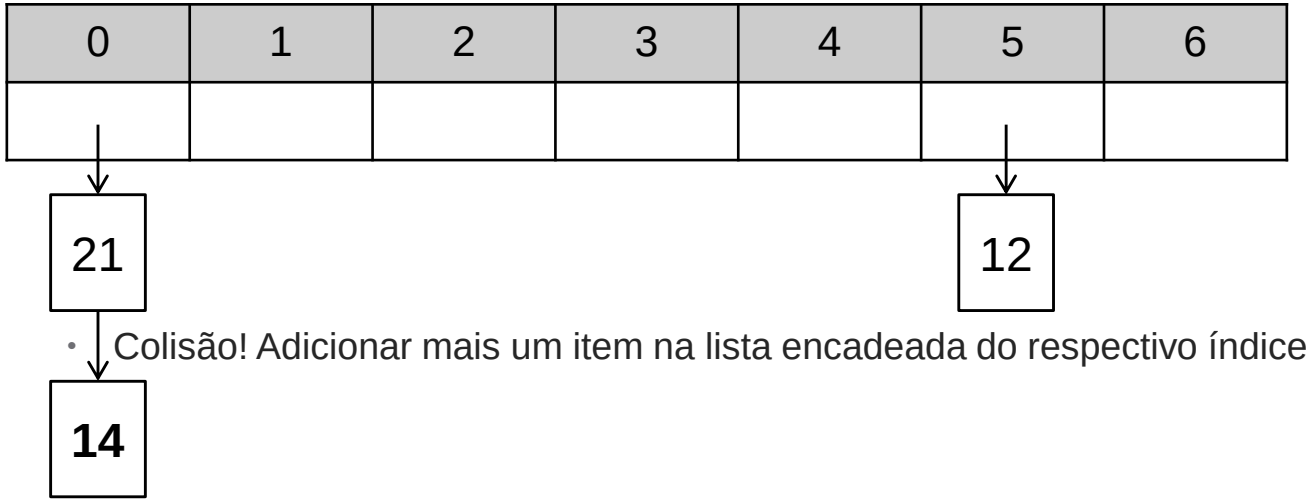
Resolução de colisões

- Listas encadeadas
  - Exemplo
    - hash (12) = 5
    - **hash (21) = 0**
    - hash (14) = 0
    - hash (5) = 5
    - hash (19) = 5



Resolução de colisões

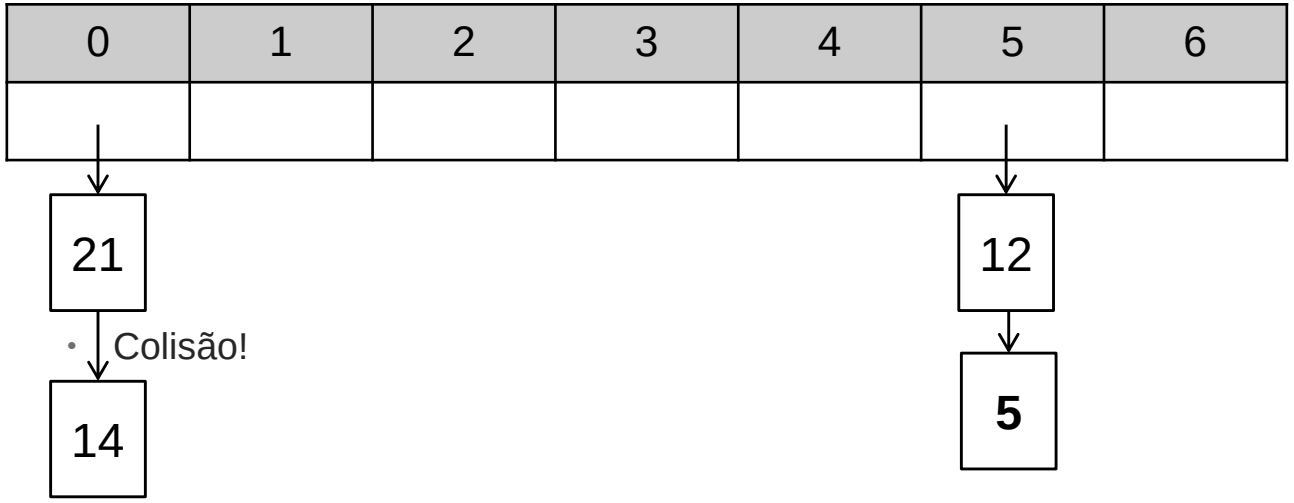
- Listas encadeadas
  - Exemplo
    - hash (12) = 5
    - hash (21) = 0
    - **hash (14) = 0**
    - hash (5) = 5
    - hash (19) = 5



- Colisão! Adicionar mais um item na lista encadeada do respectivo índice

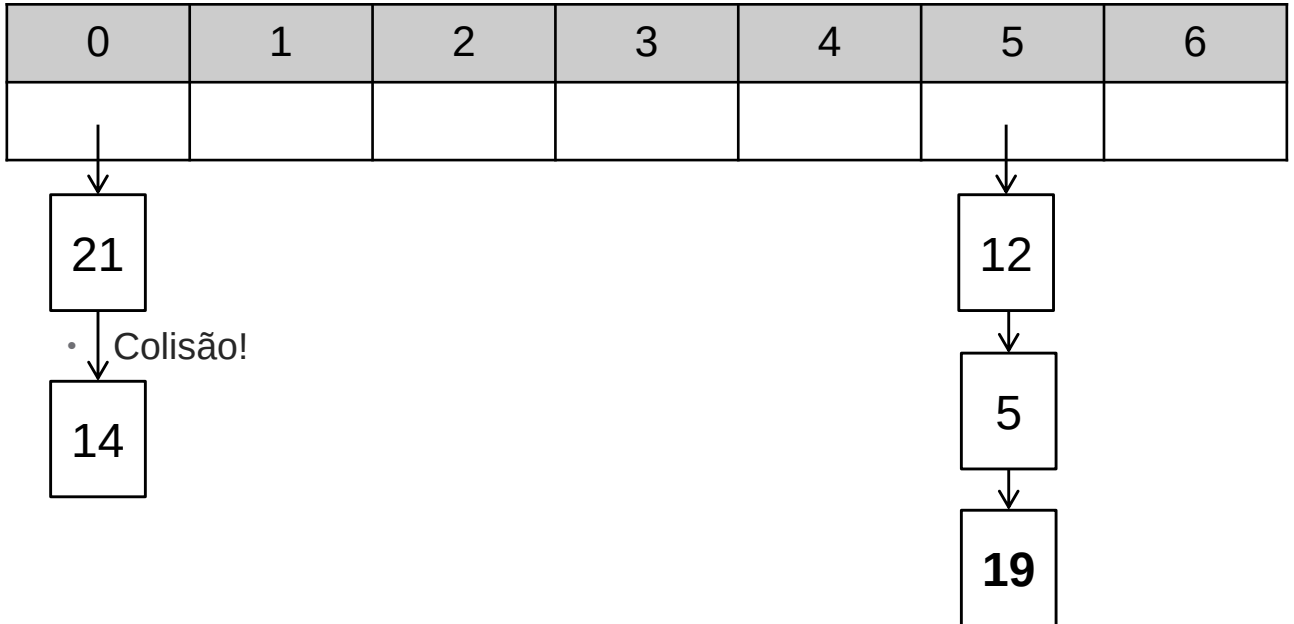
Resolução de colisões

- Listas encadeadas
  - Exemplo
    - hash (12) = 5
    - hash (21) = 0
    - hash (14) = 0
    - **hash (5) = 5**
    - hash (19) = 5



Resolução de colisões

- Listas encadeadas
  - Exemplo
    - hash (12) = 5
    - hash (21) = 0
    - hash (14) = 0
    - hash (5) = 5
    - **hash (19) = 5**



Resolução de colisões

- Listas encadeadas
  - Assumindo que qualquer item do conjunto tem probabilidade igual de ser endereçado, então o comprimento esperado de cada lista encadeada é N/M
    - N representa o número de registros na tabela
    - M representa o tamanho da tabela
  - Se tivermos um conjunto de dados de 10.000 elementos e uma tabela de tamanho 1.000, qual vai ser em média o tamanho de cada lista encadeada?
    - Quantas comparações serão necessárias para encontrar um item nesta Tabela Hash?

Exercícios

- Implemente a tabela hash
  - Insira 10.000 chaves aleatórias
  - Crie uma função para retornar o tamanho de uma lista encadeada
  - Imprima o tamanho de cada lista contida na tabela hash
  - Crie um histograma com o resultado anterior
    - As chaves estão bem distribuídas?
- Crie outra tabela hash para aceitar valores não-numéricos como chave
- Crie outra tabela hash para usar o endereçamento aberto ao invés das listas encadeadas no tratamento de colisões

Implementação

- Estrutura

```
typedef struct {
	int chave;
} TItem;

typedef struct celula {
	struct celula *pProx;
	TItem item;
} TCelula;

typedef struct {
	TCelula *pPrimeiro, *pUltimo;
} TLista;

typedef struct {
	int n;
	int m;
	TLista *v;
} THash;
```

Implementação

- Protótipos

```
void iniciaHash (THash *hash, int m);
int h (THash *hash, int chave);
TItem* pesquisa (THash *hash, int chave);
TCelula* pesquisaCelula (THash *hash, int chave);
int inserir (THash *hash, TItem x);
int remover (THash *hash, int chave);
```

Implementação

- Inicialização e função hash

```
void iniciaHash (THash *hash, int m) {
	int i;
	hash->n = 0;
	hash->m = m;

	hash->v = (TLista*) malloc (sizeof(TLista) * m);

	for (i = 0; i < m; i++)
		iniciaLista (&hash->v[i]);
}

int h (THash *hash, int chave) {
	return chave % hash->m;
}
```

Implementação

- Pesquisa

```
TItem* pesquisa (THash *hash, int chave) {
        TCelula *aux = pesquisaCelula (hash, chave);

        if (aux == NULL)
                return NULL;

        return &(aux->item);
}

TCelula* pesquisaCelula (THash *hash, int chave) {
        int i = h (hash, chave);

        if ( isVazia (&hash->v[i]) )
                return NULL;

        TCelula *aux = hash->v[i].pPrimeiro;
        while (aux->pProx != NULL && chave != aux->item.chave)
                aux = aux->pProx;

        if (chave == aux->item.chave)
                return aux;
        else
                return NULL;
}
```

Implementação

- Inserção e remoção

```
int inserir (THash *hash, TItem x) {
	if (pesquisaCelula (hash, x.chave) == NULL) {
		insereLista (&hash->v[h(hash, x.chave)], x);
		hash->n++;
		return 1;
	}
	return 0;
}

int remover (THash *hash, int chave) {
	TCelula *aux = pesquisaCelula (hash, chave);

	if (aux == NULL)
		return 0;

	removeLista (&hash->v[h(hash, chave)], aux);
	hash->n--;
	return 1;
}
```

Implementação

- Teste

```
int main (void) {
    THash hash;
    iniciaHash (&hash, 97);

    TItem item;
    int i;
    for (i = 0; i < 1000; i++) {
        item.chave = rand() % 100000;
        inserir (&hash, item);
    }

    item.chave = 54109; inserir (&hash, item);
    item.chave = 100; inserir (&hash, item);

    remover (&hash, 54109);

    TItem* p;
    p = pesquisa (&hash, 0);
    printf ("Procurando %d: %s\n", 0, p == NULL ? "não" : "sim");

    p = pesquisa (&hash, 54109);
    printf ("Procurando %d: %s\n", 54109, p == NULL ? "não" : "sim");

    p = pesquisa (&hash, 100);
    printf ("Procurando %d: %s\n", 100, p == NULL ? "não" : "sim");
}
```

Estrutura de Dados

Material elaborado por:
Thiago Meirelles Ventura

Baseado em:

- Ascencio, A. F. G; Araújo, G. S. Estruturas de Dados. Pearson, 2011.
- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. Algoritmos: teoria e prática. Elsevier, 2002.
- Aulas do Prof. Reinaldo Silva Fortes (http://www.decom.ufop.br/reinaldo/)
- Demaine, E., Devadas, S. Introduction to Algorithms (MIT OpenCourseWare), http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011
- http://www.ft.unicamp.br/liag/siteEd/