

Projeto de Arquivos - PRA 0002

Rogério Eduardo da Silva - *rogerio.silva@udesc.br*
Gilmário Barbosa dos Santos - *gilmario.santos@udesc.br*

Universidade do Estado de Santa Catarina
Departamento de Ciência da Computação

4 de maio de 2016

Conteúdo Programático: Apresentação da Disciplina

Revisão de Conceitos

Projeto de Arquivos

Revisão sobre Arquivos em C

Arquivos (TP_1)

Revisão sobre Estruturas de Dados

Listas

Pilhas

Filas

Árvores

Programação orientada a objetos em C++

Documentação de Código - Doxygen

Ordenação Externa

Método: Intercalação

Merge (TP_2)

Método: Árvores Multi-vias

Árvores B (TP_3)

Método: Tabelas de Espalhamento

Hashing (TP_4)

Método de Ensino

- Aulas expositivas em sala e em laboratório
- Listas de exercícios teóricos e práticos
- Atendimento presencial (sala do professor) e/ou através da lista de emails da disciplina tads-pra@googlegroups.com

Avaliações

- 4 trabalhos práticos:

Arquivos criação, manipulação e construção da base de testes (10% da média final);

Intercalação criação de índices com base no método da intercalação (30% da média final);

Árvore B criação de índices com base no método de árvores multi-vias (30% da média final);

Hashing criação de índices com base no método de tabelas de espalhamento (30% da média final);

- Exame Final (caso média semestral < 7.0)

Data prevista: **03 de Julho de 2014**

Bibliografia Básica Sugerida

 Claybrook, B. G. (1983).

Técnicas de Gerenciamento de Arquivos.
Editora Campus.

 Ferraz, I. N. (2003).

Programação com Arquivos.
Editora Manole.

Revisão de Conceitos

- Arquivos em C
- Estruturas de Dados
- Programação Orientada a Objetos (C++)
- Documentação de Códigos (Doxygen)

Projeto de Arquivos - O que trataremos hoje?

Conceitos envolvendo: SO, I/O, blocagem, registros lógicos, registros físicos, abstração de arquivos, tipos de dispositivos de armazenamento, ...

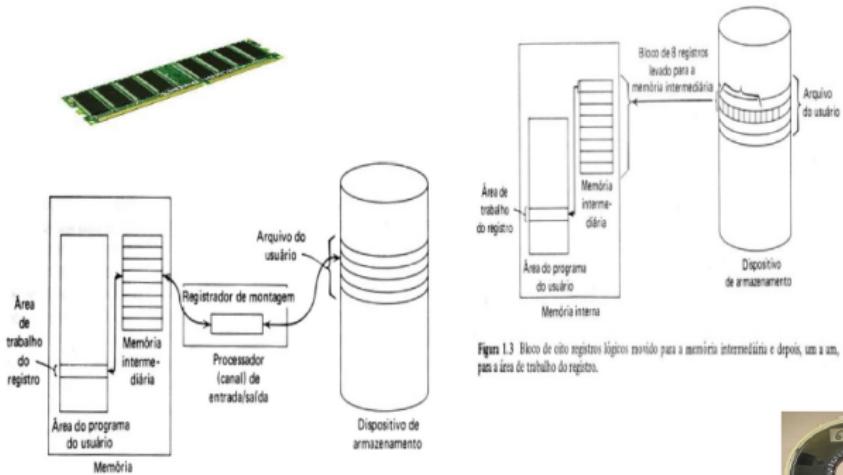


Figura 1.3 Bloco de oito registros lógicos movido para a memória intermediária e depois, um a um, para a área de trabalho do registro.



Fonte: [Claybrook, 1983]

Projeto de Arquivos - Definição

A definição de *projeto*, de acordo com o dicionário Michaelis é

“Plano para a realização de um ato”

Por analogia, podemos dizer que *projeto de arquivos* significa

“plano/estudo visando a implementação mais adequada de arquivos de dados”.

Nos sistemas computacionais de interesse em PRA esse estudo estará associado ao gerenciamento e organização de arquivos de dados.

Projeto de Arquivos

O gerenciamento de arquivos a ser implementada depende:

1. do tipo de dispositivo de armazenamento, suas características físicas e operacionais;
2. da necessidades da aplicação que utiliza os dados armazenados;
3. das características funcionais não apenas dos processadores de entrada e saída mas também do próprio SO.

Projeto de Arquivos - Programador de Aplicação

Mais ainda...

Na visão do programador de aplicação o foco desse projeto está na manipulação das estruturas de dados para a atualização, remoção, inserção, ordenação, intercalação, cópia, etc, de arquivos de maneira eficiente...

Um arquivo, nesse caso, é uma entidade lógica “vista” através de estruturas de dados que o acessam através de uma abstração de sua real condição conforme os dispositivos que o armazenam.

Essa abstração é fornecida através do sistema operacional...

Projeto de Arquivos - Programador de Aplicação

Entidades são objetos sobre os quais se armazenam informações. Ex: empregados de uma Cia

Entidades possuem propriedades características chamadas atributos. Ex: nome, endereço, idade, matrícula,...



Funcionário:
Nome
Matrícula
Endereço
Estado civil
...

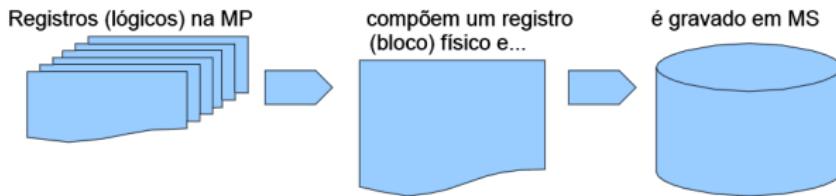
Projeto de Arquivos - Programador de Aplicação

Um registro (ou registro lógico) é um conjunto de pares

$< atributo, valor >$

Ex: Nome: João, idade: 55, matrícula: 1234...

Um conjunto de registros de uma classe de entidades é armazenado na memória principal em arrays/tabelas e em memória secundária em arquivos.



O acesso a determinado registro (subconjunto de registros) é realizado através de chave primária (secundária). Ex: matrícula (idade)

Projeto de Arquivos - Programador de Aplicação

Arquivo coleção de estruturas de dados agregando valores heterogêneos (registros) ou não (caracteres).

A linguagem de programação expressa o nível lógico-conceitual (armazenamento/recuperação) na forma de registros e arquivos.

O sistema operacional mapeia os registros e arquivos em dispositivos reais de armazenamento [Ferraz, 2003].

Projeto de Arquivos - Programador de Sistema

O programador de sistema, por sua vez, lida com aspectos de projeto tais como:

- Sistema de blocagem,
- *caching*,
- diretórios,
- canais de I/O,
- rotinas de baixo nível que permitem acesso a registros independentemente do dispositivo,
- etc...

Projeto de Arquivos - Programador de Sistema

Começando pela visão do programador de sistema:

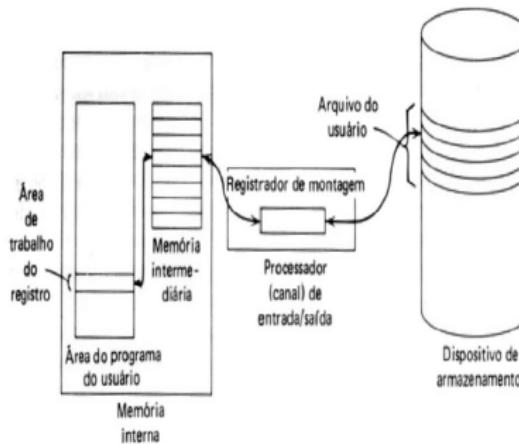


Figura 1.2 Fluxo de dados entre a memória interna e o armazenamento externo.

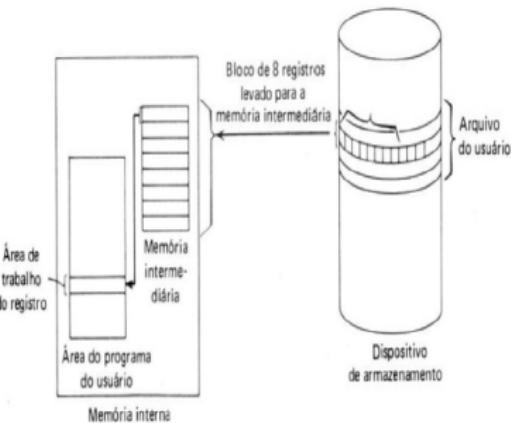


Figura 1.3 Bloco de oito registros lógicos movido para a memória intermediária e depois, um a um, para a área de trabalho do registro.

Figura 1: Comunicação Memória Interna X Memória Externa

Fonte: [Claybrook, 1983]

Projeto de Arquivos - Programador de Sistema

Programadores de sistema devem criar/prover abstrações de dispositivos de armazenamento.

Essas abstrações são manipuladas pelo programador de aplicação.

Através dessas abstrações o programador de aplicação pode se referir a diferentes e complexos dispositivos de armazenamento de forma sintaticamente simples utilizando as linguagens de programação:

```
FILE *fp=NULL;  
...  
fp = fopen(nomeArq,"w+b");  
rewind(fp);  
fread(&aux,sizeof(reg),1,fp);  
...
```

Figura 2: Exemplo em C

Projeto de Arquivos - Programador de Sistema

Os dispositivos de armazenamento secundário mais comuns são:

- Fitas,
- Discos magnéticos,
- Discos ópticos e ultimamente
- os Solid-State Drives.

Independente do dispositivo, é importante analisar:

1. Capacidade de armazenamento;
2. **Portabilidade:** quando se tratar de dispositivos removíveis;
3. Custo relativo;
4. Tamanho de registro (dados contíguos) endereçável;
5. **Método de acesso:** seqüencial, direto (aleatório)
6. Velocidade de transferência da (ou para a) memoria principal
7. Tempo de posicionamento para mover a cabeça r/w até o a trilha contendo o registro alvo
8. **Latência:** tempo de retardo rotacional para discos ou o tempo de partida para a fita alcançar velocidade operacional.

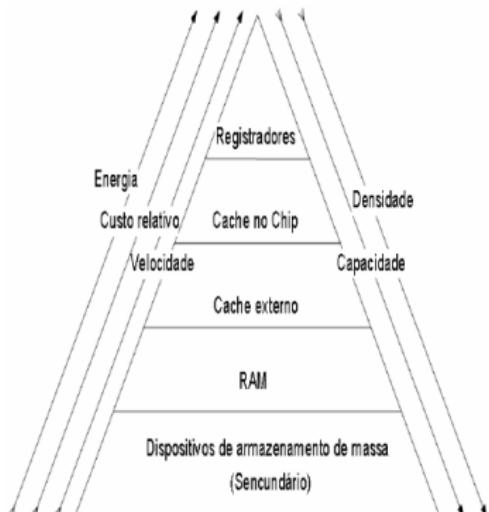
Projeto de Arquivos - Programador de Sistema

Dispositivos de armazenamento secundário são mais baratos, mais lentos e de maior capacidade do que os dispositivos de memória interna (principal):



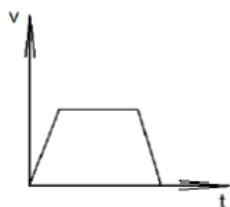
Fonte: [Ferraz, 2003]

Fonte: Desconhecida



Projeto de Arquivos - Programador de Sistema

Fitas em Rolos:



Fitas em Cartuchos:

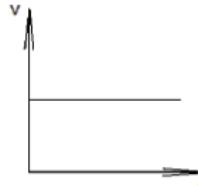


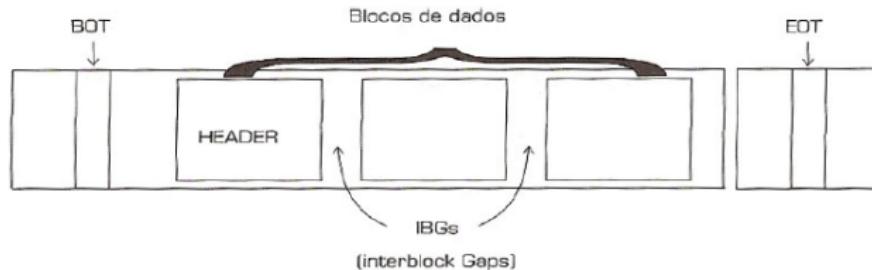
Figura 4: Acionador Streaming

Figura 3: Acionador Start-Stop

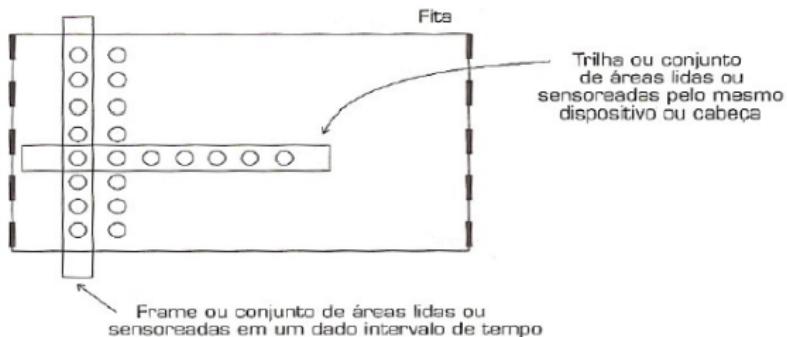
Fontes: [Ferraz, 2003] e http://en.wikipedia.org/wiki/Magnetic_tape_data_storage

Projeto de Arquivos - Programador de Sistema

Fitas em Rolos:



Blocos de dados e intervalos entre blocos.



Projeto de Arquivos - Programador de Sistema

Fitas em Rolos - Cálculo do Comprimento S_T [Claybrook, 1983]:

comprimentoTotal = numeroTotaldeBlocos × comprimentoDeUmBloco →

$$S_T = \frac{N_L}{BF} \left(IRG + \frac{BF \times LRL}{DEN} \right)$$

onde:

S_T : comprimento total da fita para o arquivo (polegadas)

N_L : quantidade de registros lógicos do arquivo

BF : fator de blocagem (quantos registros lógicos cabem em um registro físico/bloco físico)

IRG : separador de registros (polegadas)

LRL : comprimento do registro lógico (caracteres)

DEN : densidade (bits ou caracteres por polegada)

Projeto de Arquivos - Programador de Sistema

Fitas em Rolos - Exemplo de Cálculo do Comprimento S_T :

Dados:

N_L : 10.000 registros lógicos

BF : 100 (registros por bloco)

IRG : 0.75"

LRL : 160 caracteres

DEN : 800 cpi

$$S_T = \frac{10000}{100} \left(0.75 + \frac{100 \times 160}{800} \right) = 2075 \text{ polegadas} \cong 52.7 \text{ metros}$$

Projeto de Arquivos - Programador de Sistema

Fitas em Rolos - Cálculo do Tempo para ler o arquivo T_T [Claybrook, 1983]:

$$\text{tempo Total} = \text{numTotaldeBlocos} \times (\text{tempoParaUltrapassarIRG} + \text{tempoParaLerUmBloco}) \rightarrow$$

$$T_T = \frac{N_L}{BF} \left(T_A + \frac{BF \times LRL}{SPD \times DEN} \right)$$

onde:

T_T : tempo total de leitura para o arquivo

SPD : velocidade de leitura da fita (em polegadas/segundo)

$SPD \times DEN$: velocidade de transferência

T_A : tempo de partida/parada ou tempo para ultrapassar IRG (em milissegundos)

Projeto de Arquivos - Programador de Sistema

Fitas em Rolos - Exemplo de Cálculo do Tempo de Leitura T_T :

Dados:

N_L : 10.000 registros lógicos

BF : 100 (registros por bloco)

T_A : 12,6 ms

RL : 160 caracteres

DEN : 800 cpi

SPD : 75 pol/seg

$$T_T = \frac{10000}{100} \left(0.75 + \frac{100 \times 160}{75 \times 800} \right) \cong 27,9 \text{ segundos}$$

Projeto de Arquivos - Programador de Sistema

Discos Magnéticos (HD)

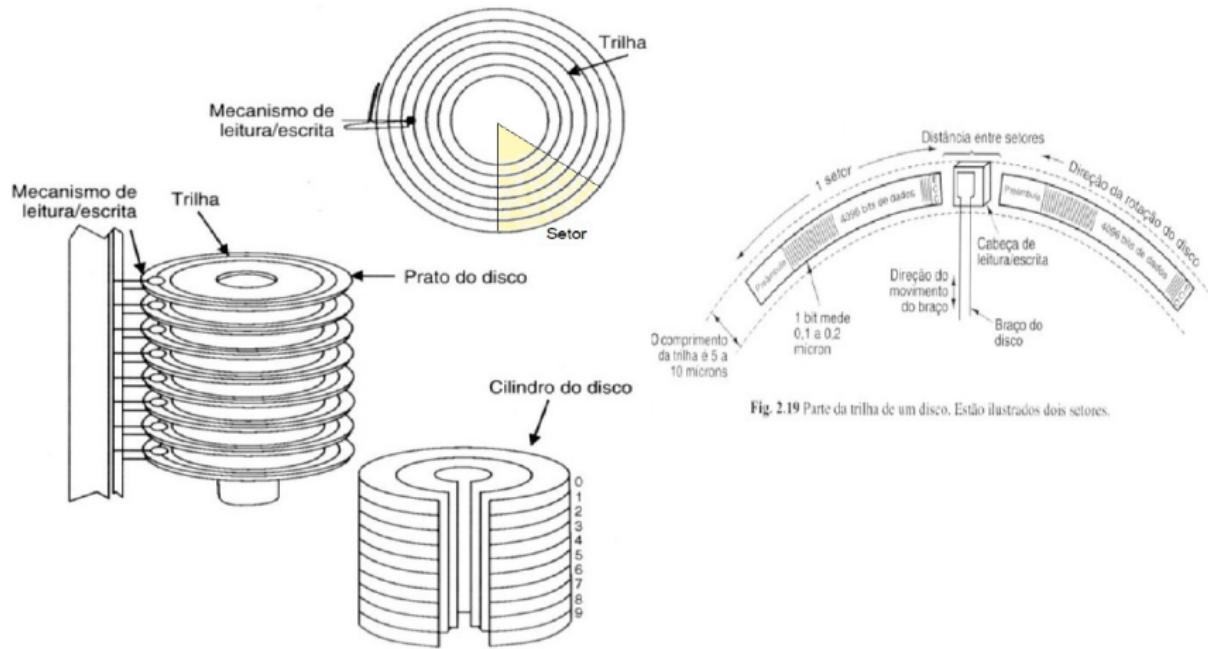


Fig. 2.19 Parte da trilha de um disco. Estão ilustrados dois setores.

Projeto de Arquivos - Programador de Sistema

Discos Magnéticos (HD)

O ideal é que o arquivo ocupe setores/trilhas e cilindros contíguos, caso contrário o disco estará fragmentado e exigirá um tempo maior para posicionamento do cabeçote r/w nas diversas regiões onde se fragmentou o arquivo.

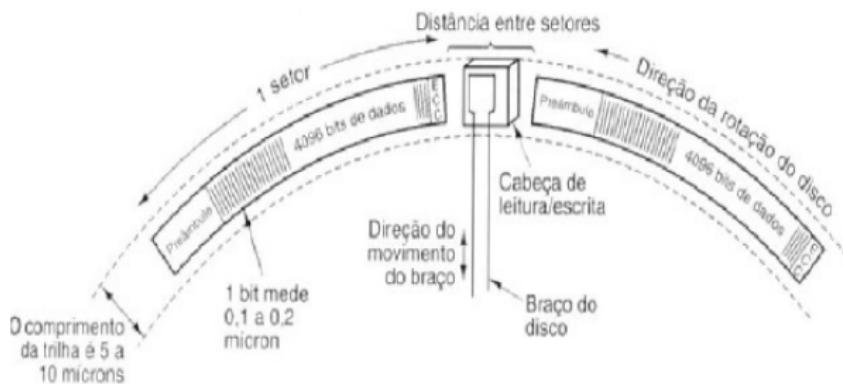


Figura 5: Parte da trilha de um disco. Estão ilustrados dois setores.

Projeto de Arquivos - Programador de Sistema

Discos Magnéticos - Cálculo do Número de Trilhas para armazenar um arquivo N_T [Claybrook, 1983]:

$$N_T = \frac{N_B}{N_{B/T}} \quad N_B = \frac{N_L}{BF} \quad N_{B/T} = \left\lfloor \frac{T_{CAP}}{B_S} \right\rfloor \quad B_S = G_S + NDI + BF \times LRL$$

onde:

N_B : número de blocos por arquivo

N_L : Número de registros lógicos de um arquivo

BF : Fator de bloco (blocagem)

$N_{B/T}$: Número de blocos por trilha

T_{CAP} : Capacidade de trilha

B_S : Tamanho do bloco em caracteres

G_S : Espaço total do separador de bloco

NDI : Dados do sistema por bloco (caracteres)

LRL : Comprimento do Registro Lógico (caracteres)

Projeto de Arquivos - Programador de Sistema

Discos Magnéticos - Capacidade de Armazenamento:

$$T_{CAP} = N_{S/T} \times N_{B/S}$$

$$T_C = N_{T/C} \times T_{CAP}$$

$$T_{HD} = N_C \times T_C$$

onde:

T_{HD} : capacidade do HD

T_{CAP} : capacidade da trilha

$N_{S/T}$: número de setores por trilha

$N_{B/S}$: número bytes por setor

T_C : Capacidade do cilindro

$N_{T/C}$: número de trilhas por cilindro

N_C : número de cilindros

Projeto de Arquivos - Programador de Sistema

$$T_{ACCESS} = T_{SEEK} + T_{LAT} + T_{DT}$$

- Tempo de Acesso

T_{SEEK} : tempo de posicionamento

T_{LAT} : (tempo de latência), tempo para que o bloco (alvo) passe sob a cabeça R/W e possa ser processada a transferência de dados

T_{DT} : tempo de transferência

O tempo para mudar entre cabeças de R/W é desprezível e não entra na conta.

- Em um disco não fragmentado, ler um arquivo seqüencialmente é mais rápido, pois o acesso seqüencial minimiza o tempo de busca e de latência.
- Para uma leitura aleatória (fragmentada), quase todos os acessos terão tempo de busca, tempo de latência e tempo de transferência.

Projeto de Arquivos - Programador de Sistema

Considere uma unidade de disco magnético com as seguintes características :

- Setor : 512 bytes
- Taxa de transferência : 20 Mbytes/seg
- Tempo de seek médio : 8 ms
- Tempo latência médio : 4 ms
- Capacidade por trilha : 85 Kbytes
- Capacidade total : 9100 Mbytes
- N° de pratos/superfícies : 10/20
- Responda:
 1. Qual é o número de cilindros do disco ?
 2. Quantos registros de 128 bytes podem ser armazenados em 1 cilindro do disco ?
 3. Calcule o tempo gasto para a leitura seqüencial de 50.000 setores.
 4. Calcule o tempo gasto para a leitura aleatória de 50.000 setores.

Projeto de Arquivos - Programador de Sistema

- Qual é o número de cilindros do disco ?

$$\frac{9100 \text{ MB}}{20 \text{ superfícies}} = 455 \text{ MB/superfície}$$

$$455 \times 1024 \times \frac{1024 \text{ bytes}}{85 \times 1024 \text{ bytes/trilha}} = 5.481 \text{ trilhas/superfície}$$

O número de cilindros no disco é igual ao número de trilhas por superfície. Portanto, há 5.481 cilindros no disco.

Projeto de Arquivos - Programador de Sistema

- Quantos registros de 128 bytes podem ser armazenados em 1 cilindro do disco ?

$$(85 \times 1024 \text{ bytes/trilha}) \times 20 \text{ trilhas/cilindro} = 1.740.800 \text{ bytes}$$

Portanto cabem $\frac{1.740.800 \text{ bytes}}{128 \text{ bytes/registro}} = 13.600$ registros em um cilindro.

Projeto de Arquivos - Programador de Sistema

■ Calcule o tempo gasto para a leitura seqüencial de 50.000 setores.

- A leitura seqüencial envolve um tempo de seek, um tempo de latência e o tempo gasto com a transferência. Tempo de seek: 8 ms. Tempo de latência: 4ms

- Tempo de transferência:
$$50.000 \text{ setores} \times 512 \text{ bytes/setor} = \frac{25.600.000 \text{ bytes}}{20 \times 1024 \times 1024 \text{ bytes/seg}} = 1,22 \text{ seg}$$

- Tempo total: $0,008 + 0,004 + 1,22 \cong 1,23 \text{ seg}$

Projeto de Arquivos - Programador de Sistema

- **Calcule o tempo gasto para a leitura aleatória de 50.000 setores.**
 - A leitura de cada setor requer um tempo de seek, um tempo de latência e o tempo de transferência. Dessa maneira, serão necessários 50.000 seeks e 50.000 latências no total, além do tempo de transferência.
 - Tempo total: $50.000 \times (0,008 + 0,004 \text{ seg}) + 1,22 \text{ seg} = 601,22\text{seg}$

Revisão sobre Arquivos em Linguagem C

Arquivos em C

Arquivos Texto: os dados são armazenados sem uma representação fixa, ou seja, meramente como uma sequência de caracteres.

Isto permite que programas editores de texto (Notepad, Word, etc.) possam abri-los e editá-los como qualquer outro arquivo, sem a necessidade de se utilizar o programa de onde o arquivo foi originalmente produzido.

Este tipo de arquivo requer que o programa que o esteja manipulando seja responsável por “dar sentido” aos caracteres lidos/gravados (*Parsing*). Por exemplo: os primeiros 10 caracteres de cada linha representam um nome, os próximos 3 um código numérico e assim por diante.

Arquivos Binários: os dados são armazenados como sequências de bytes formados (variáveis, estruturas, etc.).

Dado que há um formato específico para o armazenamento da informação, apenas o programa de origem é capaz de ler os dados armazenados no arquivo. Outra vantagem esta no fato de que dados complexos podem ser lidos com apenas um acesso à disco (p.ex. ler um estrutura composta por vários campos simultaneamente).

Arquivos em C

Abrir um arquivo em C significa criar uma *stream* e conectá-la ao arquivo em disco. Em C, **fopen()** é a função que abre (ou cria) arquivos, seu protótipo encontra-se abaixo e seu uso necessita da inclusão de **stdio.h**.

FILE fopen(const char* filename, const char* mode);

filename indica o nome do arquivo a ser aberto/criado

mode mode associado à *stream*

Se a operação de abertura transcorre sem problemas **fopen()** devolve um ponteiro de referência que será usado para manipular o arquivo, caso haja algum erro um **NULL** será devolvido.

Arquivos em C

Os modos de abertura do arquivo **TEXTO** podem ser:

| Valor | Descrição |
|---------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| "r" ou "rt" | Abre arquivo texto apenas para leitura e posiciona o ponteiro no início do arquivo. Se o arquivo não existe (previamente), fopen devolverá NULL. |
| "w" ou "wt" | Abre arquivo texto apenas para escrita. Se o arquivo não existe ele será criado. Se o arquivo já existe ele será aberto para escrita, porém seu conteúdo será apagado. Posiciona o ponteiro no início do arquivo. |
| "a" ou "at" | Abre arquivo texto para anexação, nesse caso só será possível acrescentar dados a partir do ponto de abertura que sempre ocorrerá no final do arquivo, portanto não há possibilidade de perda dos que já existam no arquivo. Se o arquivo já existe ele será aberto para anexação e o ponteiro de arquivo será posicionado no final deste (EOF – End Of File). Se não existe, ele será criado e o ponteiro posicionado no início deste. |
| "r+" ou "r+t" | Abre arquivo texto para leitura e escrita e posiciona o ponteiro no início do arquivo. Se o arquivo não existe (previamente), fopen devolverá NULL. |
| "w+" ou "w+t" | Abre arquivo texto para escrita e leitura. Se o arquivo não existe ele será criado. Se já existe ele será aberto, porém seu conteúdo será apagado. Posiciona o ponteiro no início do arquivo. |
| "a+" ou "a+t" | Abre arquivo texto para anexação (será possível acrescentar dados , ou seja, escrever dados sem perda dos que já existam no arquivo) e para leitura. Será possível retornar o ponteiro inclusive para posições anteriores à da abertura, porém para estas posições só será possível e efetuar operações de leitura. Se o arquivo já existe ele será aberto para anexação e o ponteiro de arquivo será posicionado no final deste (EOF). Se não existe, ele será criado e o ponteiro posicionado no inicio deste. |

OBS.: Para abrir arquivos **binários** basta substituir o símbolo “t” nos casos acima por “b”.

Arquivos em C - Exemplos

```
1 #include "stdio.h"
2 ....
3 main(void) {
4     FILE * fp;
5     char filename [] = "arq.txt";
6
7     if ((fp = fopen(filename, "w+")) == NULL) {
8         printf ("Erro na abertura do arquivo ");
9         exit (0);
10
11     ....
12 }
```

Arquivos em C - Exemplos

```
1 #include "stdio.h"
2 ....
3 main(void) {
4     FILE * fp;
5     char filename [] = "c:\\ teste \\arq.txt",
6         modo[] = "w+";
7
8     if ((fp = fopen(filename, modo)) == NULL) {
9         printf ("Erro na abertura do arquivo ");
10        exit (0); }
11
12     ....
13 }
```

Arquivos em C - Exemplos

```
1 #include "stdio.h"
2 #define MSG_ERR "Qtde Invalida de Parametros"
3 ....
4 main(int argc, char * argv []) {
5     FILE * fp;
6
7     if (argc < 2) {
8         printf (MSG_ERR);
9         exit (0); }
10
11    if ((fp = fopen(argv[1], "w+")) == NULL) {
12        printf ("Erro na abertura do arquivo ");
13        exit (0); }
14
15    ....
16 }
```

Arquivos em C

Stream: fluxo de movimentação de dados entre a memória principal e a secundária.

Buffer: memória lógica auxiliar associada a um *stream*.

- Para esvaziar o *buffer* corrente usa-se o comando:

int fflush(FILE * fp)

- Para fechar o *stream* corrente usa-se o comando:

int fclose(FILE * fp)

Arquivos em C

Acesso Sequencial vs Aleatório

1. Arquivos em C funciona como uma fita lógica;
2. Eles são manipulados através de um *apontador* que indica a posição (em bytes) do ponto exato sendo acessado na ‘fita’;
3. Início do arquivo é indicado pela posição zero;
4. Arquivos recém-criados tem comprimento zero e o ponteiro sempre indicará a posição zero;
5. Para arquivos pré-existentes recém abertos, o ponteiro indicará a posição definida pelo modo de abertura do mesmo (a, r, w, ...);
6. Tanto a leitura quanto a escrita em arquivo, levam em consideração a posição do apontador;
7. A posição do apontador é atualizada após cada operação em função da quantidade de bytes lidos/escritos;

Arquivos em C

Exemplo:

- Ao abrir um arquivo texto para leitura, solicitou-se a leitura de 10 caracteres;
- Cada caracter tem tamanho 1 byte;

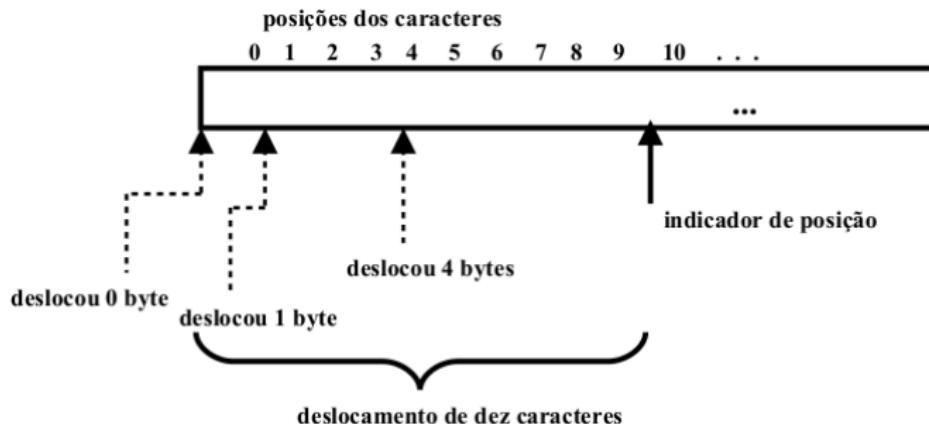


Figura 6: Situação de um arquivo após operação de leitura de 10 caracteres

Arquivos em C - Acesso Aleatório

- Para reposicionar o apontador de leitura novamente no início do arquivo, usa-se:

void rewind(FILE * fp)

- Para reposicionar o apontador de leitura para uma posição específica, usa-se:

int fseek(FILE * fp, long numbytes, int origin)

onde:

1. ponteiro para o arquivo a ser manipulado
2. quantidade de bytes a serem deslocados
3. ponto de referência, em relação ao qual, o deslocamento é realizado.

Possíveis valores:

| Macro | Valor | Descrição |
|----------|-------|-------------------------------------|
| SEEK_SET | 0 | Mover à partir do início do arquivo |
| SEEK_CUR | 1 | Mover à partir da posição atual |
| SEEK_END | 2 | Mover à partir do final do arquivo |

Arquivos em C - Acesso Aleatório

- Para determinar a posição atual (em bytes) do apontador de leitura em relação ao início do arquivo. Preferencialmente utilizado para arquivos binários. Uso:

```
int ftell( FILE * fp )
```

Observação: Se um arquivo for aberto em modo *append* ("a"), o comando *ftell* retorna zero se, antes disso, não for executado um

```
fseek( fp, 0L, SEEK_END )
```

Arquivos em C - Acesso Aleatório

Exemplos de uso:

- Para posicionar o apontador no nono registro do arquivo:

fseek(fp, 9 * sizeof(struct list_type), SEEK_SET);

- Para posicionar o apontador no início do arquivo:

fseek(fp, 0L, SEEK_SET);

- Para re-posicionar o apontador no início do arquivo:

rewind(fp);

ou

fseek(fp, - ftell(fp), 1);

- Para posicionar o apontador no final do arquivo:

fseek(fp, 0L, SEEK_END);

Arquivos em C - I/O (modo texto)

Comandos de leitura e/ou gravação em arquivos texto:

- **int putc(int ch, FILE * fp)** escreve um caracter *ch* no arquivo *fp*. Se bem sucedido, a função retorna o próprio caracter, caso contrário retorna EOF (End-Of-File).
- **int getc(FILE * fp)** lê o caracter posicionado sob o apontador de leitura. Retorna EOF se o fim de arquivo for alcançado.
- **int fputs(const char * str, FILE * fp)** grava uma cadeia de caracteres *str* em *fp*. Retorna EOF caso um erro de gravação ocorra. A função também traduz o caracter '\n' para os símbolos CRLF = carriage return / line feed.
- **int fgets(char * str, int tam, FILE * fp)** lê uma cadeia de caracteres em *str* (terminada em '\0'), de um comprimento máximo *tam* (será menor que *tam* se um fim de linha for lido antes). A função retorna NULL caso ocorra um erro de leitura ou EOF seja encontrado.

Arquivos em C - I/O (modo texto)

Comandos de leitura e/ou gravação em arquivos texto:

- Para uso de formato específico para o processo de leitura/escrita utilizam-se os comandos:

```
int fprintf( FILE * fp, const char * string_formato, <lista de
            variáveis...> )
```

```
int fscanf( FILE * fp, const char * string_formato, <lista de
            variáveis...> )
```

Arquivos em C - I/O (modo texto)

Para se verificar se o fim de arquivo foi alcançado:

```
while( ( c = fgetc( fp ) ) != EOF )
```

...

Arquivos em C - I/O (modo binário)

Comandos de leitura e/ou gravação em arquivos binários:

- **int fread(void * buf, int size, int count, FILE * fp)** efetua a leitura para um *buffer* de dados (*buf*), de uma certa quantidade de dados (*count*) de um certo tipo de dados de tamanho *size* (em bytes) a partir de um arquivo *fp*.
- **int fwrite(void * buf, int size, int count, FILE * fp)** efetua a gravação de uma certa quantidade (*count*) de um *buffer* de dados (*buf*), de um certo tipo de dados de tamanho *size* (em bytes) para um arquivo *fp*.

Arquivos em C - I/O (modo binário)

Exemplo:

```
1  typedef struct {
2      int i;
3      char str [30];
4  }reg;
5
6 main(void)
7 {
8     FILE *fp;
9     int count, size;
10    reg registro , aux;
11    char nome_arq[ ] = "teste . bin ";
12    registro . i = 100;
13    strcpy ( registro . str , "valor ");
14
15    if ( ( fp = fopen( nome_arq,"w+b" ) ) == NULL )
16    { printf ("Erro na abertura do arquivo "); exit (0); }
17
18    size = sizeof(reg);
19    count = 1;
20    if ( ( fwrite( &registro , size , count, fp ) ) < count)
21    { puts("Erro na operacao de escrita "); exit (0); }
22
23    rewind(fp );
24    fread(&aux,sizeof(reg).1,fp );
25    printf (" valores lidos : %i, %s ", aux.i ,aux.str );
26    fclose (fp );
27 }
```

Arquivos em C - I/O (modo binário)

Mais duas funções úteis

- **int rename(const char * oldname, const char * newname)**
renomeia o arquivo chamado *oldname* para *newname*. Atenção!: o arquivo *oldname* não pode estar em uso.
A função devolve zero caso a operação resulte em sucesso ou diferente de zero caso ocorra erro.
- **int remove(const char * filename)** apaga o arquivo definido por *filename*. Atenção!: o arquivo *filename* não pode estar em uso.
A função devolve zero caso a operação resulte em sucesso ou diferente de zero caso ocorra erro.

Organização de Arquivos

- Um banco de dados é armazenado em uma coleção de arquivos. Cada arquivo é uma sequência de registros. Um registro é uma sequência de campos;
- Uma chave é uma sequência de um ou mais campos;
- O valor de uma chave-primária identifica um único registro. O valor de uma chave-secundária identifica um conjunto de registros;
- Os registros podem ser de tamanho fixo ou variável;

Org. de Arquivos - Registros de Tamanho Fixo

- Mais fáceis de implementar
- Cada arquivo possui registros de um tipo particular
- Arquivos diferentes são usados por relações (conceito de banco de dados) diferentes
- Abordagem simples: “concepção vetorial”
 - Armazene o registro i começando no byte $n * (i^1)$, onde n é corresponde ao tamanho de cada registro.
 - O acesso é simples porém, a não ser que o buffer/bloco seja de tamanho múltiplo de um registro, pode ocorrer o cruzamento da fronteira do bloco
- Alternativas para remoção do registro i :
 - move registros $i + 1, \dots, n$ para i, \dots, n^1
 - move apenas o registro n para i
 - Não move nada. Utilize uma lista encadeada de “registros livres”: *Free List*

Org. de Arquivos - Registros de Tamanho Fixo

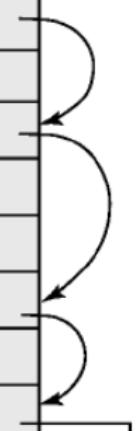
| | | | |
|----------|-------|------------|-----|
| record 0 | A-102 | Perryridge | 400 |
| record 1 | A-305 | Round Hill | 350 |
| record 2 | A-215 | Mianus | 700 |
| record 3 | A-101 | Downtown | 500 |
| record 4 | A-222 | Redwood | 700 |
| record 5 | A-201 | Perryridge | 900 |
| record 6 | A-217 | Brighton | 750 |
| record 7 | A-110 | Downtown | 600 |
| record 8 | A-218 | Perryridge | 700 |

Org. de Arquivos - Reg. de Tamanho Fixo - *Free list*

- O endereço do primeiro registro deletado é armazenado no header.
- Use o primeiro registro para armazenar o endereço do segundo registro deletado e assim por diante.
- Imagine que os endereços armazenados são como ponteiros porém apontando para uma localização no arquivo

Org. de Arquivos - Reg. de Tamanho Fixo - *Free list*

| | | | | |
|----------|-------|------------|-----|--|
| header | | | | |
| record 0 | A-102 | Perryridge | 400 | |
| record 1 | | | | |
| record 2 | A-215 | Mianus | 700 | |
| record 3 | A-101 | Downtown | 500 | |
| record 4 | | | | |
| record 5 | A-201 | Perryridge | 900 | |
| record 6 | | | | |
| record 7 | A-110 | Downtown | 600 | |
| record 8 | A-218 | Perryridge | 700 | |



The diagram illustrates a linked list of fixed-size records. Each record is represented as a row in a table. A pointer (represented by a curved arrow) is placed at the end of each row, pointing to the start of the next row. This continues until the last record, which has its pointer pointing to a small square containing a minus sign (-), indicating it is a free list node.

Org. de Arquivos - Registros de Tamanho Variável

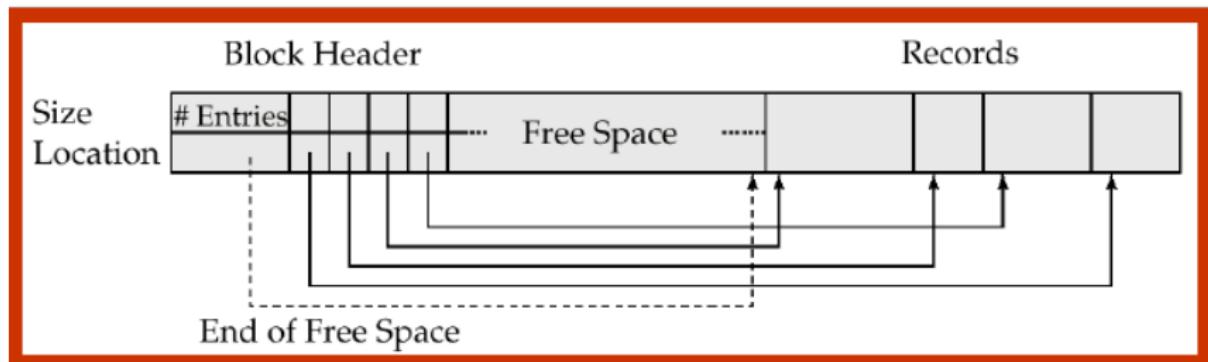
- Esse tipo de registro surge em decorrência de vários fatores:
 - Armazenamento de tipos variados de registros em um arquivo.
 - O tipo de registro permite comprimento variável de um ou mais campos.
 - O registro permite campos repetidos.
- Representação na forma de cadeia de bytes (**Byte string**)
 - Utilize um caracter de controle *end-of-record* (\perp) para marcar o fim de cada registro
 - Problemas com a remoção do registro: não é eficiente a reutilização do espaço o que provoca fragmentação no disco
 - Problemas se há crescimento (maior) do registro: preciso realocar o registro em operação delicada.

Org. de Arquivos - Reg. de Tamanho Variável - *Byte string*

| | | | | | | | | |
|---|------------|-------|-----|-------|-----|-------|-----|---|
| 0 | Perryridge | A-102 | 400 | A-201 | 900 | A-218 | 700 | ⊥ |
| 1 | Round Hill | A-305 | 350 | ⊥ | | | | |
| 2 | Mianus | A-215 | 700 | ⊥ | | | | |
| 3 | Downtown | A-101 | 500 | A-110 | 600 | ⊥ | | |
| 4 | Redwood | A-222 | 700 | ⊥ | | | | |
| 5 | Brighton | A-217 | 750 | ⊥ | | | | |

Org. de Arquivos - Reg. de Tamanho Variável - *Slotted Page Structure*

- O descritor da **Slotted page** contém:
 - Número de entradas de registro
 - Fim do espaço livre no bloco
 - Um vetor contendo a localização e tamanho de cada registro



Org. de Arquivos - Registros de Tamanho Variável

- Representação por meio de vários registros de tamanho fixo:
 - Espaço reservado
 - Ponteiros
 - **Espaço reservado** – utiliza o registro de máximo tamanho como um tamanho fixo; registros menores que este recebem marcas *end-of-record* no espaço não utilizado.

| | | | | | | | |
|---|------------|-------|-----|-------|-----|-------|-----|
| 0 | Perryridge | A-102 | 400 | A-201 | 900 | A-218 | 700 |
| 1 | Round Hill | A-305 | 350 | ⊥ | ⊥ | ⊥ | ⊥ |
| 2 | Mianus | A-215 | 700 | ⊥ | ⊥ | ⊥ | ⊥ |
| 3 | Downtown | A-101 | 500 | A-110 | 600 | ⊥ | ⊥ |
| 4 | Redwood | A-222 | 700 | ⊥ | ⊥ | ⊥ | ⊥ |
| 5 | Brighton | A-217 | 750 | ⊥ | ⊥ | ⊥ | ⊥ |

Org. de Arquivos - Reg. de Tamanho Variável - Ponteiro

■ Método do Ponteiro

- Um registro de tamanho variável é representado por uma lista encadeada de registros de tamanho fixo.
- É interessante quando é desconhecido o registro de tamanho máximo

| | | | | |
|---|------------|-------|-----|--|
| 0 | Perryridge | A-102 | 400 | |
| 1 | Round Hill | A-305 | 350 | |
| 2 | Mianus | A-215 | 700 | |
| 3 | Downtown | A-101 | 500 | |
| 4 | Redwood | A-222 | 700 | |
| 5 | | A-201 | 900 | |
| 6 | Brighton | A-217 | 750 | |
| 7 | | A-110 | 600 | |
| 8 | | A-218 | 700 | |



Org. de Arquivos - Reg. de Tamanho Variável - *Ponteiro*

- Desvantagem: desperdiça espaço em todos os registros encadeados, exceto o primeiro.
- A solução é permitir dois tipos de blocos:
 - Bloco âncora contém o primeiro registro da cadeia
 - Bloco de Overflow contém o demais registros que não aqueles que são os primeiros de uma cadeia



Organização de Registros no Arquivo

- **Heap** – O registro pode ser colocado em qualquer local livre no arquivo
- **Sequencial** – Armazena registros em ordem sequencial baseado no valor de uma chave de pesquisa
- **Hashing** – Uma *hash-function* é computada sobre algum atributo de registro; o resultado especifica em qual bloco do arquivo o registro será colocado
- Registros de cada relação podem ser armazenados em arquivos separados. Porém, em uma organização **clustering** os registros de relações diferentes podem ser armazenadas no mesmo arquivo
 - Motivação: armazenar registros em um mesmo bloco minimiza I/Os

Avaliação #1 - Arquivos

Título: Trabalho Prático #1 - Construção de uma Base de Dados

Objetivo: Construir uma base de dados inicial que servirá como argumento de entrada para todos os demais trabalhos de indexação de arquivos a serem vistos na disciplina.

Forma de Entrega: Deve ser entregue todos os códigos-fonte (devidamente documentados/comentados), bem como deve ser preparada uma apresentação que inclui uma demonstração de funcionamento do sistema.

Deve ainda ser entregue uma cópia do arquivo construído previamente pelo sistema, que contenha (pelo menos) 1 GB de tamanho em disco.

Prazo de Entrega: 2 semanas

Avaliação #1 - Arquivos

- Critérios:
1. Equipes de 3 alunos
 2. Cada equipe recebe um tema diferente para a construção dos registros da base de dados
 3. O programa a ser desenvolvido deve permitir ao usuário:
 - 3.1 Escolher a quantidade de registros a ser gerada (pelo processo estocástico) ou o tamanho físico do arquivo a ser gerado;
 - 3.2 Permitir a gravação de registros através de paginação (tamanho definido pelo usuário);
 - 3.3 Permitir a recuperação de registros através de paginação (tamanho definido pelo usuário) e de leitura sequencial;
 - 3.4 Exibir ao final do processo o tempo gasto para o processamento.

Avaliação #1 - Arquivos

Temas:

- **Tema 1 - Escola:** cadastro de alunos contendo: nome do aluno, data de nascimento (dd/mm/aaaa), nº de matrícula, lista de disciplinas matriculadas (quantidade indeterminada);
- **Tema 2 - Diário de Classe:** relatório acadêmico contendo: nome do aluno, nº de matrícula, notas das provas realizadas (quantidade indeterminada), nº de faltas, média aritmética e situação acadêmica (aprovado, exame ou reprovado);
- **Tema 3 - Loja:** cadastro de clientes contendo: nome do cliente, endereço, telefone, data de nascimento (dd/mm/aaaa), código identificador, lista de itens (códigos dos produtos) comprados (qtde. indeterminada);

Avaliação #1 - Arquivos

Temas:

- **Tema 4 - Vendas:** listagem de vendas de uma loja: código do cliente, código do vendedor, data da venda (dd/mm/aaaa), valor da venda (R\$), listagem dos itens comprados (qtde. indeterminada);
- **Tema 5 - Fluxo de Caixa:** relatório de operações realizadas: código sequencial da operação, indicador de compra/venda (C ou V), valor da operação, data da operação (dd/mm/aaaa);
- **Tema 6 - Resultados do futebol:** relatório com os resultados do campeonato: nome do time A, nome do time B, placar do jogo, data de realização da partida (dd/mm/aaaa), público pagante, local do jogo;

Avaliação #1 - Arquivos

Temas:

- **Tema 7 - Maratona:** relatório de chegada dos corredores: código do corredor, nome do corredor, tempo de duração da corrida (hh:mm:ss:ms), data da corrida (dd/mm/aaaa);
- **Tema 8 - Previsão do Tempo:** dados da previsão do tempo: nome do local, data da coleta dos dados (dd/mm/aaaa), hora da coleta dos dados (hh:mm:ss:ms), temperatura medida ($^{\circ}$ C);
- **Tema 9 - Imobiliária:** listagem de imóveis para locação: tipo do imóvel (casa ou apto), endereço, n° de quartos, preço do aluguel (R\$), data em que o imóvel ficou disponível (dd/mm/aaaa);

Avaliação #1 - Arquivos

Temas:

- **Tema 10 - Hospital:** relatório médico dos pacientes: código sequencial da operação, nome do médico, nome do paciente, data da internação (dd/mm/aaaa), código do motivo da internação, lista de sintomas;
- **Tema 11 - ACM ICPC:** equipes da maratona de programação: nome do componente #1, nome do componente #2, nome do componente #3, nome da equipe, n° de balões obtidos, n° de tentativas falsas;

Revisão sobre Estruturas de Dados

Estruturas de Dados

Lineares definição, tipos:

Listas definição, encadeadas vs. duplamente encadeadas, métodos de inserção, busca e remoção;

Pilhas definição, métodos de inserção e remoção;

Filas definição, métodos de inserção e remoção;

Hierárquicas definição, tipos: árvores (binária, AVL, ...), grafos, ...

Revisão sobre Estruturas de Dados

Listas

Listas

- Uma **lista encadeada** é uma estrutura de dados que armazena uma sequência linear de elementos de forma dinâmica através de uma cadeia de *átomos* ou *nós*.
- Cada nó contém o objeto a ser armazenado (uma informação) e uma referência ao próximo nó na lista.
- O último nó referencia o vazio (NULL) para indicar fim de lista.

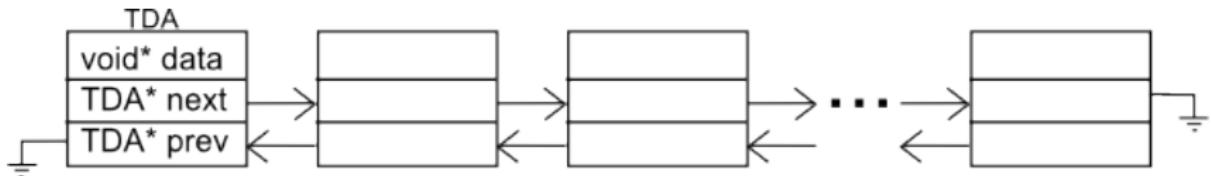


Listas

```
1 struct _TDA_{
2     void * data;
3     struct _TDA_ * next;
4 };
5
6 typedef struct _TDA_ TDA;
7 ...
8
9 TDA * ptrHeader = malloc( sizeof( TDA ) );
10 ptrHead->data = info;
11 ptrHead->prox = NULL;
```

Listas

- Uma lista **duplamente encadeada** referencia (para cada nó) tanto o próximo nó na sequência, quanto o nó anterior.
- Analogamente, o primeiro nó referencia vazio para indicar seu anterior.



Listas

```
1 struct _TDA_{
2     void * data;
3     struct _TDA_ * next;
4     struct _TDA_ * prev;
5 };
6
7 typedef struct _TDA_ TDA;
8 ...
9
10 TDA * ptrHeader = malloc( sizeof( TDA ) );
11 ptrHead->data = info;
12 ptrHead->next = ptrHead->prev = NULL;
```

Operações com Listas

Inserção: para se acrescentar um elemento em uma lista, há três possíveis casos a considerar:

- Inserção no início da lista
 1. Apontar a referência `next` do novo nó sendo inserido para o atual primeiro nó da lista
 2. Apontar a referência `prev` do atual primeiro nó da lista para o novo nó sendo inserido
- Inserção no final da lista
 1. Apontar a referência `next` do último nó na lista para o novo nó sendo inserido
 2. Apontar a referência `prev` do novo nó para o atual último nó da lista
- Inserção em uma posição intermediária na lista. Necessita o uso de um ponteiro auxiliar `curr` que referencia o nó corrente (em relação ao qual a inserção será realizada):
 1. Para o nó anterior a `curr`: aponta-se a referência `next` para o novo nó sendo inserido
 2. Para o próprio nó `curr` aponta-se a referência `prev` para o novo nó sendo inserido
 3. Para o novo nó: aponta-se a referência `prev` para o nó anterior a `curr` e a referência `next` para o próprio nó `curr`

Operações com Listas

Inserção no início da lista:

```
1     ptrNewNode->next = ptrHead;
2     ptrHead->prev = ptrNewNode;
3     ptrHead = ptrNewNode;
4
```

Operações com Listas

Inserção no final da lista:

```
1     ptrTail->next = ptrNewNode;
2     ptrNewNode->prev = ptrTail;
3     ptrTail = ptrNewNode;
4
```

Operações com Listas

Inserção em um ponto intermediário da lista:

```
1     ptrNewNode->prev = ptrCurr->prev;
2     ptrNewNode->next = ptrCurr;
3     ptrCurr->prev->next = ptrNewNode;
4     ptrCurr->prev = ptrNewNode;
5
```

Operações com Listas

Remoção: para se remover um elemento em uma lista, também há três possíveis casos a considerar:

- Remoção no início da lista
 1. Apontar a referência `prev` do próximo nó da lista para `NULL`
- Remoção no final da lista
 1. Apontar a referência `next` do penúltimo nó da lista para `NULL` (o último nó aponta para este nó por seu ponteiro `prev`)
- Remoção em uma posição intermediária da lista (em relação a `curr`):
 1. Apontar a referência `next` do nó anterior a `curr` para `curr->next`
 2. Apontar a referência `prev` do próximo nó a `curr` para `curr->prev`

Operações com Listas

Remoção no início da lista:

```
1     ptrCurr = ptrHead;
2     ptrHead = ptrHead->next;
3     ptrHead->prev = NULL;
4     free( ptrCurr );
5
```

Operações com Listas

Remoção no final da lista:

```
1     ptrCurr = ptrTail ;
2     ptrTail = ptrTail->prev;
3     ptrTail->next = NULL;
4     free( ptrCurr );
5
```

Operações com Listas

Remoção em um ponto intermediário da lista:

```
1     ptrCurr->prev->next = ptrCurr->next;  
2     ptrCurr->next->prev = ptrCurr->prev;  
3     free( ptrCurr );  
4
```

Operações com Listas

Busca: para se recuperar um elemento em uma lista encadeada, a busca sequencial é a única alternativa possível:

```
1     while( ptrCurr ) {  
2         if( ptrCurr->data == info )  
3             break;  
4         ptrCurr = ptrCurr->next;  
5     }  
6     return ptrCurr;  
7
```

Operações com Listas - Exercícios

1. Criar uma lista duplamente encadeada de N números inteiros (gerados aleatoriamente)
2. Percorrer a lista exibindo sequencialmente o conteúdo de cada nó
3. Percorrer a lista exibindo sequencialmente o conteúdo de cada nó na ordem inversa
4. Remover da lista todos os números pares (exiba novamente a lista resultante)

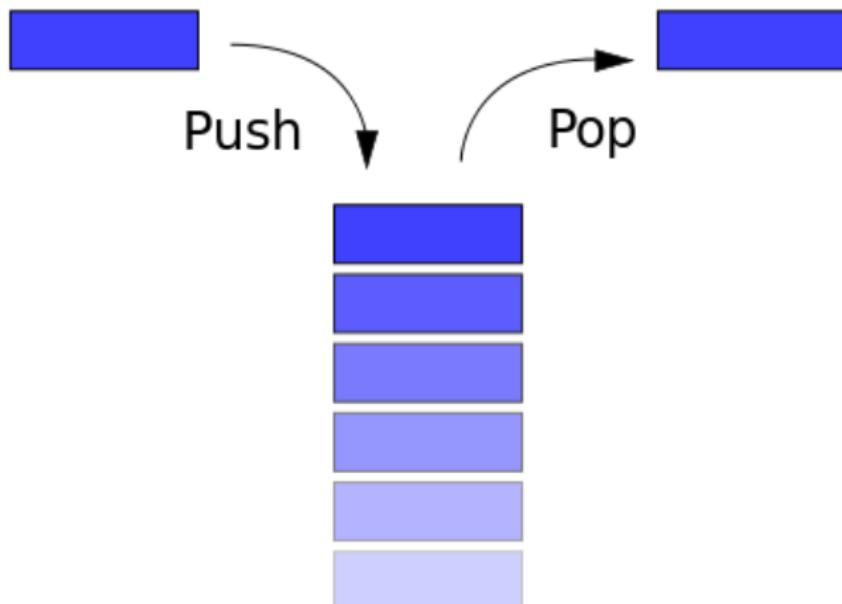
Revisão sobre Estruturas de Dados

Pilhas

Pilhas

- Uma **pilha** é uma estrutura de dados que representa um caso particular de lista, de forma a implementar uma restrição de acesso a apenas uma das extremidades da estrutura.
- O elemento na extremidade editável (geralmente o fim da lista) da estrutura é denominado *topo da pilha*.
- Trabalha através do princípio '**Last-In First-out**' - **LIFO**, ou seja, o último elemento que entrou será o primeiro a sair da estrutura.
- A operação de inserção de elementos na pilha é denominado **empilhar** (*push*).
- A operação de remoção de elementos na pilha é denominado **desempilhar** (*pop*).

Pilhas



Pilhas - Exercício

- Implemente um programa (em C) capaz de ler uma sequência de parênteses e retorne “**Correto**” caso haja correspondência de cada símbolo “(” com seu respectivo “)”, e “**Incorreto**” caso contrário
- Modifique o programa anterior para incluir “{ }” e “[]”
- Exemplos de expressões corretas
 - (() ())
 - (((())) () (()))
 - () () () () ()
- Exemplos de expressões incorretas
 - (() ()
 - ((())))
 - ([{ })]

Revisão sobre Estruturas de Dados

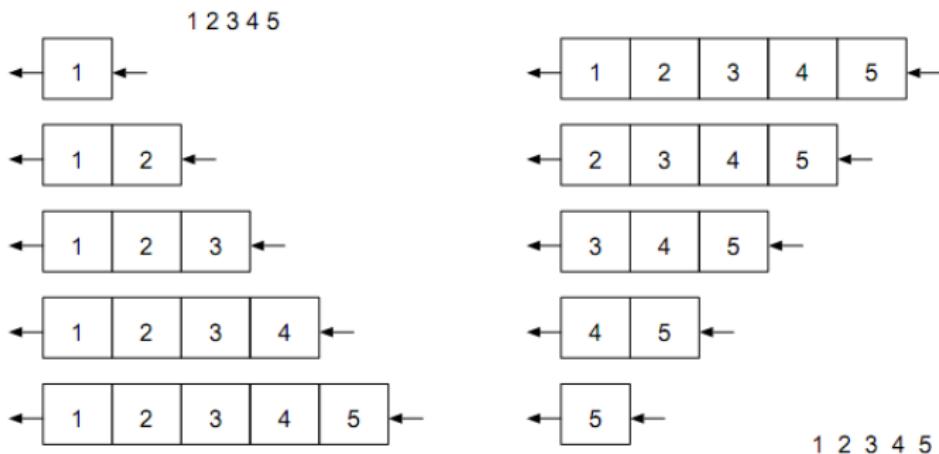
Filas

Filas

- Uma **fila** é mais uma estrutura de dados que representa um caso particular de lista. Implementa uma restrição de acesso para a inserção a apenas uma das extremidades e remoção na outra.
- O elemento posicionado no início da estrutura é denominado *cabeça da fila* e o elemento no fim da fila é a *cauda*.
- Trabalha através do princípio '**First-In First-out**' - **FIFO**, ou seja, o primeiro a entrar é o primeiro a sair da estrutura.
- A operação de inserção de elementos na fila é denominado **empilhar** (*push*) e é sempre realizado pela cauda da fila.
- A operação de remoção de elementos na pilha é denominado **desempilhar** (*pop*) e é sempre realizado pela cabeça da fila.

Filas

Funcionamento de filas



Filas - Maratona de Programação

Descrição do Problema: Dado um baralho contendo n cartas ordenadas (numeradas de 1 a n). A seguinte operação é realizada sucessivas vezes enquanto ainda houverem pelo menos duas cartas no baralho: “*descarte a carta no topo do baralho, mova o agora novo topo para o fim do baralho*”

Seu trabalho é encontrar a sequência de cartas descartadas e qual foi a carta restante.

Entrada de Dados: deve ser fornecido o número n ($n \leq 50$) que indica o total de cartas do baralho.

Saída de Resultados: devem ser reportadas duas linhas: na primeira, deve ser exibida a sequência de cartas descartadas e na segunda, qual foi a carta restante no baralho ao final do ‘jogo’.

Exemplo: para $n = 7$ temos Descarte: 1, 3, 5, 7, 4, 2 e Restou: 6

Revisão sobre Estruturas de Dados

Árvores

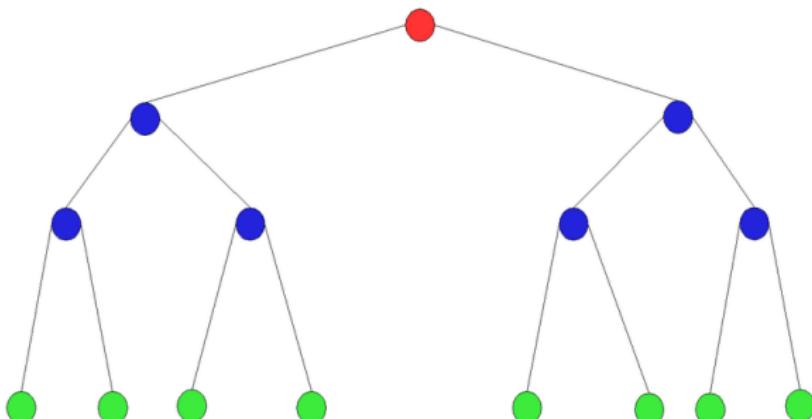
Árvores

- Conceitualmente difere das estruturas listas, pilhas e filas pelo fato de ser uma representação hierárquica (em níveis) ao invés de uma estrutura sequencial
- Essa mudança de representação tem impacto nos processos de inserção, busca e remoção de nós da estrutura
- Uma árvore genérica é composta por um elemento principal denominado **raiz** que contém referências para k outros elementos similares denominados **filhos** do nó. Recursivamente cada filho, é uma sub-raiz para outra árvore similar
- Os nós que não contém filhos são chamados de **folhas** ou terminal
- O número máximo de filhos (k) que cada nó pode conter é chamado de **ordem** da árvore
- Caso particular de interesse: $K = 2$ (árvore binária)
- Durante o decorrer do semestre, ordens mais altas de árvores serão estudadas

Árvores Binárias

Define-se uma árvore binária como uma estrutura de dados que:

- ou não contém nenhum nó (árvore vazia);
- ou contém um nó (raiz) que contém dois ponteiros para os elementos raiz de outras duas 'sub-árvores' (denominadas *esquerda* e *direita*)

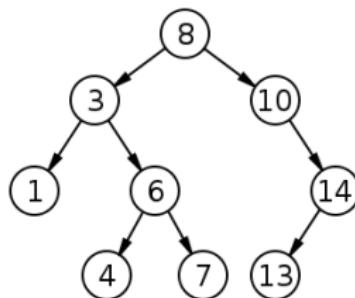


Árvores Binárias - Declaração do TDA

```
1   struct TDA {  
2       void * data ;  
3       struct TDA * ptrLeft ,  
4                   * ptrRight;  
5   };  
6  
7   struct TDA * ptrRoot = malloc( sizeof( struct TDA ));  
8   ptrRoot->data = info;  
9   ptrRoot->ptrLeft = ptrRoot->ptrRight = NULL;  
10
```

Árvores Binárias de Busca - ABB

- é um caso particular de árvore binária onde se propõe uma organização aos dados inseridos a fim de facilitar a ‘busca’ por determinados elementos na hierarquia da árvore
- Critério de organização: para cada raiz (ou sub-raiz) da árvore os elementos menores (que o dado armazenado na própria raiz) são armazenados na sub-árvore da esquerda, e os elementos maiores na sub-árvore da direita



Buscando elementos em uma ABB

1. Caso o elemento na raiz seja a chave que procuramos = chave encontrada!
2. Caso a chave procurada seja menor, verifique recursivamente a raiz da sub-árvore à esquerda
3. Caso a chave procurada seja maior, verifique recursivamente a raiz da sub-árvore à direita
4. Caso a respectiva sub-árvore esteja vazia = chave inexistente!

Buscando elementos em uma ABB

```
1 TDA *busca( TDA* ptrRoot, int k) {
2     if (!ptrRoot || r->data == k)
3         return ptrRoot;
4     if (ptrRoot->data > k)
5         return busca( ptrRoot->ptrLeft, k);
6     else
7         return busca( ptrRoot->ptrRight, k);
8 }
9 }
```

Predecessor e Sucessor de um Nô

- Um **nô predecessor** a um elemento X em uma ABB é definido como o "*maior elemento na árvore que seja menor que X* ", ou seja, o maior dos elementos à esquerda de X .
- Um **nô sucessor** a um elemento X em uma ABB é definido como o "*menor elemento na árvore que seja maior que X* ", ou seja, o menor dos elementos à direita de X .

Predecessor e Sucessor de um Nô

```
1 TDA * Pred( TDA **x ) {
2     if (x->ptrLeft != NULL) {
3         TDA *y = x->ptrLeft;
4         while (y->ptrRight != NULL) y = y->ptrRight;
5         return y;
6     }
7     return NULL;
8 }
```



```
1 TDA * Succ( TDA **x ) {
2     if (x->ptrRight != NULL) {
3         TDA *y = x->ptrRight;
4         while (y->ptrLeft != NULL) y = y->ptrLeft;
5         return y;
6     }
7     return NULL;
8 }
```

Inserindo elementos em uma ABB

1. Caso o elemento raiz esteja vazio: insira o novo elemento na raiz
2. Enquanto o elemento raiz não seja o vazio:
 - 2.1 caso o novo elemento seja menor que a raiz: consulte a raiz da sub-árvore à esquerda
 - 2.2 caso contrário: consulte a raiz da sub-árvore à direita
 - 2.3 caso a respectiva sub-árvore esteja vazia: insira o novo elemento naquela posição

Atenção! Deve-se determinar um critério para a inserção de elementos repetidos

Inserindo elementos em uma ABB

```
1 TDA* insere( TDA* ptrRoot, TDA * ptrNew) {
2     TDA *ptrCurr, *ptrPrev;
3
4     if (!ptrRoot)
5         ptrRoot = ptrNew;
6
7     else {
8         ptrCurr = ptrRoot;
9
10        while (ptrCurr) {
11            ptrPrev = ptrCurr;
12
13            if (ptrCurr->data > ptrNew->data) ptrCurr = ptrCurr->ptrLeft;
14            else    ptrCurr = ptrCurr->ptrRight;
15        }
16
17        if (ptrPrev->data > ptrNew->data) ptrPrev->ptrLeft = ptrNew;
18        else    ptrPrev->ptrRight = ptrNew;
19    }
20    return ptrRoot;
21 }
22 }
```

Exercício: Inserir as seguintes chaves = 8, 3, 1, 6, 5, 4, 9

Removendo elementos em uma ABB

1. Caso o elemento raiz não contenha um dos filhos, então faça o outro filho se tornar a nova raiz
2. Caso o elemento raiz contenha os dois filhos, faça o predecessor ou o sucessor do elemento ser a nova raiz

Removendo elementos em uma ABB

```
1 TDA* removeraiz( TDA* ptrRoot ) {
2     TDA *ptrPrev, *ptrCurr;
3     if ( !ptrRoot->ptrLeft) {
4         ptrCurr = ptrRoot->ptrRight;
5         free(ptrRoot);
6         return ptrCurr;
7     }
8
9     ptrPrev = ptrRoot;
10    ptrCurr = ptrRoot->ptrLeft;
11
12    while (ptrCurr->ptrRight) {
13        ptrPrev = ptrCurr;
14        ptrCurr = ptrCurr->ptrRight;
15    }
16
17    if (ptrPrev != ptrRoot) {
18        ptrPrev->ptrRight = ptrCurr->ptrLeft;
19        ptrCurr->ptrLeft = ptrRoot->ptrLeft;
20    }
21    ptrCurr->ptrRight = ptrRoot->ptrRight;
22    free(ptrRoot);
23    return ptrCurr;
24 }
```

Percursos em uma ABB

Há 3 formas de se percorrer uma ABB:

Pre-order segue a sequência **raiz** → **esquerda** → **direita**, ou seja:

1. visita-se o nó raiz;
2. visita-se recursivamente a sub-árvore da esquerda;
3. visita-se recursivamente a sub-árvore da direita;

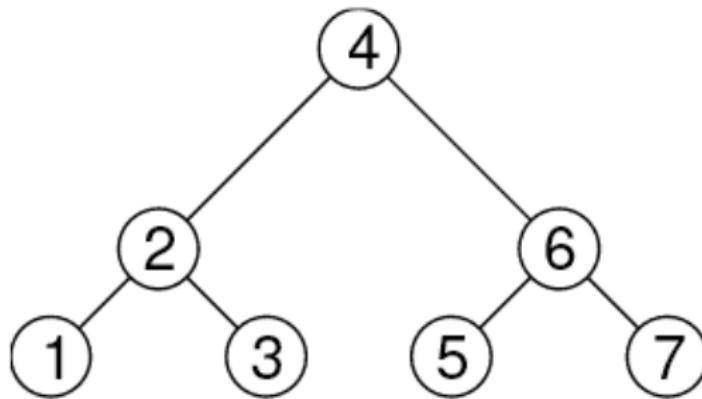
In-order segue a sequência **esquerda** → **raiz** → **direita**, ou seja:

1. visita-se recursivamente a sub-árvore da esquerda;
2. visita-se o nó raiz;
3. visita-se recursivamente a sub-árvore da direita;

Pos-order segue a sequência **esquerda** → **direita** → **raiz**, ou seja:

1. visita-se recursivamente a sub-árvore da esquerda;
2. visita-se recursivamente a sub-árvore da direita;
3. visita-se o nó raiz;

Percorso em uma ABB - *Pre-Order*

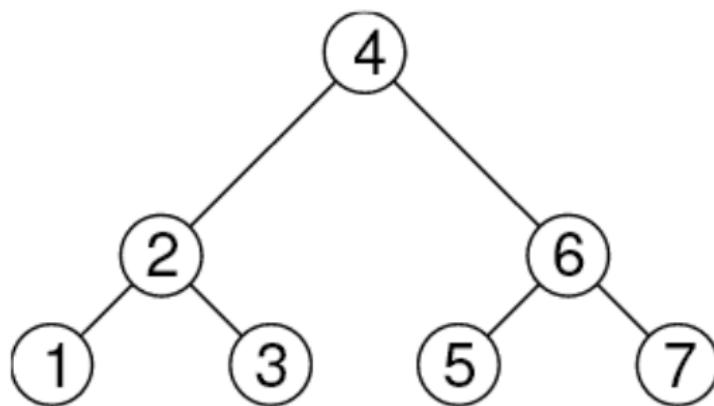


4 2 1 3 6 5 7

Percorso em uma ABB - *Pre-Order*

```
1 void PreOrder( TDA * ptrRoot ) {  
2     if( ptrRoot ) {  
3         process( ptrRoot );  
4         PreOrder( ptrRoot->ptrLeft );  
5         PreOrder( ptrRoot->ptrRight );  
6     }  
7 }
```

Percorso em uma ABB - *In-Order*

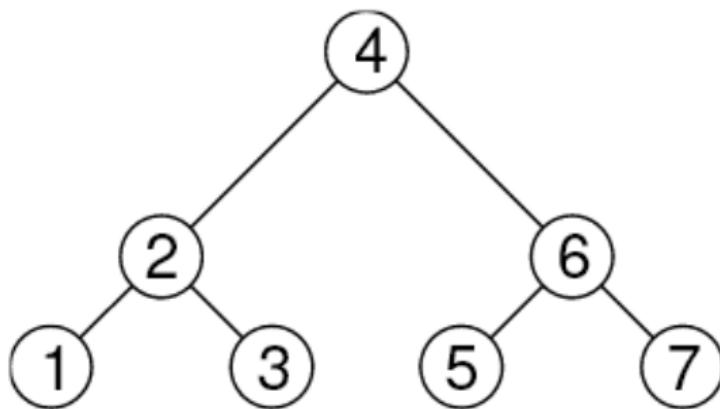


1 2 3 4 5 6 7

Percorso em uma ABB - *In-Order*

```
1 void InOrder( TDA * ptrRoot ) {  
2     if( ptrRoot ) {  
3         InOrder( ptrRoot->ptrLeft );  
4         process( ptrRoot );  
5         InOrder( ptrRoot->ptrRight );  
6     }  
7 }
```

Percorso em uma ABB - *Pos-Order*



1 3 2 5 7 6 4

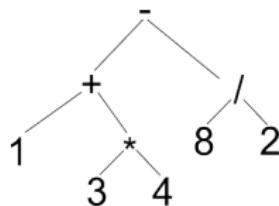
Percorso em uma ABB - *Pos-Order*

```
1 void PosOrder( TDA * ptrRoot ) {  
2     if( ptrRoot ) {  
3         PosOrder( ptrRoot->ptrLeft );  
4         PosOrder( ptrRoot->ptrRight );  
5         process( ptrRoot );  
6     }  
7 }
```

Percorso em uma ABB - Exercício

- Crie um programa que, dada uma expressão aritmética representada através de uma árvore binária, a converta para sua equivalente '*notação polonesa*':

$$1 + 3 * 4 - 8 / 2 \Rightarrow 1\ 3\ 4\ * + 8\ 2\ / -$$



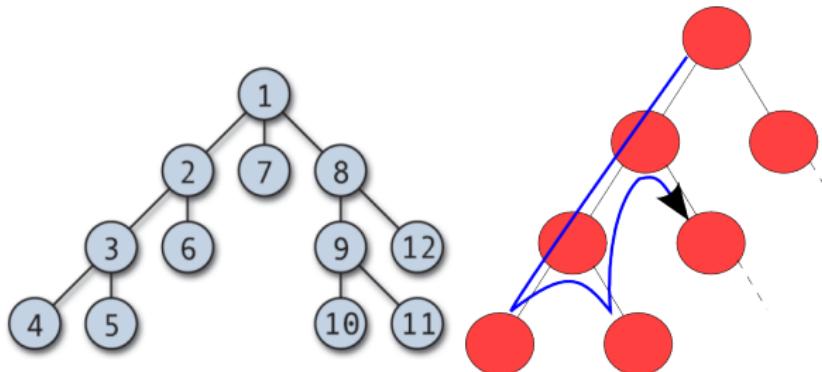
- Considere:
 - Cada número contém apenas 1 dígito
 - Precedência dos operadores de acordo com as regras da Álgebra
- Resolva a expressão resultante (por pilha) e apresente o resultado

Percorso em uma Árvore de Ordem N

- Algoritmos de busca tem por objetivo percorrer a estrutura da árvore (qualquer ordem) a fim de localizar um elemento específico.
- Duas abordagens:
 - **Busca em Profundidade:** (ou *depth-first search* - DFS) objetiva encontrar rapidamente um elemento folha específico na árvore
 - **Busca em Largura:** (ou *breadth-first search* - BFS) objetiva encontrar a menor distância entre a raiz e um elemento específico na árvore

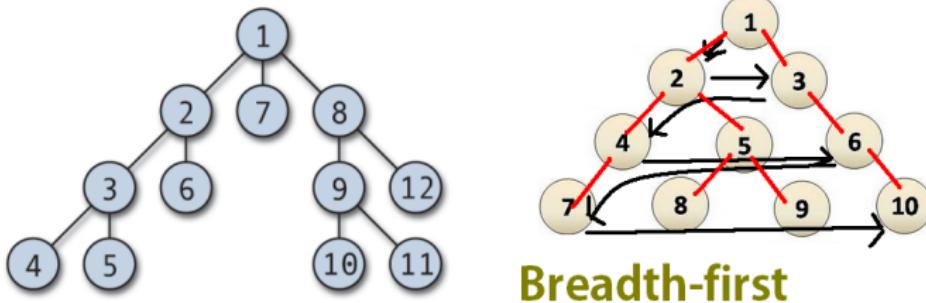
Percorso em uma Árvore de Ordem N - Profundidade

- O percurso em profundidade tem por princípio percorrer cada ‘ramo’ da árvore da raiz até a folha, antes de decidir pelo *retrocesso (backtracking)*, a fim de continuar a busca no ramo subsequente.



Percorso em uma Árvore de Ordem N - Largura

- O percurso em largura tem por princípio percorrer cada ‘nível’ da árvore partindo do nível raiz até o nível das folhas, com o objetivo de otimizar a distância entre a raiz e um **elemento-objetivo**.



Programação Orientada a Objetos (C++)

Classe & Objeto **Classes** são descrições expandidas de estruturas de dados que podem conter *membros*: atributos e métodos. **Objetos** são instanciações das classes. Objetos podem ser estáticos ou dinâmicos.

Políticas de Acesso são identificadores que definem o escopo de acesso aos membros de uma classe. Podem ser públicos, privados ou protegidos.

Construtores & Destruitor **Construtor** é um método especial que é chamado no momento em que um objeto é criado. **Destruitor** é o método chamado no momento em que um objeto é desalocado da memória.

Uma classe pode conter vários construtores (polimorfismo) porém apenas um destrutor.

Programação Orientada a Objetos (C++)

```
1 class Circle
2 {
3     private :
4         int raio ;
5         float centro_x , centro_y ;
6     public :
7         Circle ();           // default
8         Circle ( float , float , int ); // inicializador
9         Circle ( const Circle &);    // clonagem
10        Circle ( Circle *);       // clonagem dinamica
11
12        ~ Circle ();
13
14        void setRadius( int );
15        void setCenter( float , float );
16    };
17
18 ...
19
20 Circle circle , circle1 ( 1.0f,-2.0f, 5), circle2 ( circle1 );
```

Programação Orientada a Objetos (C++)

Ponteiros para **Classes** referênciação dinâmica a objetos de uma classe. Uso dos operadores **new** e **delete**.

Referenciação dinâmica através do operador '**->**'.

Parâmetro **Default** permite especificar um valor inicial a um parâmetro de um método caso um valor formal não seja fornecidos no momento da sua chamada.

Métodos **inline** método que não produz desvio de execução nas chamadas (substituição de código).

Programação Orientada a Objetos (C++)

```
1 class Circle
2 {
3     private :
4         int raio ;
5         float centro_x , centro_y ;
6     public :
7         Circle (); // default
8         Circle ( float , float , int = 1); // inicializador aceita 2 ou 3 parametros
9         Circle (const Circle &); // clonagem
10        Circle ( Circle *); // clonagem dinamica
11
12        ~ Circle ();
13
14        void setRadius( int );
15        inline int getRadius() { return this ->raio; }
16        void setCenter( float , float );
17    };
18 ...
19 Circle * ptrCircle = new Circle(), circle (-1,-1);
20 ptrCircle ->setRadius(10);
21 (* ptrCircle ).setCenter(-1.0f, 3.4f);
22 ...
23 delete ptrCircle ;
```

Programação Orientada a Objetos (C++)

Cabeçalho & Implementação abordagem popular para especificação do código-fonte em um programa C++.

Etapas:

1. Criar um projeto (geralmente uma Console Application)
2. Incluir nos arquivos de cabeçalhos (*.h) apenas os protótipos para a classe, exceto nos casos de funções *inline* e descrição de templates.

Importante :! os cabeçalhos devem ser envolvidos por uma diretiva de compilação `#ifndef CONST` e `#endif` a fim de evitar erros de duplicidade de compilação.

3. Descrever nos arquivos de implementação (*.cpp) as implementações para os métodos declarados nos protótipos

Pré-inicializadores em C++ é possível efetuarmos inicializações de atributos mesmo antes da execução do construtor. Para tal declara-se na implementação do construtor uma seção com o operador ‘:’ seguido de uma lista de variáveis e seus respectivos valores entre parênteses

Programação Orientada a Objetos (C++)

circle.h

```

1 #ifndef _CIRCLE_H_
2 #define _CIRCLE_H_
3
4 class Circle
5 {
6     private :
7         int raio ;
8         float centro_x , centro_y ;
9     public :
10        Circle ();
11        Circle ( float , float , int = 1 );
12        Circle ( const Circle & );
13        Circle ( Circle * );
14
15        ~ Circle ();
16
17        void setRadius( int );
18        inline int getRadius()
19        {
20            return this ->raio;
21        }
22        void setCenter( float , float );
23    };
24 #endif

```

circle.cpp

```

1 #include " circle .h"
2
3     Circle :: Circle () :
4         raio (0),
5         centro_x (0.0f),
6         centro_y (0.0f)
7     {
8
9     ...
10
11    void Circle :: setRadius( int raio )
12    {
13        this ->raio = raio;
14    }
15
16    ...

```

Programação Orientada a Objetos (C++)

Sobrecarga & Polimorfismo definição de múltiplas funcionalidades para um mesmo membro da classe. Dois tipos: método ou operador.

- Na **sobrecarga de operador**, redefine-se a funcionalidade de um operador (existente na linguagem C). Por exemplo: operador atribuição para a classe Circle
- Através do **polimorfismo de método** é possível se definir métodos que executem diferentes ações em diferentes contextos.

Programação Orientada a Objetos (C++)

```
1 class Circle
2 {
3     private :
4         int raio ;
5         float centro_x, centro_y;
6     public :
7         Circle ( float = 0.0f, float = 0.0f, int = 1); // inicializador aceita ate 3 parametros
8         Circle (const Circle &);           // clonagem
9         Circle ( Circle *);             // clonagem dinamica
10
11     ~ Circle ();
12
13     Circle operator = (const Circle &);
14     bool    operator == (const Circle &);
15 }
```

Programação Orientada a Objetos (C++)

Herança especificação de uma hierarquia de classes onde uma classe-pai compartilha seus membros públicos e protegidos com suas classes subordinadas.

Um classe-pai pode ter mais de uma classe subordinada, bem como uma classe pode estar subordinada a mais de uma classe-pai, porém não pode haver ambiguidade de herança (ciclos).

Tipos de herança:

Pública os membros (públicos e protegidos) da classe-pai serão todos herdados e a política de acesso será mantida

Protegida os membros (públicos e protegidos) da classe-pai serão todos herdados porém usando acesso protegido

Privada os membros (públicos e protegidos) da classe-pai serão todos herdados porém usando acesso privado.

Programação Orientada a Objetos (C++)

```
1 class Mother {
2     public:
3         Mother ()
4             { cout << "Mother: no parameters\n"; }
5         Mother (int a)
6             { cout << "Mother: int parameter\n"; }
7     };
8
9 class Daughter : public Mother {
10    public:
11        Daughter (int a)
12            { cout << "Daughter: int parameter\n\n"; }
13    };
14
15 class Son : public Mother {
16    public:
17        Son (int a) : Mother (a)
18            { cout << "Son: int parameter\n\n"; }
19    };
```

Programação Orientada a Objetos (C++)

Membros Estáticos (ou variáveis de classe) são membros que podem ser consultados externamente porém só podem ser modificados pelos próprios métodos da classe

Membros e Classes Constantes não podem ser modificados, apenas consultados

Programação Orientada a Objetos (C++)

```
1 class Dummy {  
2     public:  
3         static int n;  
4         Dummy () { n++; };  
5         ~Dummy () { n--; };  
6     };  
7  
8     int Dummy::n=0;  
9  
10    int main () {  
11        Dummy a;  
12        Dummy b[5];  
13        Dummy * c = new Dummy;  
14        cout << a.n << '\n';  
15        delete c;  
16        cout << Dummy::n << '\n';  
17        return 0;  
18    }
```

```
1     int main() {  
2         const MyClass foo(10);  
3  
4         // invalido : x nao pode ser modificado  
5         // foo.x = 20;  
6  
7         // ok: os membros da classe podem ser acessados  
8         cout << foo.x << '\n';  
9  
10        return 0;  
11    }
```

Programação Orientada a Objetos (C++)

Templates permitem a especificação de tipos parametrizáveis para classes e membros

Programação Orientada a Objetos (C++)

```
1 #include <iostream>
2 using namespace std;
3
4 template <class T>
5 class mypair {
6     T a, b;
7     public:
8     mypair (T first , T second)
9         {a=first ; b=second;}
10    T getmax ();
11 }
```

```
1 template <class T>
2 T mypair<T>::getmax ()
3 {
4     T retval ;
5     retval = a>b? a : b;
6     return retval ;
7 }
```

```
1 int main () {
2     mypair <int> myobject (100, 75);
3     cout << myobject.getmax();
4     return 0;
5 }
```

Gerador de Documentação de Código

- Ferramenta sugerida: DOXYGEN
 - <http://www.doxygen.org/>
- Originalmente proposta para C++, hoje permite suporte a múltiplas linguagens: C, Objective-C, C#, PHP, Java, Python, IDL, Fortran, VHDL, Tcl
- Suporte a Windows, Mac OS X e Linux
- Três possíveis aplicações:
 1. Gerador online de documentação (HTML) e offline (\LaTeX , RTF, PS, PDF, CHM, man pages)
 2. Visualizador de relações entre os arquivos não documentados do código-fonte
 3. Gerador de documentação normal

Gerador de Documentação de Código

Uso do Doxygen para C++:

- A documentação é realizada dentro de blocos especiais de comentários definidos por:

```
/**  
...  
*/
```

- Alternativamente, pode-se adotar o comentário por linha, como em

```
/// ...  
//! ...  
//< ...
```

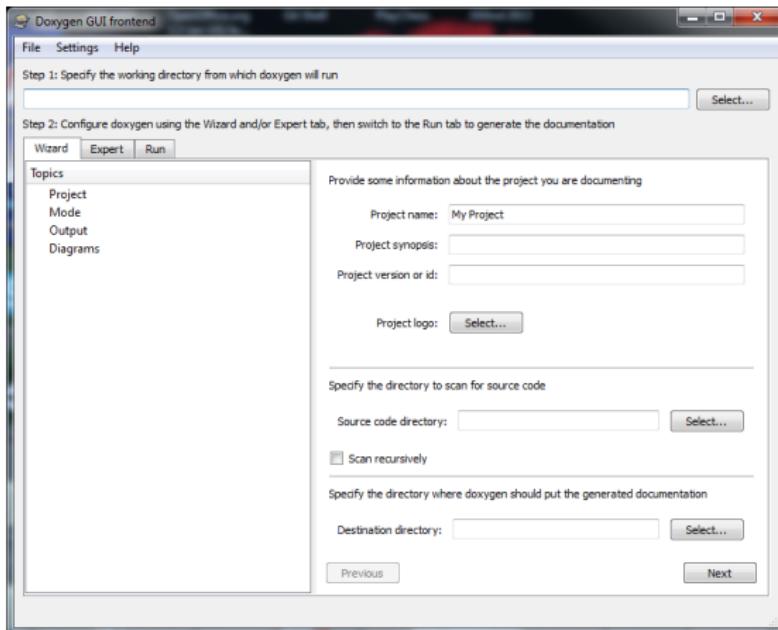
Gerador de Documentação de Código

- \file
- \class
- \brief
- \param
- \sa (*see also*)
- \struct
- \union
- \enum
- \fn (*function*)
- \def (#*define*)
- \typedef
- \namespace

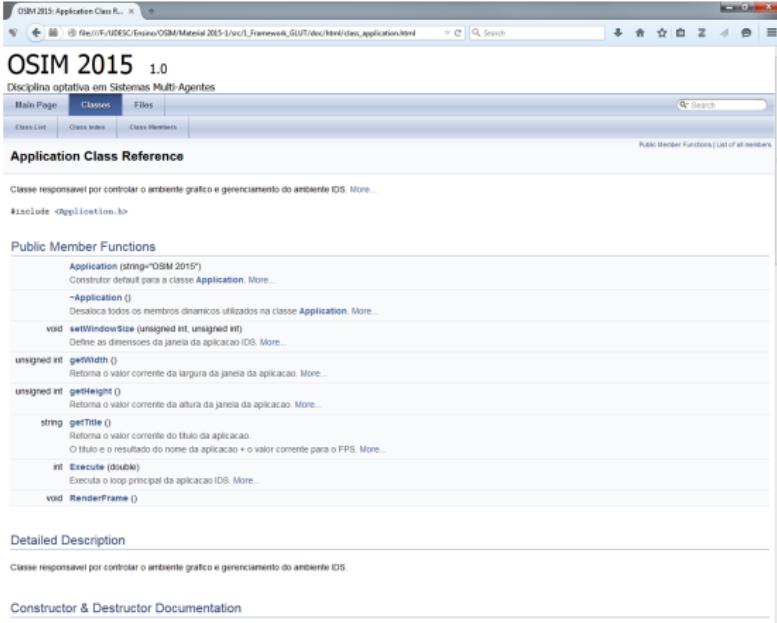
Gerador de Documentação de Código

```
1  /**
2   \ file Application.h
3   \ brief Definicao do prototipo da classe Application para gerencimento do IDS<p>
4
5   Desenvolvido por <b>Rogerio Eduardo da Silva</b><br>
6   Fevereiro , 2015. Universidade do Estado de Santa Catarina (UDESC)
7 */
8 #include <string>
9 using namespace std;
10
11 /**
12  \ class Application
13  \ brief Classe responsavel por controlar o ambiente grafico e gerenciamento do ambiente IDS.
14 */
15 class Application
16 {
17 private :
18     string      strAppName,    //! nome do titulo da janela da aplicacao
19                 strCurrentFPS; //! valor corrente (convertido em texto) do FPS disponivel para execucao da aplicacao
20     unsigned int uiWindowWidth, //! largura da janela da aplicacao
21                 uiWindowHeight, //! altura da janela da aplicacao
22                 uiFrames; //! contador de frames para o calculo do FPS
23     double      dCounterTimer; //! contador de tempo para o calculo do FPS
24
25     void setWindowSize( unsigned int, unsigned int );
26 /**
27  \ brief Retorna o valor corrente da largura da janela da aplicacao
28  \return unsigned int o valor da largura da janela
29 */
30     ...
31 }
```

Gerador de Documentação de Código



Gerador de Documentação de Código



The screenshot shows a web browser displaying the "OSIM 2015 Application Class Reference". The title bar reads "OSIM 2015: Application Class R...". The main content area has a header "OSIM 2015 1.0" and "Disciplina optativa em Sistemas Multi-Agentes". Below this is a navigation menu with tabs: Main Page, Classes (which is selected), and Files. Sub-tabs under Classes are Class List, Class Index, and Class Members. A search bar is at the top right. The main content is titled "Application Class Reference". It describes the class as responsible for controlling the graphical environment and managing the iOS window. It includes sections for "Public Member Functions" and "Detailed Description". The "Public Member Functions" section lists methods like Application, ~Application, setWindowSize, getTitle, getHeight, getWidth, setTitle, execute, and RenderFrame, each with a brief description. The "Detailed Description" section repeats the general description of the class.

OSIM 2015 1.0
Disciplina optativa em Sistemas Multi-Agentes

Main Page Classes Files

Class List Class Index Class Members

Search

Application Class Reference

Classe responsável por controlar o ambiente gráfico e gerenciamento do ambiente iOS. More...

#include <Application.h>

Public Member Functions

Application (string="OSIM 2015")
Constructor default para a classe **Application**. More...

~Application ()
Desaloca todos os membros dinâmicos utilizados na classe **Application**. More...

void setWindowSize (unsigned int, unsigned int)
Define as dimensões da janela da aplicação iOS. More...

unsigned int getTitle ()
Retorna o valor corrente da largura da janela da aplicação. More...

unsigned int getHeight ()
Retorna o valor corrente da altura da janela da aplicação. More...

string getTitle ()
Retorna o valor corrente do título da aplicação.
O título é o resultado do nome da aplicação + o valor corrente para o FPS. More...

vt Execute (double)
Executa o loop principal da aplicação iOS. More...

void RenderFrame ()

Detailed Description

Classe responsável por controlar o ambiente gráfico e gerenciamento do ambiente iOS.

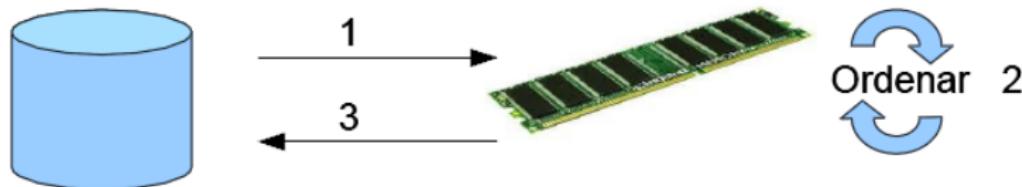
Constructor & Destructor Documentation

Ordenação Externa

- Método da Intercalação

Ordenação Externa

- **Objetivo:** ordenar os dados armazenados em um arquivo sequencial em memória externa (secundária)
- **Abordagem #1:**
 1. transferir o conteúdo do arquivo para uma estrutura de dados na memória principal (RAM);
 2. ordenar os dados na estrutura através de algum método de ordenação eficiente;
 3. transferir o conteúdo da estrutura de dados resultante novamente para um arquivo sequencial em memória secundária.



Ordenação Externa

- **Problema:** Como ordenar arquivos de tamanho maior que a memória interna disponível?
- **Solução?** Particionar o arquivo em blocos que cabem na memória
- Algoritmos devem diminuir o número de acessos às unidades de memória externa
 - Custo relacionado à transferência de dados
- Dados armazenados como um arquivo sequencial
- Métodos dependentes do estado atual da tecnologia
 - Tipos de memória externa torna os métodos dependentes de vários parâmetros

Método da Intercalação - Algoritmo

1. **Entrada:** 2 pilhas ordenadas ($Pilha_A$ e $Pilha_B$)
2. **Saída:** fila contendo as duas pilhas intercaladas
3. **Processo:** Compara os elementos no topo de cada uma as pilhas ($Topo_A$ e $Topo_B$):
 - 3.1 $Topo_A = Topo_B$: insere qualquer dos elementos na fila de saída e desempilha em ambas as pilhas
 - 3.2 $Topo_A < Topo_B$: insere $Topo_A$ na fila de saída e desempilha da $Pilha_A$
 - 3.3 $Topo_A > Topo_B$: insere $Topo_B$ na fila de saída e desempilha da $Pilha_B$

Ordenação Externa - Intercalação



Método da Intercalação - Algoritmo

```
1 gera_pilhas_ordenadas ( pilha_A , pilha_B );
2 fila_result = NULL;
3
4 while( pilha_A && pilha_B ) {
5     if ( pilha_A.topo == pilha_B.topo ) {
6         elem = pop( pilha_A );
7         pop( pilha_B );
8     }
9     else if ( pilha_A.topo < pilha_B.topo ) {
10        elem = pop( pilha_A );
11    }
12    else {
13        elem = pop( pilha_B );
14    }
15    fila_result = insere_na_fila ( elem );
16 }
17
18 fila_result = copia_para_fila ( pilha_A ? pilha_A : pilha_B );
```

Ordenação Externa por Intercalação - Algoritmo

1. Paginar o arquivo sequencial em blocos de tamanho suportado pela memória principal
2. Cada bloco é então ordenado na memória principal. O bloco ordenado resultante é denominado uma *corrida*
3. As corridas resultantes são então intercaladas através uma árvore estática.
 - O processo de intercalação pode ser realizado em 2 ou mais vias

Ordenação Externa por Intercalação - Exemplo

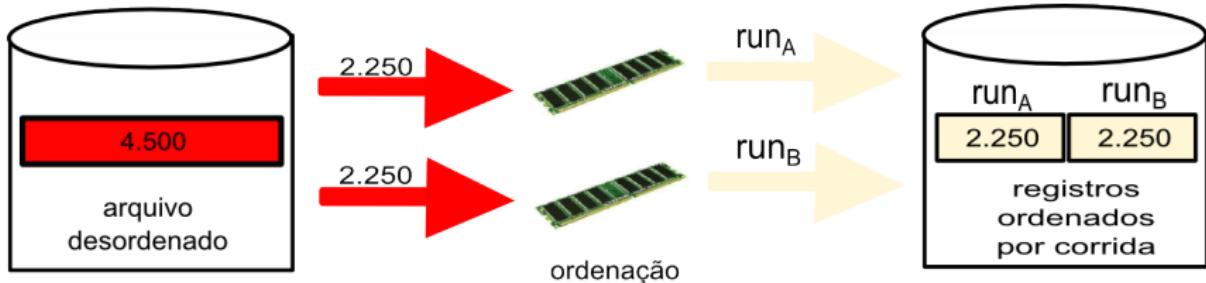
Intercalação em 2 vias

- Classificar um arquivo contendo 4.500 registros num computador com memória RAM capaz de suportar no máximo 2.250 registros.
- O arquivo de entrada está em HD e a unidade de I/O usa blocagem de 250 registros.
- Suponha que há espaço no HD.

Ordenação Externa por Intercalação - Exemplo

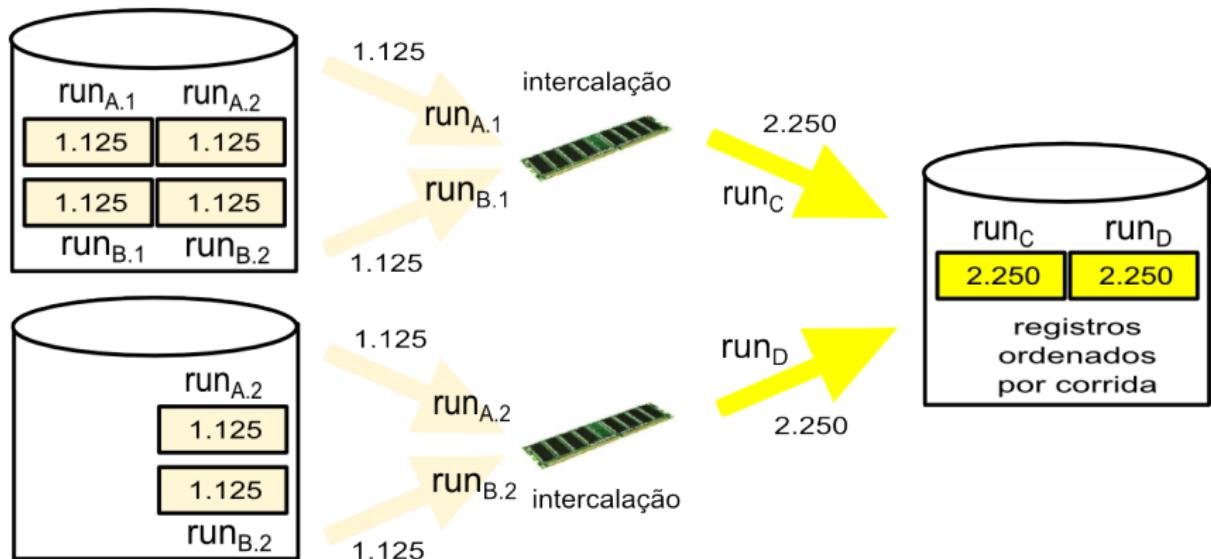
Intercalação em 2 vias

- Arquivo: 4.500 registros
- Unidade de I/O: 250 registros
- RAM: 2.250 registros = 9 operações de I/O
- Supor que há espaço suficiente no HD externo



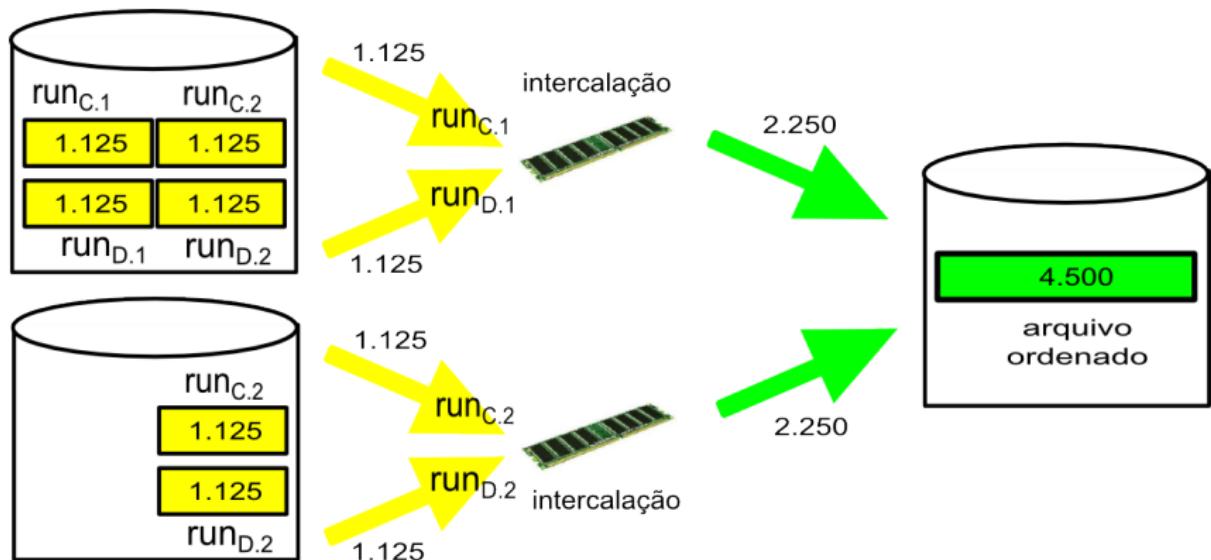
Ordenação Externa por Intercalação - Exemplo

Intercalação em 2 vias



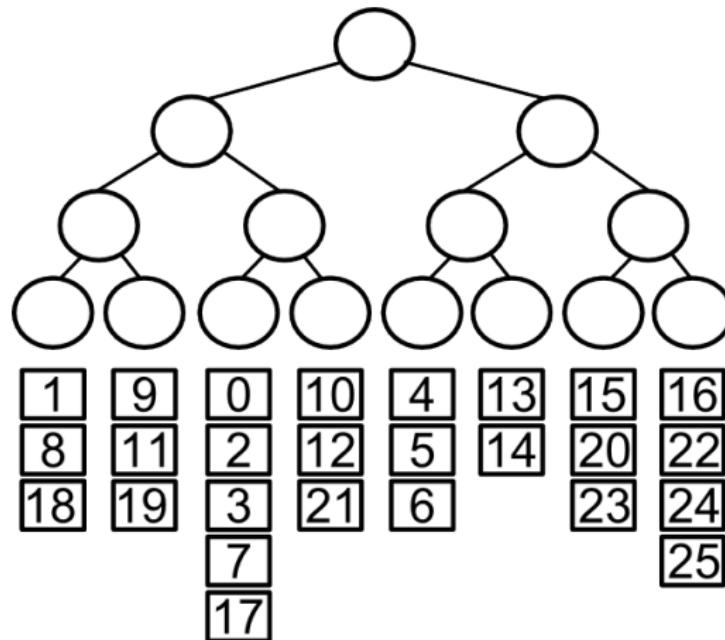
Ordenação Externa por Intercalação - Exemplo

Intercalação em 2 vias



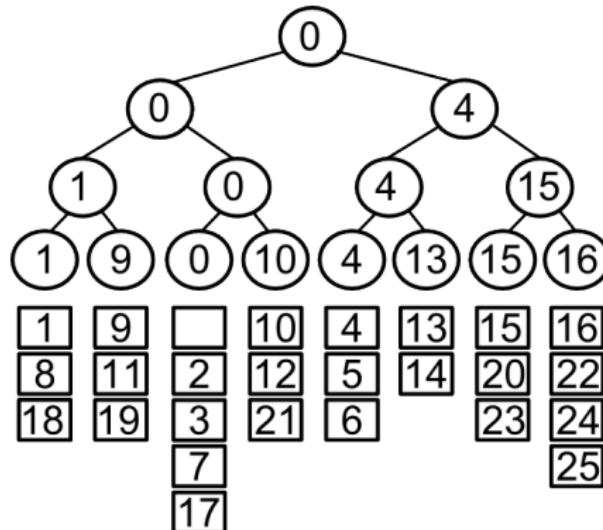
Ordenação Externa por Intercalação

Intercalação em k vias ($k = 8$)



Ordenação Externa por Intercalação

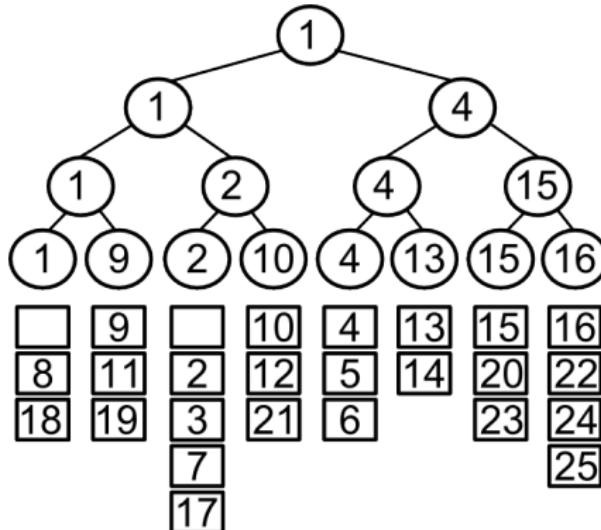
Intercalação em k vias ($k = 8$)



Resultado: 0

Ordenação Externa por Intercalação

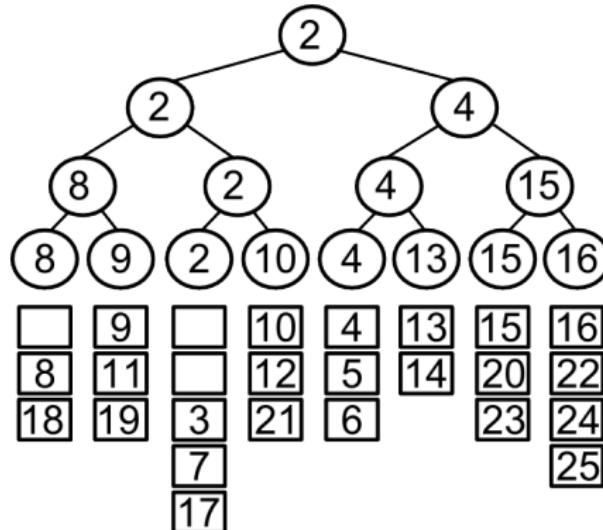
Intercalação em k vias ($k = 8$)



Resultado: 0, 1

Ordenação Externa por Intercalação

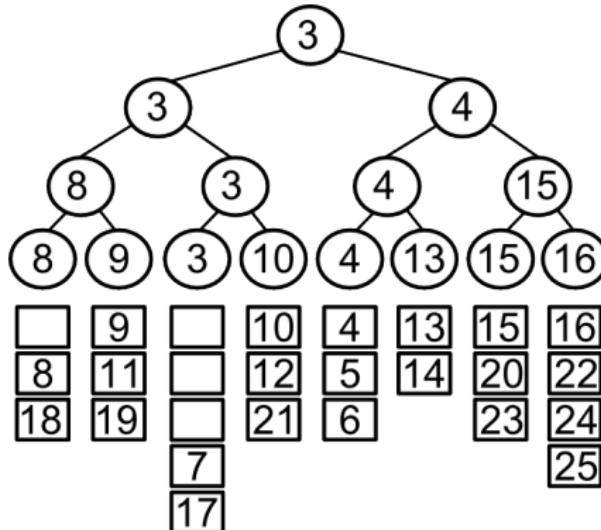
Intercalação em k vias ($k = 8$)



Resultado: 0, 1, 2

Ordenação Externa por Intercalação

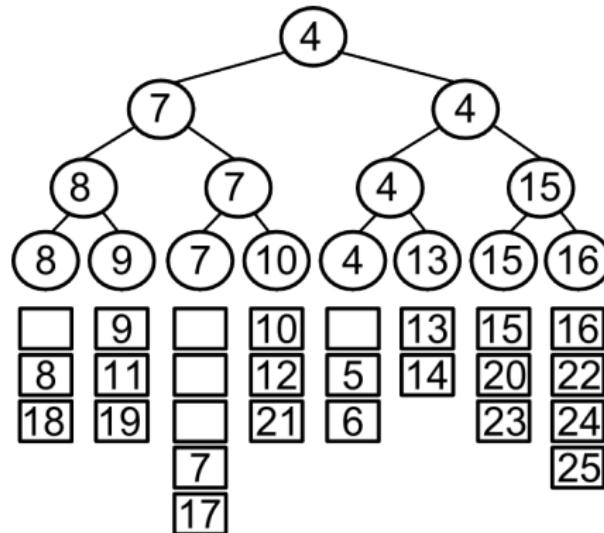
Intercalação em k vias ($k = 8$)



Resultado: 0, 1, 2, 3

Ordenação Externa por Intercalação

Intercalação em k vias ($k = 8$)



Resultado: 0, 1, 2, 3, 4 (continua...)

Ordenação Externa por Intercalação (k -vias) - Algoritmo

1. O arquivo sequencial é de tamanho $\text{size}(\text{file}) >> \text{size(RAM)}$ (arquivo muito maior que o tamanho da memória principal)
2. O arquivo é então sub-dividido em K partições (corridas) de tamanho L , onde L é menor ou igual ao tamanho disponível da memória principal
3. Cada partição K_i é ordenada (em memória interna) separadamente
4. Os próximos passos são repetidos até que todo o arquivo seja processado:
 - 4.1 Carregam-se tantos blocos de dados quanto possível (das partições ordenadas) para a memória principal
 - 4.2 Executa-se o processo de 'árvore de vencedores' armazenando os vencedores em um bloco na fila de saída
 - 4.3 Cada vez que um bloco lido se esvazia, novos blocos são lidos (a partir de uma dada corrida)
 - 4.4 Cada vez que o bloco na fila de saída se enche, grave-o em disco e o esvazie para continuar o processo

Ordenação Externa por Intercalação (k-vias) - Algoritmo

```
1 #define MIN( X, Y ) X->topo < Y->topo ? X : Y
2
3 winner = MIN( MIN( MIN( pilha0, pilha1 ), MIN( pilha2, pilha3 ) ),
4                 MIN( MIN( pilha4, pilha5 ), MIN( pilha6, pilha7 ) ) )
5
6 insere_na_fila ( winner->topo )
7 pop( winner )
```

Avaliação #2 - Merge

Título: Trabalho Prático #2 - Ordenação externa com intercalação de 8 vias

Objetivo: Construir um sistema gerador de tabelas auxiliares de indexação de dados para o banco de dados produzido no trabalho anterior.

Forma de Entrega: Deve ser entregue todos os códigos-fonte (devidamente documentados/comentados), bem como deve ser preparada uma apresentação que inclui uma demonstração de funcionamento do sistema.

Deve ainda ser entregue uma cópia do arquivo construído previamente pelo sistema, que contenha a tabela indexada (por campo à escolha do usuário) e a listagem resultante.

Prazo de Entrega: 07 de Outubro de 2014

Avaliação #2 - Merge

- Critérios:
1. Equipes de 3 alunos
 2. Deve ser utilizada a base de dados de 1 Gb construída no trabalho anterior como base para o processo de indexação.
 3. O programa a ser desenvolvido deve permitir ao usuário:
 - 3.1 Escolher o campo de indexação, a partir de uma lista de opções (que podem incluir chaves compostas se necessário)
 - 3.2 Uma tabela de índices = lista de endereços (em bytes) que referenciam os registros ordenadamente, deve ser gerada para a escolha definida pelo usuário. Um endereço de um registro é o número de bytes que são necessários serem deslocados para se acessar aquela informação;
 - 3.3 Visualizar uma listagem (paginada) dos dados indexados resultantes em um arquivo texto de saída;
 - 3.4 Exibir ao final do processo o tempo gasto para o processamento.

Avaliação #2 - Merge

Exemplo:

| | | | index: NOME | | index: DATA+NOME | |
|----|--------------------|------------|-------------|-----|------------------|-----|
| 01 | JOÃO DA SILVA | 01/02/2012 | 0125 | #2 | 0875 | #8 |
| 02 | ANA PAULA PEREIRA | 05/10/1985 | 0375 | #4 | 0250 | #3 |
| 03 | MIGUEL TORGÀ | 30/05/1952 | 0750 | #7 | 0125 | #2 |
| 04 | FERNANDA BLARGH | 15/12/1999 | 0000 | #1 | 0750 | #7 |
| 05 | JOSÉ MARIA MARTINS | 10/10/2010 | 0500 | #5 | 0375 | #4 |
| 06 | PEDRO PAULA POWER | 05/05/2005 | 0875 | #8 | 1125 | #10 |
| 07 | JOANA MARANHÃO | 03/09/1997 | 1000 | #9 | 1000 | #9 |
| 08 | LUÍZ MARCOS SOUZA | 01/01/1911 | 1125 | #10 | 0625 | #6 |
| 09 | LUÍZA MARIA SOUZA | 05/05/2005 | 0250 | #3 | 0500 | #5 |
| 10 | MARIA MARTA MOURA | 29/07/2004 | 0625 | #6 | 0000 | #1 |

Tamanho do registro: 125 bytes

Avaliação #2 - Critérios

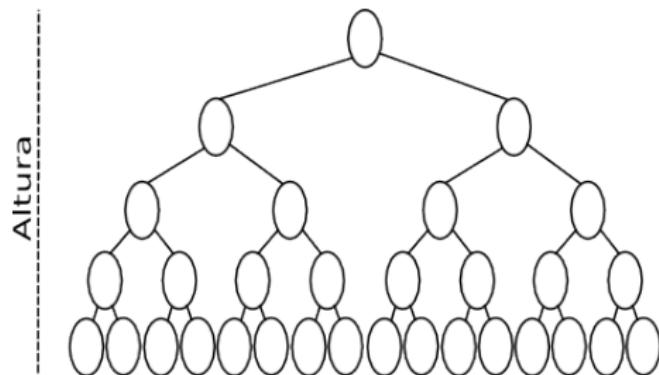
| | |
|------------------------------------------------------------------------------------|------|
| Eficácia | 50 |
| Pontualidade | 10 |
| Organização/Legibilidade do código | 10 |
| Códigos-fontes foram entregues | 20 |
| Documentação | 10 |
| Uso de recursos multimídia | 10 |
| Usa paginação | 20 |
| Intercala em 8-vias | 20 |
| Seleção de campos | 20 |
| Lê/grava blocos de registros | 20 |
| Grava arquivo de índices | 10 |
| Gera relatório de saída | 10 |
| Permite inserção | 20 |
| Permite remoção | 20 |
| Permite consulta | 20 |
| Base de dados 1Gb | 10 |
| Apresenta relatório de desempenho: tempo, nº acessos a disco | 20 |
| | |
| Bônus: análise de performance com múltiplas bases de diferentes tamanhos (min = 4) | 50 |
| Punição por plágio | -300 |
| | |
| Pontuação máxima | 300 |

Árvores Multi-vias

- Árvores B

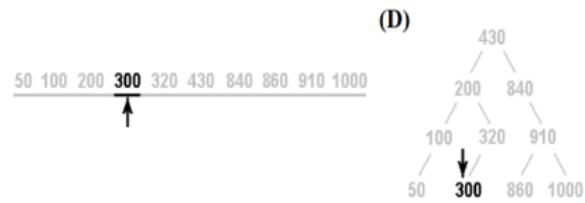
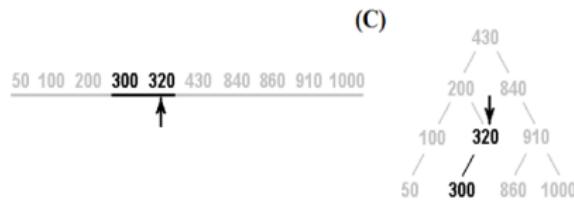
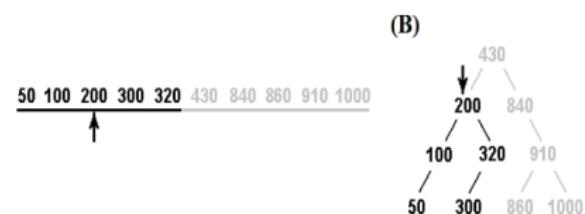
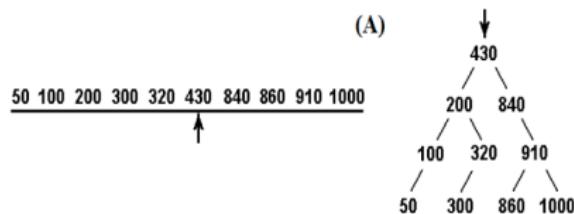
Árvores Multi-vias

- Algoritmos de busca em listas lineares apresentam complexidade $O(n)$, ou seja, a eficiência do algoritmo cresce em proporção linear à quantidade n de elementos do conjunto
- Já árvores binárias de busca (ABB) (quando balanceadas) apresentam complexidade da ordem $O(\log n)$, ou em outras palavras, a eficiência é uma função da altura da árvore



Árvores Multi-vias

Processo de busca do elemento **300**



Árvores Multi-vias

- Teríamos como melhorar a eficiência de acesso a uma árvore ?

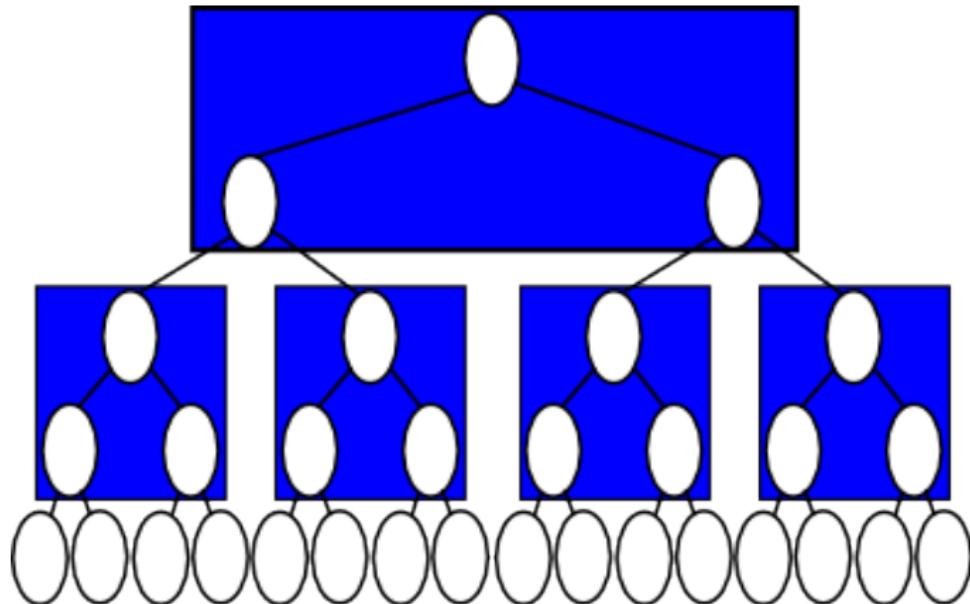
Árvores Multi-vias

- Teríamos como melhorar a eficiência de acesso a uma árvore ?
 - **Sim**, dado que o processo depende da altura da árvore
 - Então, se conseguirmos (para uma mesma quantidade n de elementos), montar uma árvore de altura mais baixa, teríamos uma maior eficiência

Árvores Multi-vias

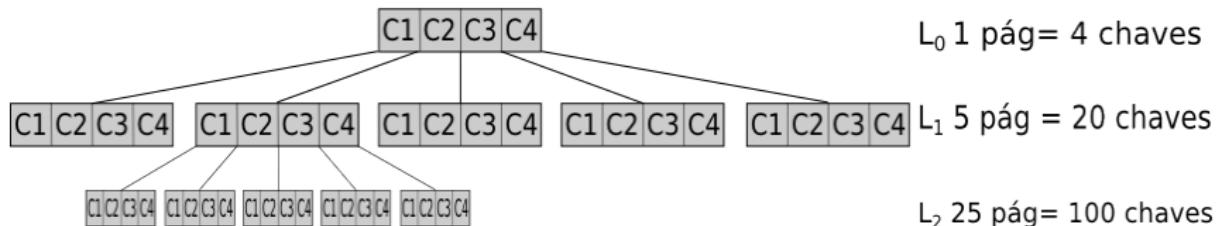
- Teríamos como melhorar a eficiência de acesso a uma árvore ?
 - **Sim**, dado que o processo depende da altura da árvore
 - Então, se conseguirmos (para uma mesma quantidade n de elementos), montar uma árvore de altura mais baixa, teríamos uma maior eficiência
- Uma ABB armazena uma única chave por nó da estrutura. Se ao invés disso, armazenássemos múltiplas chaves por nó (conceito de paginação), teríamos então uma **árvore multivias/multidirecional**.

Árvores Multi-vias

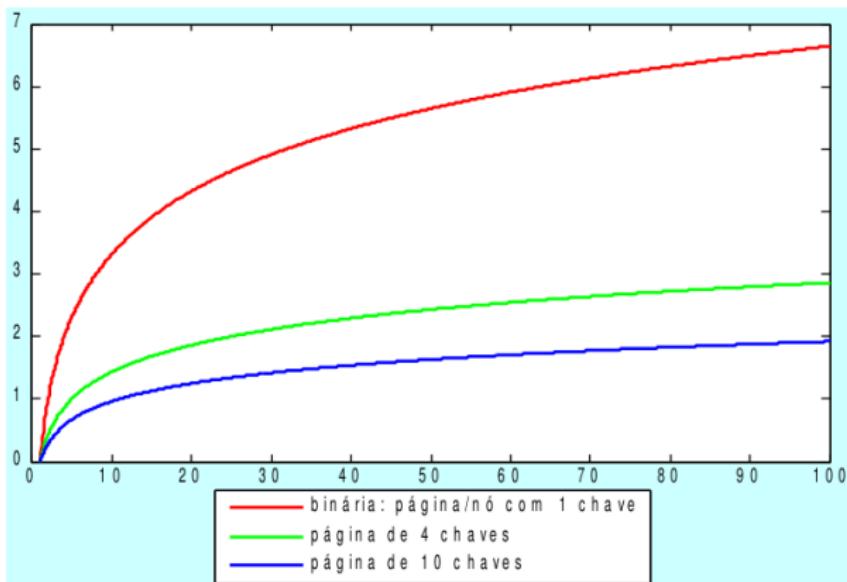


Árvores Multi-vias

- Número de chaves por página: $d = 4$



Árvores Multi-vias

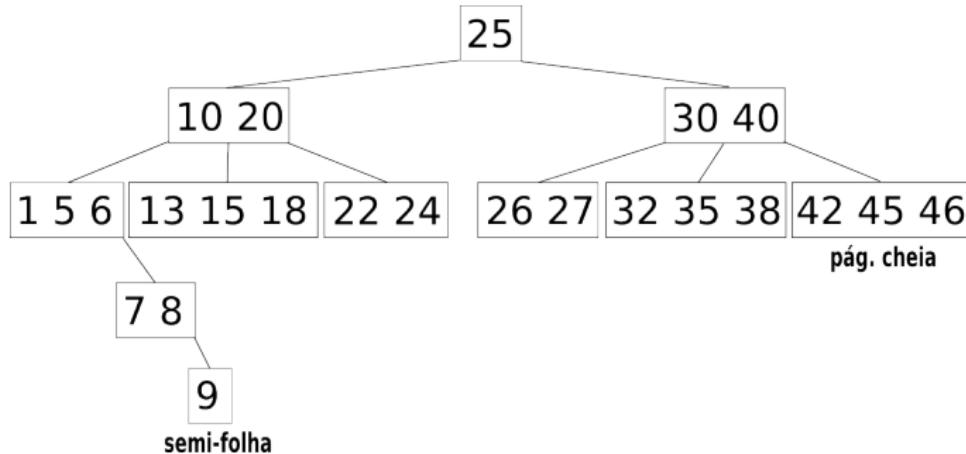


Árvores Multi-vias

- O uso de múltiplas vias aumenta a eficiência computacional para a inserção, consulta e remoção de elementos na estrutura.
- Uma árvore multi-vias contém:
 - cada nó é uma página que pode conter múltiplos registros
 - é denominada **ordem** da árvore, o tamanho da página (p.ex. $K = 4$)
 - cada página contém seus elementos armazenados em ordem crescente
 - cada página contém K elementos e $K + 1$ ponteiros para sub-árvores
 - cada sub-árvore à esquerda da chave C_i contém todos os elementos menores que C_i
 - cada sub-árvore à direita da chave C_i contém todos os elementos maiores que C_i

Árvores B

Exemplo: $K = 3$



Árvore Multi-vias - Operações

Inserção é realizado numa página existente não-cheia ou em uma nova página. Uma página pode ser implementada como uma lista encadeada ou um vetor. Neste caso, uma inserção pode implicar mover dados (a fim de manter a ordenação intra-página).

Uma árvore multi-vias não necessariamente é balanceada.

Solução ⇒ árvores B

Remoção pode levar a destruição de uma página que ficaria vazia. Pode implicar em mover elementos (a fim de manter a ordenação intra-página). Pode implicar na substituição do elemento pelo seu antecessor/sucessor (a fim de manter a ordenação inter-página).

Busca similar ao processo de busca nas árvores binárias de busca, apenas adaptando para o caso de K-vias

Árvores B

- Uma árvore B é uma árvore multi-vias onde cada página (exceto a raiz) contém no mínimo n elementos e no máximo $2n$ elementos ordenados ($K = 2n$)

$$n \leq m \leq 2n$$

- Cada página ou é uma folha ou contém no máximo $m + 1$ filhos
- Todas as páginas folhas ocorrem no mesmo nível

Árvores B

```
1      #define ordem 4
2      typedef int TipoChave;
3      typedef struct _No{
4          char ehFolha;
5          int numChaves;
6          TipoChave chaves[ordem];
7          struct _No *filhos [ordem+1];
8      }No;
9      typedef No *Arvore;
10
```

Árvores B - Inserção

- se a página possuir $n < m < 2n$ elementos, então a inserção ordenada ocorrerá na própria página
 - **Atenção!** a raiz pode receber menos do que n elementos
- Se a página estiver cheia ($m = 2n$), então a inserção provoca um processo de alteração na estrutura da árvore através da abrodagem '*split-and-promote*', ou seja, a divisão da página em duas e a promoção do elemento central para o nível anterior

Árvores B - Inserção

Inserindo 30: 30

Inserindo 40: 30 40

Inserindo 10: 10 30 40

Inserindo 5: 5 10 30 40

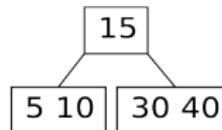
página cheia!

Árvores B - Inserção

Inserindo 15:

5 10 15 30 40

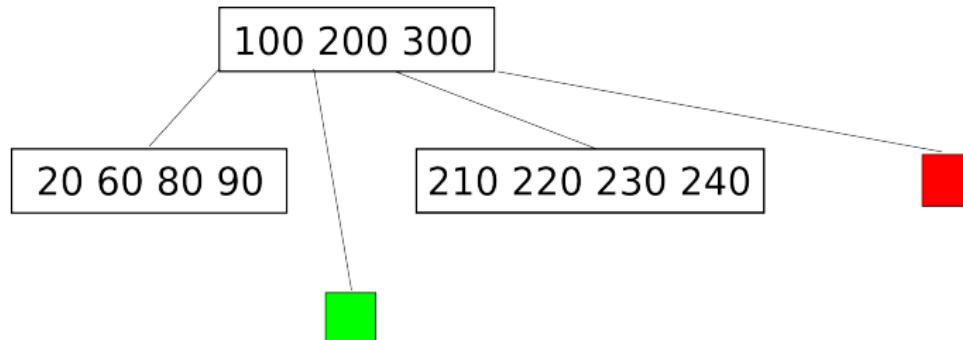
overflow!!



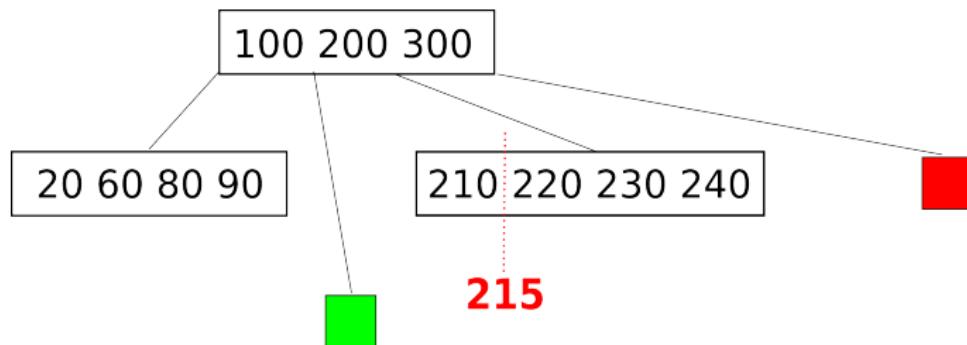
split-and-promote = 15

Árvores B - Inserção - Exercício

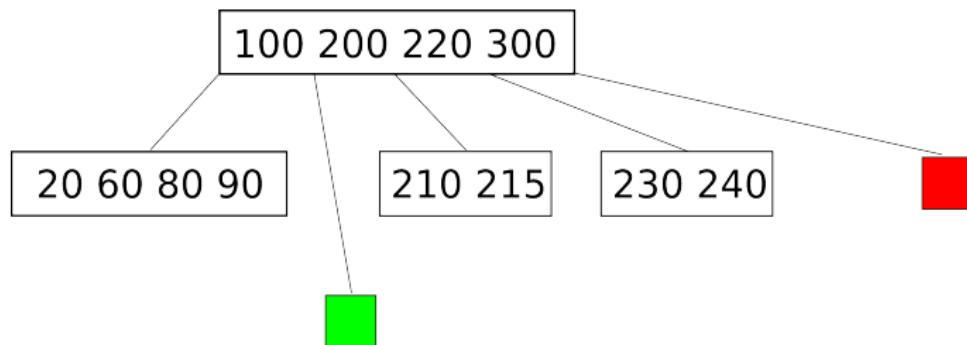
Dada a configuração de uma árvore B como apresentado abaixo, insira o elemento **215**



Árvores B - Inserção - Exercício



Árvores B - Inserção - Exercício

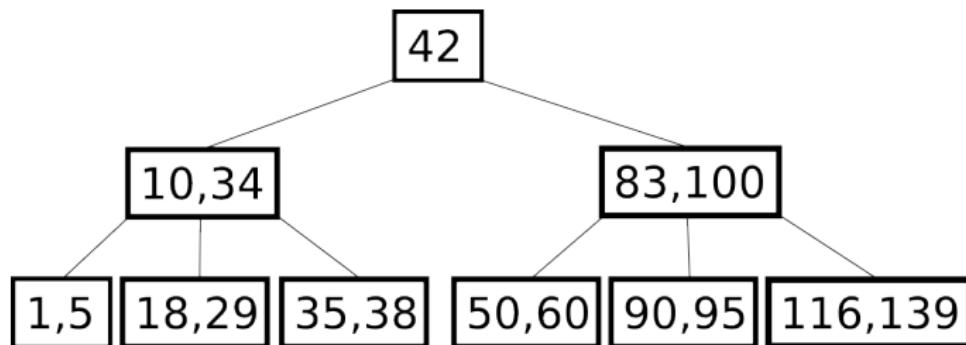


Árvores B - Inserção - Exercício II

- Apresentar a árvore B resultante ($K = 4$) após a inserção da seguinte sequência numérica

100, 10, 83, 18, 90, 1, 5, 29, 95, 42, 139, 35, 116, 34, 38, 50, 60

Árvores B - Inserção - Exercício II



Árvores B - Inserção - Algoritmo (1/5)

```
1      const T = 2,
2          MAX_CHAVES = 2 * T - 1, // qtde max de chaves
3          MAX_FILHOS = 2 * T, // qtde max de filhos
4          MIN_OCUP = T - 1; // ocupacao min em cada no
5
6      typedef struct no_arvoreB arvoreB;
7
8      struct no_arvoreB {
9          int num_chaves; // qtde chaves do no
10         int chaves[ MAX_CHAVES ];
11         arvoreB *filhos [ MAX_FILHOS ];
12     };
13
```

Árvores B - Inserção - Algoritmo (2/5)

```
1 void insere_chave( arvoreB *raiz , int info , arvoreB *filhodir ) {
2     int k, pos;
3
4     // obtem o endereco para insercao da nova chave
5     pos = busca_binaria( raiz , info );
6     k = raiz->num_chaves;
7
8     // libera espaco para insercao da chave na pagina
9     while( k > pos && info < raiz->chaves[ k - 1 ] ){
10         raiz->chaves[ k ] = raiz->chaves[ k - 1 ];
11         raiz->filhos[ k + 1 ] = raiz->filhos[ k ];
12         k--;
13     }
14
15     // insere a chave
16     raiz->chaves[ pos ] = info;
17     raiz->filhos[ k + 1 ] = filhodir ;
18     raiz->num_chaves++;
19 }
20 }
```

Árvores B - Inserção - Algoritmo (3/5)

```
1     arvoreB* insere( arvoreB *raiz, int info , bool *h, int *info_retorno ) {
2         int i, j, pos, info_mediano;
3         arvoreB *temp, *filho_dir ;
4
5         if( ! raiz ) {
6             *h = true; // atingiu no folha
7             *info_retorno = info;
8             return NULL;
9         }
10        else {
11            pos = busca_binaria( raiz, info );
12            if( raiz->num_chaves > pos && raiz->chaves[ pos ] == info ) {
13                printf( "Chave ja contida na arvore" );
14                *h = false;
15            }
16            else {
17                filho_dir = insere( raiz->filhos[ pos ], info , h, info_retorno );
18                if( *h ) {
19                    if( raiz->num_chaves < MAX_CHAVES ) { // tem espaco na pagina
20                        insere_chave( raiz , *info_retorno , filho_dir );
21                        *h = false;
22                    }
23                else { // overflow! SPLIT-AND-PROMOTE
24                    temp = ( arvoreB* ) malloc( sizeof( arvoreB ) );
25                    temp->num_chaves = 0;
26                    for( i = 0; i < MAX_FILHOS; i++ )
27                        temp->filhos[ i ] = NULL;
28
29                    info_mediano = raiz->chaves[ MIN_OCUP ]; // PROMOTE
30                }
31            }
32        }
33    }
```

Árvores B - Inserção - Algoritmo (4/5)

```
1      // insere metade da pagina no temp (SPLIT)
2      temp->filhos[ 0 ] = raiz->filhos[ MIN_OCUP + 1 ];
3      for( i = MIN_OCUP + 1; i < MAX_CHAVES; i++ )
4          insere_chaves( temp, raiz->chaves[ i ], raiz->filhos[ i + 1 ] );
5
6      // atualiza raiz
7      for( i = MIN_OCUP; i < MAX_CHAVES; i++ ) {
8          raiz->chaves[ i ] = 0;
9          raiz->filhos[ i + 1 ] = NULL;
10     }
11     raiz->num_chaves = MIN_OCUP;
12
13     // verifica onde inserir a nova chave
14     if( pos <= MIN_OCUP )
15         insere_chave( raiz, *info_retorno, filho_dir );
16     else
17         insere_chave( temp, *info_retorno, filho_dir );
18
19     *info_retorno = info_mediano;
20     return temp;
21 }
22 }
23 }
24 }
25 }
```

Árvores B - Inserção - Algoritmo (5/5)

```
1     arvoreB * insere_arvoreB( arvoreB *raiz, int info ) {
2         bool h;
3         int info_retorno , i;
4         arvoreB * filho_dir , *nova_raiz;
5
6         filho_dir = insere( raiz , info , &h, &info_retorno );
7         if( h ) { // aumenta altura da arvore ?
8             nova_raiz = ( arvoreB* ) malloc( sizeof( arvoreB ) );
9             nova_raiz ->num_chaves = 1;
10            nova_raiz ->chaves[ 0 ] = info_retorno ;
11            nova_raiz ->filhos[ 0 ] = raiz;
12            nova_raiz ->filhos[ 1 ] = filho_dir ;
13
14            for( i = 2; i <= MAX_CHAVES; i++ )
15                nova_raiz ->filhos[ i ] = NULL;
16            return nova_raiz ;
17        }
18        else return raiz ;
19    }
20 }
```

Árvores B - Busca - Algoritmo (1/3)

```
1     int busca_binaria( arvoreB *no, int info ) {
2         int meio, i = 0, f = no->num_chaves - 1;
3
4         while( i <= f ) {
5             meio = ( i + f ) / 2;
6             if( no->chaves[ meio ] == info )
7                 return meio; // encontrou a posicao da chave procurada
8             else if( no->chaves[ meio ] > info )
9                 f = meio - 1;
10            else
11                i = meio + 1;
12        }
13        return i; // nao encontrou
14    }
15 }
```

Árvores B - Busca - Algoritmo (2/3)

```
1     arvoreB * busca( arvoreB *raiz , int info ) {
2         arvoreB * no;
3         int pos;
4
5         no = raiz;
6         while( no ) {
7             pos = busca_binaria( no, info );
8             if( pos < no->num_chaves && no->chaves[ pos ] == info )
9                 return no;
10            else
11                no = no->filhos[ pos ];
12        }
13        return NULL;
14    }
15 }
```

Árvores B - Busca - Algoritmo (3/3)

```
1     void em_ordem( arvoreB *raiz ) {
2         int i;
3
4         if( raiz ) {
5             for( i = 0; i < raiz->num_chaves; i++ ) {
6                 em_ordem( raiz->filhos[ i ] );
7                 printf( "\n%d", raiz->chaves[ i ] );
8             }
9             em_ordem( raiz->filhos[ i ] );
10        }
11    }
12 }
```

Árvores B - Remoção

1. Caso o elemento X a ser removido esteja em uma página folha:
 - 1.1 se a página tiver $m > n$, remover o elemento X da página
 - 1.2 caso contrário:
 - 1.2.1 se ambas as páginas irmãs contiverem $m = n$: concatenação recursiva entre as páginas irmãs de X após X ser removido (**join**)
 - 1.2.2 se a página irmãs à esquerda de X tiver $m > n$: o pai de X desce para a página de X e o antecessor do pai de X é promovido para nova raiz
 - 1.2.3 se a página irmãs à direita de X tiver $m > n$: o pai de X desce para a página de X e o sucessor do pai de X é promovido para nova raiz
2. Caso o elemento esteja em uma página interna, o elemento deve ser substituído pelo seu sucessor (ou antecessor) na árvore (percurso *em ordem*) = processo denominado **doação**

No caso de doação:

 - 2.1 se a página doadora tiver $m > n$, basta doar um elemento
 - 2.2 se a página doadora tiver $m = n$:
 - 2.2.1 se alguma das páginas irmãs tiver $m > n$: redistribuição de elementos entre páginas irmãs (**rotação da sub-árvore**)
 - 2.2.2 caso contrário: concatenação recursiva entre páginas irmãs (**join**)

Avaliação #3 - Árvores B

Título: Trabalho Prático #3 - Indexação via Árvores B

Objetivo: Construir um sistema gerador de tabelas auxiliares de indexação de dados para o banco de dados produzido no trabalho 1 através de percurso *In-order* em uma árvore B.

Forma de Entrega: Deve ser entregue todos os códigos-fonte (devidamente documentados/comentados), bem como deve ser preparada uma apresentação que inclui uma demonstração de funcionamento do sistema.

Deve ainda ser entregue uma cópia do arquivo construído previamente pelo sistema, que contenha a tabela indexada (por campo à escolha do usuário) e a listagem resultante.

Prazo de Entrega: 31/05/2016

Avaliação #3 - Árvores B

- Critérios:
1. Equipes de 3 alunos
 2. Deve ser utilizada a base de dados de 1 Gb construída no trabalho anterior como base para o processo de indexação.
 3. Árvore de Ordem: 100.000
 4. O programa a ser desenvolvido deve permitir ao usuário:
 - 4.1 Escolher o campo de indexação, a partir de uma lista de opções (que podem incluir chaves compostas se necessário)
 - 4.2 Uma tabela de índices = lista de endereços (em bytes) que referenciam os registros ordenadamente, deve ser gerada para a escolha definida pelo usuário. Um endereço de um registro é o número de bytes que são necessários serem deslocados para se acessar aquela informação;
 - 4.3 Permitir a inclusão, consulta e exclusão de registros mesmo após a construção do arquivo indexado (neste caso = reindeixar)
 - 4.4 Visualizar uma listagem (paginada) dos dados indexados (percurso in-ordem) resultantes em um arquivo texto de saída;
 - 4.5 Exibir ao final do processo o tempo gasto para o processamento.

Indexação por Tabelas de Espalhamento

Tabelas de Espalhamento

Motivação:

- Estruturas de dados lineares apresentam complexidade de inserção e busca também linear: **$O(n)$**
- Estruturas de dados hierárquicas apresentam complexidade logarítmica: **$O(\log n)$**
- Uma estrutura de dados (para inserção e buscas) ideal deveria apresentar complexidade constante: **$O(1)$**

Tabelas de Espalhamento

- As **tabelas de espalhamento** (ou *Hash Tables*) são as estruturas de dados que, potencialmente, mais se aproximam de $O(1)$

- O que são tabelas de espalhamento ?
 - são estruturas de dados que implementam o conceito **chave = valor** (dicionário de dados) através de endereçamento direta de dados

- Podem ser implementadas tanto como estruturas estáticas (vetor) quanto dinâmicas (listas)

Tabelas de Espalhamento

Ideia central de *Hashing*:

- Inserir uma nova chave na estrutura em um endereço determinado por uma **função de espalhamento**

$$\text{address} = \text{Hashing}(\text{key})$$



Tabelas de Espalhamento

Assumindo que

$$\text{Hashing}(\text{Key}) = \text{Key} \bmod 11$$

Inserindo chaves em uma tabela *Hash*:

$$5 = \text{Hashing}(16)$$



Função de Espalhamento

- Uma **função de espalhamento** especifica o critério de inserção de chaves na tabela
- É responsável por mapear valores de chaves em endereços na estrutura da tabela

Nota: Idealmente, cada chave deve produzir um endereço único na tabela, porém isso é bastante complexo de ser desenvolvido = **problema dos sinônimos**

- Chaves distintas para as quais a função de espalhamento gera o mesmo endereço, são chamadas *sinônimos*
- Sempre que chaves são endereçadas na mesma posição de memória, diz-se que ocorreu uma **colisão** de chaves
 - Critérios para tratamento de colisões representam o aspecto crítico para cálculo da eficiência das tabelas de espalhamento

Função de Espalhamento

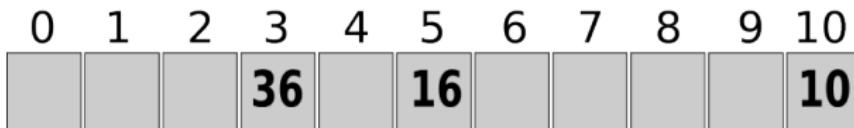
- Exemplos de funções de espalhamento (dado N o tamanho da tabela):

$$address = Hashing(key)\{return \text{len}(key)\}$$
$$address = Hashing(key)\{return \text{key mod } N\}$$
$$address = Hashing(key)\{return(key[0] \times 26^0 + key[1] \times 26^1 + key[2] \times 26^2 + \dots) \text{mod } 51\}$$

Colisões

- Colisões ocorrem quando a função de espalhamento gera endereços que já estão ocupados por outras chaves previamente inseridas

5 = Hashing(159)



Colisões

- Tratamento de colisões podem ser realizados por diversas abordagens distintas:
 1. por **re-espalhamento** de chaves: linear, quadrático, duplo
 2. por **encadeamento** de chaves

Tratando colisões por Re-espalhamento

- Re-espalhar significa utilizar um critério secundário de espalhamento, especificamente para tratar os casos de colisão
- No caso de **re-espalhamento linear** significa encontrar o primeiro espaço livre após o endereço selecionado pela função de espalhamento (e que causou a colisão)

$$\text{Hashing}(\text{key}) + i, i = 1, 2, 3, \dots$$

- O processo é cíclico: ao ser atingido o fim do vetor, volta-se ao início
- Inserção **impossível** caso o deslocamento retorne à posição de colisão

5 = Hashing(159)



Tratando colisões por Re-espalhamento

- Como recuperar um registro específico se os estamos inserindo em endereços diferentes daqueles retornados pela função de espalhamento ?
- Como recuperar a chave **159** ?

Tratando colisões por Re-espalhamento

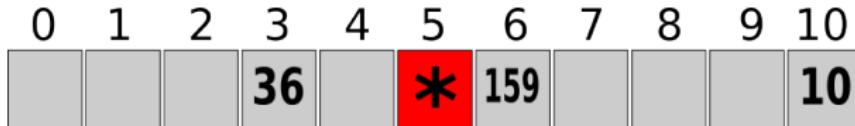
- Como recuperar um registro específico se os estamos inserindo em endereços diferentes daqueles retornados pela função de espalhamento ?
- Como recuperar a chave **159** ?
- Dado que $add = Hashing(159)$ e $add = 5$:

$while(Table[add] \neq 159)$

$\{ add = (add + 1) \bmod N \}$

Tratando colisões por Re-espalhamento

- Neste caso, a exclusão de chaves da tabela deve ser obrigatoriamente realizada através da abordagem de **exclusão lógica** (flag)
- exemplo: excluir a chave **16**



Tratando colisões por Re-espalhamento

- Alternativamente, pode-se utilizar um **re-espalhamento quadrático**

$$(Hashing(key) + i^2) \bmod N, \text{ para } i = 1, 2, 3, \dots$$

- A hipótese por trás dessa variante é que, se espalharmos mais as chaves, diminuem as chances de colisão = maior eficiência

Tratando colisões por Re-espalhamento

- Um terceira alternativa é utilizar um **re-espalhamento duplo**, através de uma segunda função de espalhamento (baseada em outro critério)

$$(Hashing(key) + i \times Hashing2(key)) \bmod N, \text{ para } i = 1, 2, 3, \dots$$

Tabelas *Hashing* - Exercício

- Crie um programa que implemente uma tabela Hash de tamanho N
- Crie uma rotina que faça a inserção de chaves numéricas (geradas aleatoriamente)
- Exiba o conteúdo da tabela ao final do processo, bem como a quantidade de colisões que ocorreram¹
- Permita a consulta a uma chave específica:
 1. O programa deve retornar a posição da chave procurada ou
 2. uma mensagem de *chave não encontrada* caso contrário
 - O programa deve ainda informar quantas consultas foram necessárias para fornecer a resposta da consulta

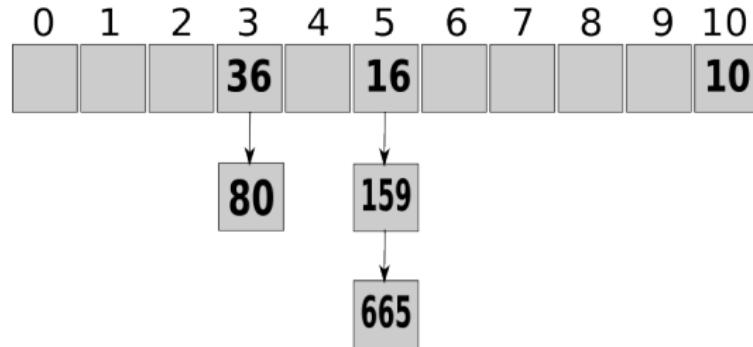
¹ Teste diferentes abordagens para tratamento de colisões
212 of 218

Tabelas *Hashing* Encadeadas

- Alternativamente, outra forma de tratamento de colisões é através da abordagem por **encadeamento**
- Nesta abordagem, cada posição da tabela, ao invés de armazenar uma chave, armazena um ponteiro para o início de uma lista encadeada que contém todas as chaves sinônimo que colidiram naquela posição específica
- A busca por uma chave específica então consiste em se determinar o endereço da chave (através da função de espalhamento) e, em seguida, uma busca linear na lista encadeada
- Para que se tenha um maior desempenho, manter a lista encadeada ordenada pode acelerar o processo de busca

Tabelas *Hashing* Encadeadas

5 = Hashing(665)



Tabelas *Hashing* Encadeadas

- Dependente da natureza dos dados sendo indexados, o número de colisões produzidas pode ser bastante elevado
- Quando isso ocorre, a abordagem de tratamento de colisão por listas encadeadas é ineficiente dado que o problema da inserção/busca começa gradativamente a se afastar do **$O(1)$** , podendo até (em casos extremos) atingir **$O(n)$**
- Um solução para melhoria desse problema é utilizar **encadeamento hierárquico** (árvores) ao invés de linear

Tabelas *Hashing* - Exercício (II)

- Crie um programa que implemente uma tabela Hash de tamanho N com tratamento de colisões por listas encadeadas
- Crie uma rotina que faça a inserção de chaves numéricas (geradas aleatoriamente)
- Exiba o conteúdo da tabela ao final do processo
- Permita a consulta a uma chave específica:
 1. O programa deve retornar a posição da chave procurada ou
 2. uma mensagem de *chave não encontrada* caso contrário
- O programa deve ainda informar quantas consultas foram necessárias para fornecer a resposta da consulta

Avaliação #4 - Hashing

Título: Trabalho Prático #4 - Tabelas de Espalhamento

Objetivo: Construir um sistema gerador de tabelas auxiliares de indexação de dados para o banco de dados produzido no trabalho 1 através de uma tabela de espalhamento.

Forma de Entrega: Deve ser entregue todos os códigos-fonte (devidamente documentados/comentados), bem como deve ser preparada uma apresentação que inclui uma demonstração de funcionamento do sistema.

Deve ainda ser entregue uma cópia do arquivo construído previamente pelo sistema, que contenha a tabela indexada (por campo à escolha do usuário) e a listagem resultante.

Prazo de Entrega: 26/06/2014

Avaliação #4 - Hashing

- Critérios:
1. Equipes de 3 alunos
 2. Deve ser utilizada a base de dados de 1 Gb construída no trabalho anterior como base para o processo de indexação.
 3. O programa a ser desenvolvido deve permitir ao usuário:
 - 3.1 A equipe deve propor o mecanismo de indexação para tabela (método da função de espalhamento)
 - 3.2 Escolher o campo de indexação, a partir de uma lista de opções (que podem incluir chaves compostas se necessário)
 - 3.3 Uma tabela de índices = lista de endereços (em bytes) que referenciam os registros ordenadamente, deve ser gerada para a escolha definida pelo usuário. Um endereço de um registro é o número de bytes que são necessários serem deslocados para se acessar aquela informação;
 - 3.4 Permitir a exclusão de registros mesmo após a construção do arquivo indexado (neste caso = reindeixar)
 - 3.5 Visualizar uma listagem (paginada) dos dados indexados resultantes em um arquivo texto de saída;
 - 3.6 Exibir ao final do processo o tempo gasto para o processamento.