

Ordenação em arquivos (externa)



Prof. André Backes | @progdescomplicada

Conceitos básicos

- Ordenação
 - Ato de colocar um conjunto de dados em uma determinada ordem predefinida
 - Fora de ordem
 - 5, 2, 1, 3, 4
 - Ordenado
 - 1, 2, 3, 4, 5
 - 5, 4, 3, 2, 1
- Algoritmo de ordenação
 - Coloca um conjunto de elementos em uma certa ordem

Conceitos básicos

- A ordenação permite que o acesso aos dados seja feito de forma mais eficiente
- É parte de muitos métodos computacionais
 - Algoritmos de busca, intercalação/fusão, utilizam ordenação como parte do processo
 - Aplicações em geometria computacional, bancos de dados, entre outras necessitam de listas ordenadas para funcionar

Conceitos básicos

- A ordenação é baseada em uma chave
 - A chave de ordenação é o **campo** do item utilizado para comparação
 - Valor armazenado em um array de inteiros
 - Campo nome de uma struct
 - etc
 - É por meio dela que sabemos se um determinado elemento está a frente ou não de outros no conjunto

Conceitos básicos

- Alguns tipos de ordenação
 - numérica
 - 1, 2, 3, 4, 5
 - lexicográfica (ordem alfabética)
 - Ana, André, Carlos, Eduardo

Conceitos básicos

- Independente do tipo, a ordenação pode ser
 - Crescente
 - 1, 2, 3, 4, 5
 - Ana, André, Carlos, Eduardo
 - Decrescente
 - 5, 4, 3, 2, 1
 - Eduardo, Carlos, André, Ana

Conceitos básicos

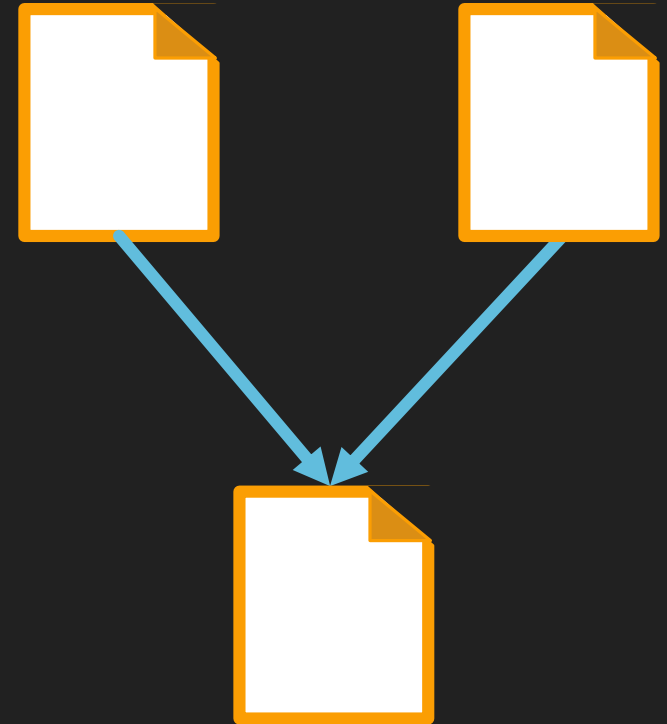
- Ordenação interna
 - O conjunto de dados a ser ordenado cabe todo na memória principal (RAM)
 - Qualquer elemento pode ser imediatamente acessado
- Ordenação externa
 - O conjunto de dados a ser ordenado não cabe na memória principal
 - Os dados estão armazenados em memória secundário (por exemplo, um arquivo)
 - Os elementos são acessados sequencialmente ou em grandes blocos

Conceitos básicos

- Além disso, a ordenação pode ser estável ou não
 - Um algoritmo de ordenação é considerado **estável** se a ordem dos elementos com chaves iguais não muda durante a ordenação
 - O algoritmo preserva a **ordem relativa** original dos valores

Processamento co-sequencial

- É o processamento coordenado de dois ou mais arquivos em disco (e.g. tabelas de índices) para produzir um único arquivo em disco
 - Esse tipo de operação é muito comum em arquivos
 - Leitura e escrita em arquivos devem ser sincronizadas
- As operações básicas são
 - *Matching*
 - *Merging*
 - *Sorting*



Processamento co-sequencial

- *matching* - intersecção de duas listas
 - Dadas duas listas de nomes de pessoas, queremos produzir uma lista com os nomes comuns a ambas, assumindo que cada uma das listas originais não contém nomes repetidos e que estão ordenadas em ordem crescente

Lista1	Lista2	Saída
Adriana	Adriana	Adriana
Carlos	Anderson	Carlos
Cid	André	Davi
Davi	Beatriz	Fábio
Fábio	Bruno	Gabriel
Gabriel	Carlos	
Tânia	Davi	
	Deise	
	Fábio	
	Gabriel	
	Gisele	
	Thaíse	
	Walter	

Processamento co-sequencial - matching

- A ideia básica do algoritmo
 - Lê um nome de cada lista e os compara
 - Se ...
 - Ambos são iguais, copiar o nome para a saída e avançar para o próximo nome em cada arquivo.
 - O nome da Lista1 é menor, avança na Lista1 (lendo o próximo nome).
 - O nome da Lista1 é maior, avança na Lista2

```
n1 = read_name(list1);
n2 = read_name(list2);
while (!feof(list1) && !feof(list2)){
    if (n1 < n2)
        n1 = read_name(list1);
    else
        if (n1 > n2)
            n2 = read_name(list2);
        else{ // n1 == n2
            write_name(n1, list3);
            n1 = read_name(list1);
            n2 = read_name(list2);
        }
}
```

Processamento co-sequencial

- *merging* - união de duas listas
 - Dadas duas listas de nomes de pessoas, queremos produzir uma lista com todos os nomes de ambas as listas, sem repetições, assumindo que cada uma das listas originais não contém nomes repetidos e que estão ordenadas em ordem crescente

Lista1	Lista2	Saída
Adriana	Adriana	Adriana
Carlos	Anderson	Anderson
Cid	André	André
Davi	Beatriz	Beatriz
Fábio	Bruno	Bruno
Gabriel	Carlos	Carlos
Tânia	Davi	Cid
	Deise	Davi
	Fábio	Deise
	Gabriel	Fábio
	Gisele	Gabriel
	Thaíse	Gisele
	Walter	Tania
		Thaíse
		Walter

Processamento co-sequencial - merging

- A ideia básica do algoritmo
 - Lê um nome de cada lista e os compara
 - Neste caso, dois nomes, um de cada lista são comparados, e um nome é gerado na saída a CADA passo do comando condicional.

```
n1 = read_name(list1);
n2 = read_name(list2);
while(!feof(list1) || !feof(list2)){
    if(n1 < n2){
        write_name(n1,list3);
        n1 = read_name(list1);
    }else
        if (n1 > n2){
            write_name(n2,list3);
            n2 = read_name(list2);
        }else{ // n1 == n2
            write_name(n1,list3);
            n1 = read_name(list1);
            n2 = read_name(list2);
        }
}
```

Processamento co-sequencial

- *Sorting* - ordenação
 - É a aplicação mais comum de processamento co-sequencial
 - Concatenar 2 listas para criar uma lista ordenada como saída
- Não há motivo para restringir o número de entradas na intercalação a 2
 - Podemos generalizar o processo para intercalar k corridas simultaneamente.
 - *k-way mergesort*

Ordenação Externa

○ Alternativas

- 1) Usar ordenação interna no arquivo
- 2) Carregar o arquivo na memória principal e usar ordenação interna
- 3) Ordenação por chave (ordenação com o apoio de um índice)
- 4) Ordenação por intercalação

Ordenação Externa

- Alternativa 1 - Usar ordenação interna no arquivo
 - Existem vários métodos disponíveis



Usar ordenação interna no arquivo

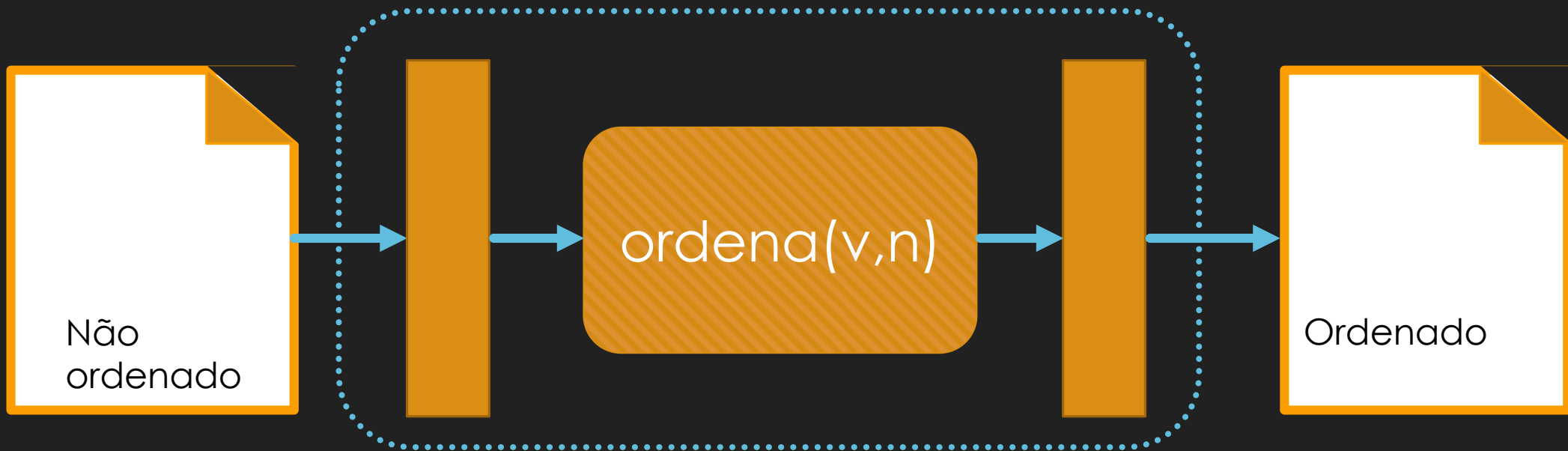
Método	Complexidade
BubbleSort	$O(N^2)$, $O(N)$ no melhor caso
InsertionSort	$O(N^2)$, $O(N)$ no melhor caso
SelectionSort	$O(N^2)$
MergeSort	$O(N \log N)$
QuickSort	$O(N \log N)$, $O(N^2)$ no pior caso
HeapSort	$O(N \log N)$
CountingSort	$O(N+k)$
BucketSort	$O(N+k)$, $O(N^2)$ no pior caso
ShellSort	$O(N \log N)$, $O(N (\log N)^2)$ no pior caso

Usar ordenação interna no arquivo

- Solução inadequada
 - Ordenação interna tenta minimizar número de comparações
 - Não tenta minimizar operações de leitura/escrita no dispositivo de memória secundária
 - Não tenta minimizar operações do tipo *seek*, que são as mais lentas
 - Aplicação muito difícil complexa em dispositivos de acesso sequencial (fitas)

Ordenação Externa

- Alternativa 2 – Carregar o arquivo na memória principal e usar ordenação interna



Carregar na memória principal e usar ordenação interna

- Útil apenas quando o arquivo cabe INTEIRO em memória principal
 - Carrega arquivo
 - Ordena em memória
 - Grava no disco
- Custo: 1 operação de leitura sequencial + 1 operação de gravação sequencial + custo da ordenação

Ordenação por chave - *Keysorting*

- Alternativa 3: o arquivo não cabe inteiro na memória mas podemos criar um índice que cabe inteiro em memória principal
 - A chave de ordenação é o **campo** do item utilizado para comparação
 - Valor armazenado em um *array* de inteiros
 - Campo nome de uma *struct*
 - Etc
 - Abordagem simples

Ordenação por chave - *Keysorting*

- Baseado na ideia de que para ordenar um arquivo em memória os únicos dados que precisam estar na memória são as chaves
 - Não é preciso ler todos os atributos
- Ordenam-se as chaves e então reorganizam-se os registros no arquivo de acordo com a nova ordem
 - Como não lê os registros inteiros para memória, é capaz de ordenar arquivos maiores que a ordenação em memória principal

Ordenação por chave - Keysorting

- Funcionamento
 - Leitura completa do arquivo de dados
 - Para cada registro do vetor em RAM obtém o RRN (*Relative Record Number*)
 - Identifica o *byte offset* do registro em disco ($\text{byte offset} = \text{RRN} * \text{tamRegistro}$)

```
M A R I A | R U A b 1 | S ...
J O A O | R U A b A | R I ...
P E D R O | R U A b X V | ...
A N T O N I A | R U A b X ...
A N A | R U A b A U G U S ...
```

arquivo desordenado em disco

<i>chave</i>	<i>RRN</i>
M A R I A	0
J O A O	1
P E D R O	2
A N T O N I A	3
A N A	4

vetor em RAM

Ordenação por chave - Keysorting

- Funcionamento
 - Ordenação do vetor de chaves em RAM usando um método de ordenação tradicional
 - Grava um novo arquivo com os registros na ordem correta

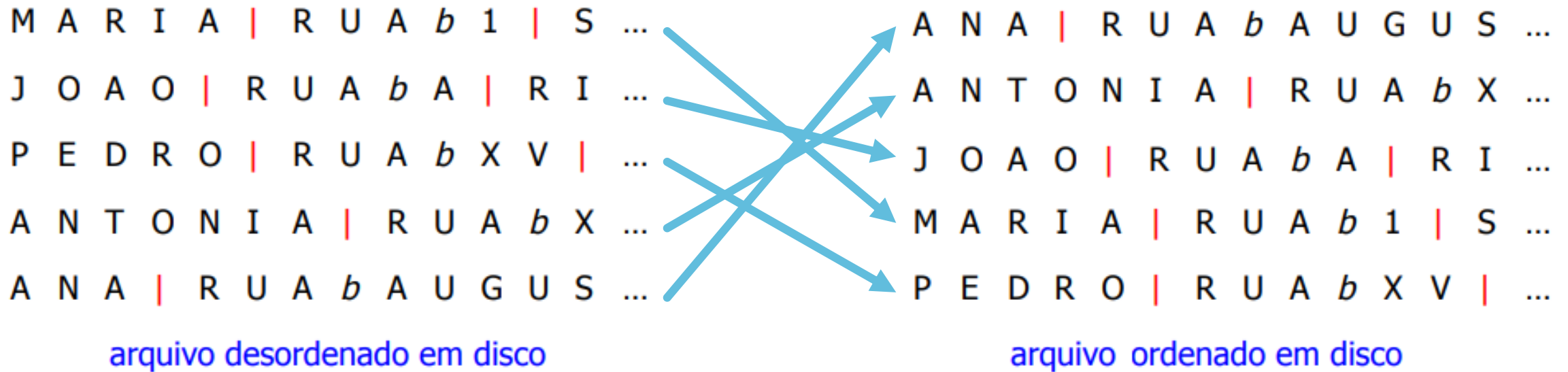
<i>chave</i>	<i>RRN</i>
M A R I A	0
J O A O	1
P E D R O	2
A N T O N I A	3
A N A	4

vetor desordenado em RAM

<i>chave</i>	<i>RRN</i>
A N A	4
A N T O N I A	3
J O A O	1
M A R I A	0
P E D R O	2

vetor ordenado em RAM

Ordenação por chave - Keysorting



Ordenação por chave - Keysorting

- *Keysort* permite ordenar um arquivo grande caso suas chaves caibam em memória
 - Entretanto apresenta alto custo em número de acessos (*seeks*) para escrever o arquivo ordenado
 - Uma operação de *seek* para cada registro
 - Operação muito cara

Ordenação por chave - *Keysorting*

- Por que realizar a tarefa custosa de escrever em disco a versão ordenada do arquivo?
- Solução melhor é trabalhar com índices
 - Grava-se a ordenação da chave em um novo arquivo (arquivo de índice)
 - Realiza-se busca binária no arquivo de índice, e recupera-se o RRN ou *byte offset*
 - Realiza-se acesso direto no arquivo original (arquivo de dados)

Ordenação por chave - Keysorting

- Infelizmente, o *Keysort* não permite a ordenação de arquivos realmente grandes
 - Arquivo com 8.000.000 de registros de 100 bytes cada
 - Chave com 10 bytes
 - Arquivo todo: 800 MB
 - Se tivéssemos disponível apenas 10 MB para o programa rodar, as chaves (80 MB) não caberiam em memória
- Dúvidas
 - E se a busca for feita por outro campo que não seja o campo ordenado?
 - O que acontece quando um novo registro é inserido?

Ordenação por intercalação

- Alternativa 4: o arquivo não cabe inteiro na memória e um índice também não cabe inteiro em memória principal
 - Criar partições ordenadas
 - Intercalar sucessivamente as partições até termos um único arquivo
 - Arquivo final está ordenado
- *K-way merging*

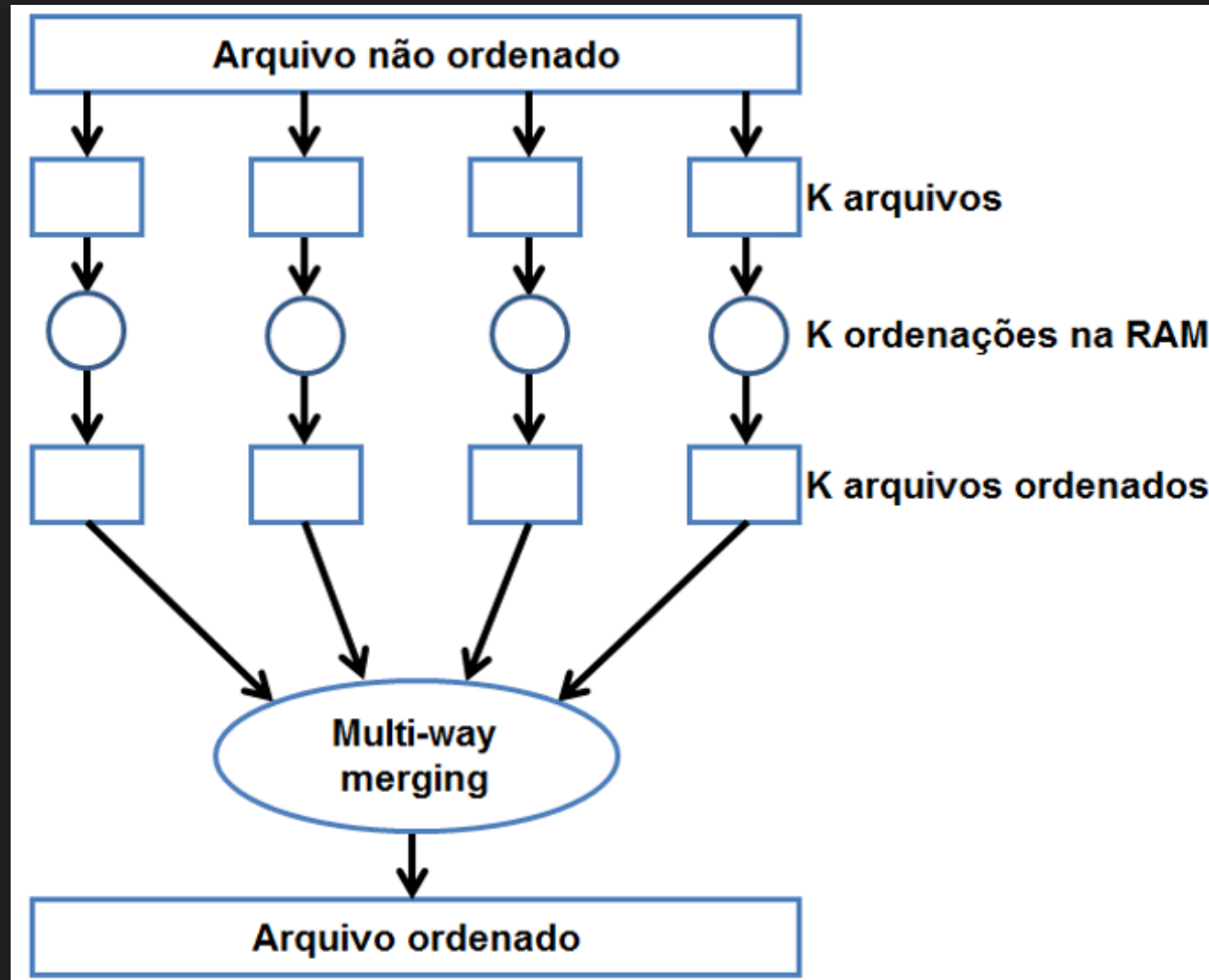
Ordenação por intercalação

- *K-way merging*
 - Pode ordenar arquivos realmente grandes;
 - Envolve apenas acesso sequencial aos arquivos;
 - A leitura e a escrita final também só envolve acesso sequencial;
 - Aplicável também a arquivos mantidos em fita, já que E/S é sequencial.

K-way merging

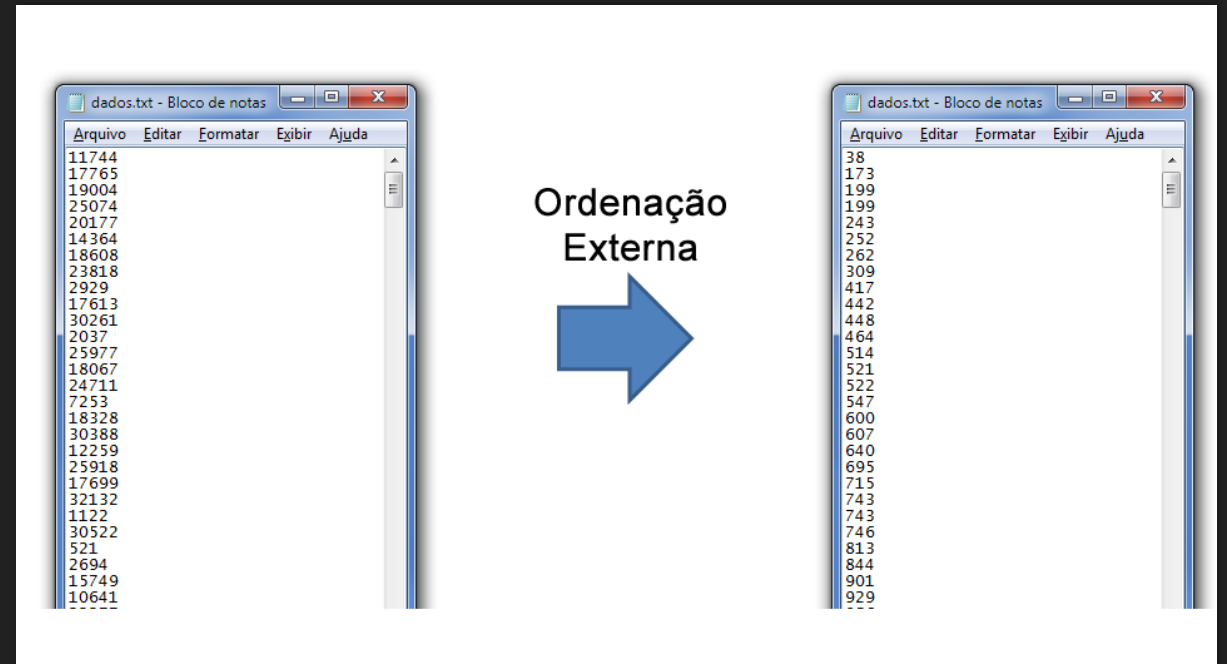
- 1) Carregar parte do arquivo na RAM (N registros de dados);
- 2) Ordenar os dados na RAM com um algoritmo tradicional (ex: *quick sort*);
- 3) Salvar os dados ordenados em um arquivo separado;
- 4) Repetir os passos de 1 a 3 até terminar o arquivo original. Ao final, teremos K arquivos ordenados;
- 5) *k-way merging*: intercalar K blocos ordenados
 - a) Criar $K+1$ buffers de tamanho $N/(K+1)$: um buffer de saída e K buffers de entrada;
 - b) Carregar parte dos arquivos ordenados nos buffers de entrada e intercalar no buffer de saída;
 - c) Se um buffer de entrada ficar vazio: carregar mais dados do respectivo arquivo;
 - d) Se o buffer de saída ficar cheio: salvar os dados no arquivo final

K-way merging



K-way merging

- Exemplo
- Vamos ordenar um arquivo contendo apenas valores inteiros



K-way merging

- Ideia básica
- Criar um arquivo com valores inteiros gerados de forma aleatório e, em seguida, ordenar o arquivo

```
void criArquivoTeste(char *nome) {  
    int i;  
    FILE *f = fopen(nome, "w");  
    srand(time(NULL));  
    for(i=1; i < 1000; i++)  
        fprintf(f, "%d\n", rand());  
    fprintf(f, "%d", rand());  
    fclose(f);  
}  
  
int main() {  
    criArquivoTeste("dados.txt");  
    mergeSortExterno("dados.txt");  
    return 0;  
}
```

K-way merging

- Funcionamento
 - A função recebe o arquivo original
 - Cria $K+1$ buffers de tamanho T
 - K de entrada (arquivos ordenados)
 - 1 de saída
 - Apaga o arquivo original
 - Faz o *merge* dos arquivos ordenados
 - Apaga os arquivos temporários

```
void mergeSortExterno(char *nome) {  
    char novo[20];  
    int K = criaArquivosOrdenados(nome);  
    int i, T = N / (K + 1);  
    remove(nome);  
    merge(nome, K, T);  
    for(i=0; i<K; i++){  
        sprintf(novo, "Temp%d.txt", i+1);  
        remove(novo);  
    }  
}
```

K-way merging

- Criando os arquivos ordenados
 - Leia N posições do arquivo para a memória
 - Chame um algoritmo de ordenação
 - Exemplo, `qsort()`
 - Salve os dados ordenados em um arquivo temporário
- Repita o processo

```
int criaArquivosOrdenados(char *nome) {  
    int V[N], cont = 0, total = 0;  
    char novo[20];  
    FILE *f = fopen(nome, "r");  
    while(!feof(f)) {  
        fscanf(f, "%d", &V[total]);  
        total++;  
        if(total == N) {  
            cont++;  
            sprintf(novo, "Temp%d.txt", cont);  
            qsort(V, total, sizeof(int), compara);  
            salvaArquivo(novo, V, total, 0);  
            total = 0;  
        }  
    }  
    if(total > 0) {  
        cont++;  
        sprintf(novo, "Temp%d.txt", cont);  
        qsort(V, total, sizeof(int), compara);  
        salvaArquivo(novo, V, total, 0);  
    }  
    fclose(f);  
    return cont;  
}
```

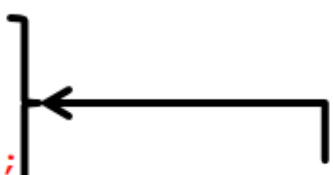
Buffer cheio:
salva em disco

Sobraram dados no
buffer: salva em disco

K-way merging

- Salvando os dados em um arquivo temporário

```
void salvaArquivo(char *nome, int *V, int tam, int mudaLinhaFinal){  
    int i;  
    FILE *f = fopen(nome, "a");  
    for(i=0; i < tam-1; i++)  
        fprintf(f, "%d\n", V[i]);  
  
    if(mudaLinhaFinal == 0)  
        fprintf(f, "%d", V[tam-1]);  
    else  
        fprintf(f, "%d\n", V[tam-1]);  
  
    fclose(f);  
}
```



Controla a mudança de linha no final do arquivo

K-way merging

- Etapa de *merge*
 - Cria um *buffer* para cada arquivo ordenado
 - Carrega dados do arquivo
 - Percorre os *buffers* comparando os valores na posição atual
 - Buffer ficou vazio? Carregar mais do arquivo

```
struct arquivo{  
    FILE *f;  
    int pos, MAX, *buffer;  
};  
  
void merge(char *nome, int numArqs, int K){  
    char novo[20];  
    int i;  
    int *buffer = (int*)malloc(K*sizeof(int));  
  
    struct arquivo* arq;  
    arq=(struct arquivo*)malloc(numArqs*  
                                sizeof(struct arquivo));  
  
    for(i=0; i<numArqs; i++){  
        sprintf(novo, "Temp%d.txt", i+1);  
        arq[i].f = fopen(novo, "r");  
        arq[i].MAX = 0;  
        arq[i].pos = 0;  
        arq[i].buffer = (int*)malloc(K*sizeof(int));  
        preencheBuffer(&arq[i], K);  
    }  
}
```

← struct para gerenciar os buffers

K-way merging

Existe menor elemento?

Coloca no buffer de saída. Salvar se buffer cheio

Sobraram dados no buffer? Salvar em arquivo

```
void merge(char *nome, int numArqs, int K){
    //continuação...

    //enquanto houver arquivos para processar
    int menor, qtdBuffer = 0;
    while(procuraMenor(arq, numArqs, K, &menor) == 1){
        buffer[qtdBuffer] = menor;
        qtdBuffer++;
        if(qtdBuffer == K){
            salvaArquivo(nome, buffer, K, 1);
            qtdBuffer = 0;
        }
    }

    //salva dados ainda no buffer
    if(qtdBuffer != 0)
        salvaArquivo(nome, buffer, qtdBuffer, 1);

    for(i=0; i<numArqs; i++)
        free(arq[i].buffer);
    free(arq);
    free(buffer);
}
```

K-way merging

- Compara todos os *buffers* de entrada e seleciona o menor elemento

Procura menor valor na primeira posição de cada buffer

```
int procuraMenor(struct arquivo* arq, int numArqs,
                 int K, int* menor) {
    int i, idx = -1;
    for(i=0; i<numArqs; i++){
        if(arq[i].pos < arq[i].MAX){
            if(idx == -1)
                idx = i;
            else{
                if(arq[i].buffer[arq[i].pos] <
                   arq[idx].buffer[arq[idx].pos])
                    idx = i;
            }
        }
    }
    if(idx != -1){
        *menor = arq[idx].buffer[arq[idx].pos];
        arq[idx].pos++;
        if(arq[idx].pos == arq[idx].MAX)
            preencheBuffer(&arq[idx], K);
        return 1;
    }else
        return 0;
}
```

Achou menor. Atualiza posição do buffer. Encher se estiver vazio.

K-way merging

- Buffer de entrada ficou vazio?
 - Carregar mais dados
 - Não há dados? Fechar arquivo

```
void preencheBuffer(struct arquivo* arq, int K) {
    int i;
    if(arq->f == NULL)
        return;

    arq->pos = 0;
    arq->MAX = 0;
    for(i=0; i<K; i++){
        if(!feof(arq->f)){
            fscanf(arq->f, "%d", &arq->buffer[arq->MAX]);
            arq->MAX++;
        } else {
            fclose(arq->f);
            arq->f = NULL;
            break;
        }
    }
}
```

Tem dados no
arquivo. Lê e
coloca no buffer

Acabou os dados.
Fecha o arquivo

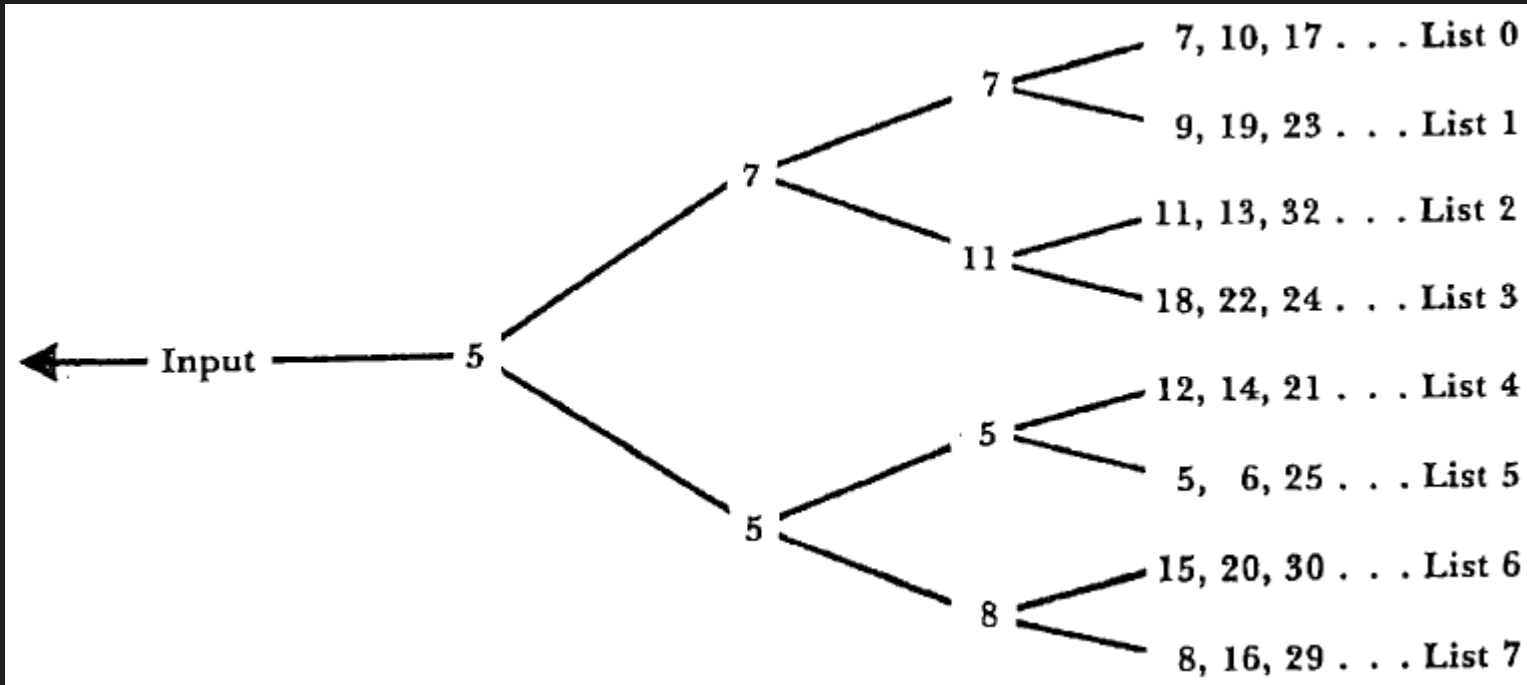
K-way merging

- O algoritmo funciona muito bem para número pequeno de *buffers* (k), até $k=8$
- Para $k > 8$ o número de comparações em sequência para achar o menor valor torna-se cara
- Solução: usar uma árvore de seleção (*selection tree*)

Árvore de seleção

- Reduz o tempo necessário para encontrar o menor valor por meio do uso de uma estrutura de dados que guarda informações sobre as chaves conforme os ciclos do laço do procedimento principal são executados
 - O valor mínimo sempre está na raiz da árvore
 - Cada chave tem uma referência para a lista de origem
 - A cada passo, pega-se a chave da raiz, lê-se o próximo elemento da lista associada, e reorganiza-se a estrutura

Árvore de seleção



Ordenação de arquivos grandes

- Como melhorar o desempenho de algoritmos que têm vários passos se não é possível melhorar o desempenho de cada passo?
- Podemos tentar realizar alguns passos em paralelo!

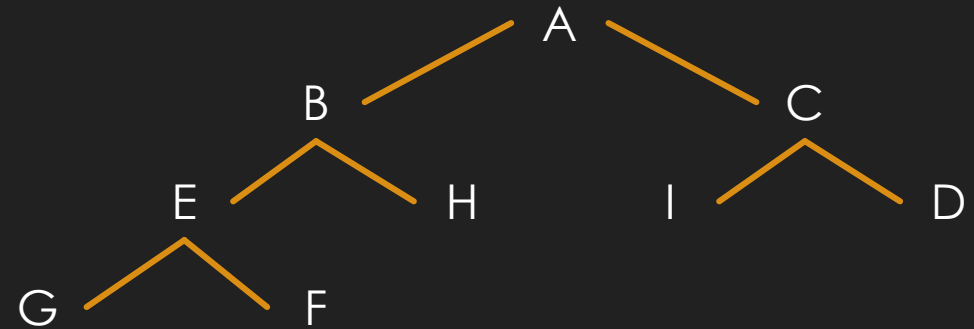
Ordenação de arquivos grandes

- Podemos usar o algoritmo **heapsort**
 - Não precisa que todos os registros sejam lidos para ordená-los
 - A ordenação ocorre em paralelo com a leitura: o registro lido é colocado na ordem correta no *heap*
 - Enquanto o *heap* é ajustado, o programa pode ler o próximo registro
 - Não usa memória extra (ordenação *in-place*), nem alocação dinâmica
 - Na gravação, os registros são retirados na ordem correta. Enquanto são gravados em disco, o *heap* pode ser ajustado para a retirada do próximo

Heapsort

- Mantém as chaves numa *heap*, que é uma árvore binária com 3 propriedades
 - Cada nó possui uma única chave, que é maior ou igual a chave do nó pai
 - É uma árvore binária completa: nós com menos de 2 filhos ficam no último ou no penúltimo nível da árvore
 - Pode ser mantida em um vetor. Os filhos a esquerda e a direita de um nó pai estão, respectivamente, nas posições
 - Filho esquerdo = $2 * \text{pai}$
 - Filho direito = $2 * \text{pai} + 1$

1	2	3	4	5	6	7	8	9
A	B	C	E	H	I	D	G	F



Heapsort

- Algoritmo em duas partes
 - Construção da *heap*: pode ser executado enquanto lê-se os dados
 - Ordenação dos dados: pode ser feita enquanto se escreve os dados no arquivo
- A ordenação heapsort requer $O(N * \log(N))$ operações independente da ordem de entrada dos dados

Sobreposição de processamento e E/S: heapsort

- Construção da *heap* durante a leitura
- Quando se insere um elemento na *heap* duas propriedades devem ser garantidas
 - A árvore deve continuar completa
 - Insere-se o elemento no último nível da árvore, o mais à esquerda possível
 - A árvore deve continuar ordenada
 - Verificamos se o elemento inserido é menor que o seu pai: se for, troca-se um pelo outro
 - Repetimos o procedimento com o novo pai
 - Processo termina quando não for mais necessário subir o novo elemento

Construção da HEAP

Inserir F

1	2	3	4	5	6	7	8	9
F								

F

Inserir D

1	2	3	4	5	6	7	8	9
D	F							



Inserir C

1	2	3	4	5	6	7	8	9
C	F	D						



Construção da HEAP

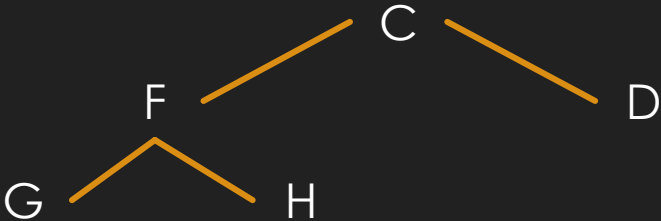
Inserir **G**

1	2	3	4	5	6	7	8	9
C	F	D	G					



Inserir **H**

1	2	3	4	5	6	7	8	9
C	F	D	G	H				



Inserir **I**

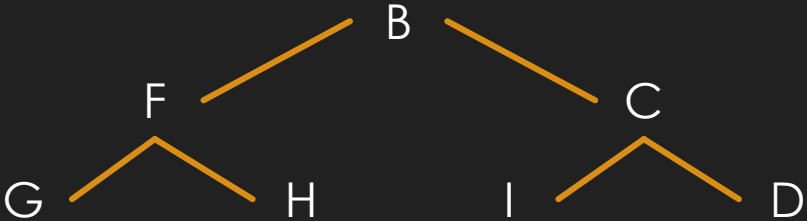
1	2	3	4	5	6	7	8	9
C	F	D	G	H	I			



Construção da HEAP

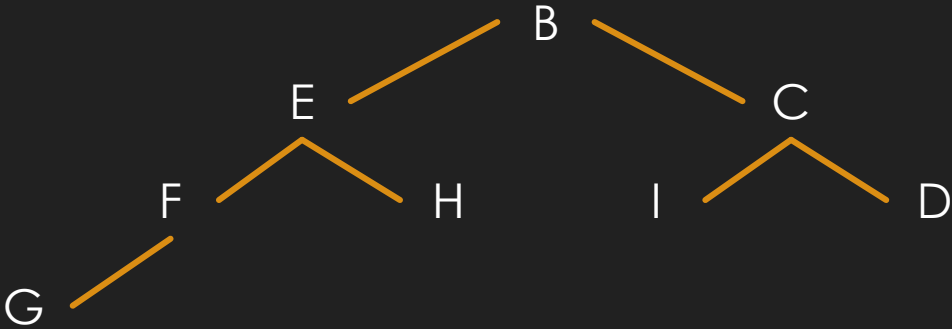
Inserir B

1	2	3	4	5	6	7	8	9
B	F	C	G	H	I	D		



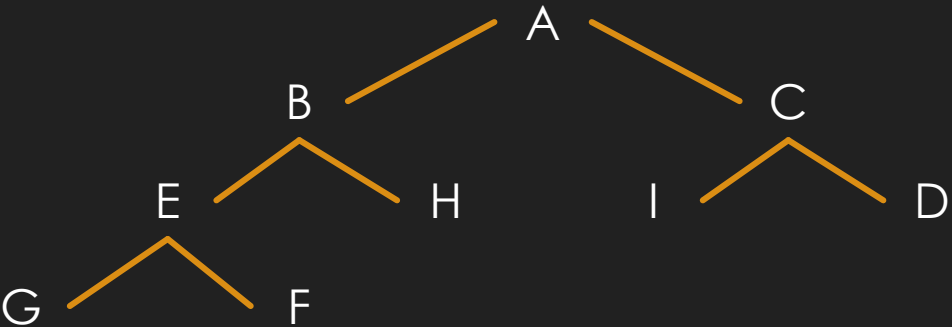
Inserir E

1	2	3	4	5	6	7	8	9
B	E	C	F	H	I	D	G	



Inserir A

1	2	3	4	5	6	7	8	9
A	B	C	E	H	I	D	G	F



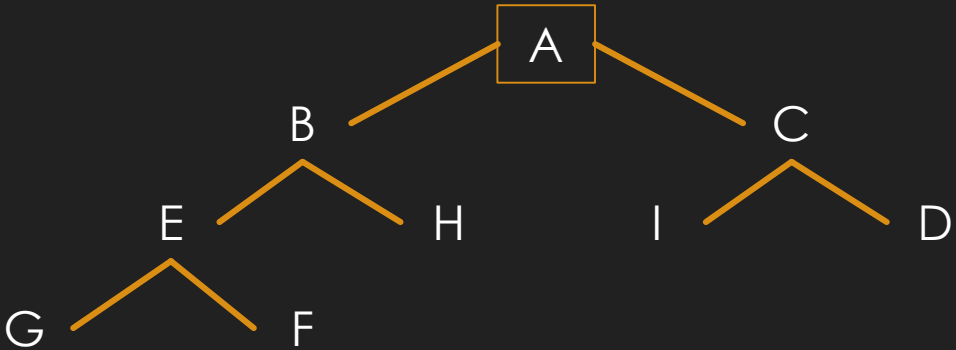
Sobreposição de processamento e E/S: heapsort

- Ordenação e escrita dos dados
 - O elemento na raiz do *heap* é o elemento de menor valor
- Remove-se os elementos a partir da raiz da *heap* e grava no arquivo garantindo
 - Manter a árvore completa
 - Passar para a raiz o nó mais a direita do último nível da árvore
 - Manter a árvore ordenada
 - Verificar se a raiz é maior que os seus filhos: em caso afirmativo, trocar pelo menor deles
 - Repetir até não ser mais necessário descer o elemento

Ordenação e escrita dos dados da heap

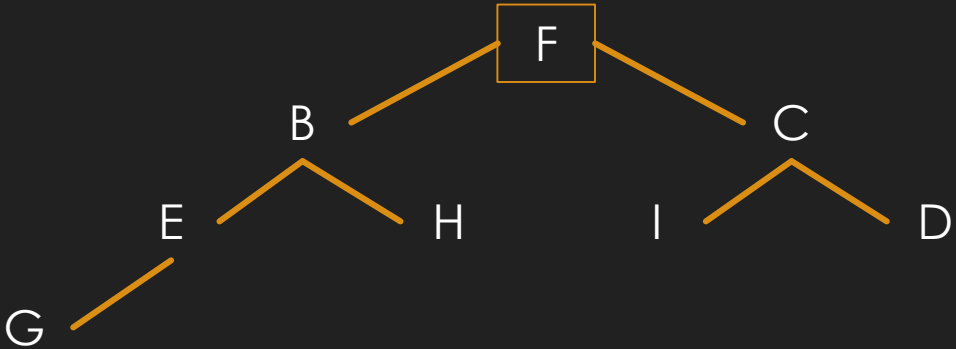
Grava **A** e
remove da heap

1	2	3	4	5	6	7	8	9
A	B	C	E	H	I	D	G	F



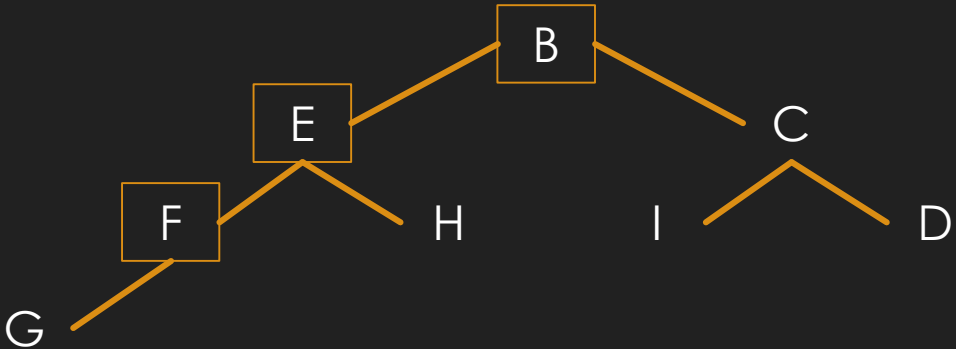
Move **F** para a
raiz da heap

1	2	3	4	5	6	7	8	9
F	B	C	E	H	I	D	G	



Corrige
ordenação da
heap

1	2	3	4	5	6	7	8	9
B	E	C	F	H	I	D	G	



Material complementar

- Estrutura de Dados em C | Aula 66 - Ordenação externa
 - <https://youtu.be/sVGbjIzgvWQ>