

## Introdução - Conceitos Básicos

- Estrutura de um registro:

```
type Item = record
    Chave: ChaveTipo;
    { outros componentes }
end;
```

- Qualquer tipo de chave sobre o qual exista uma regra de ordenação bem-definida pode ser utilizado.
- Um método de ordenação é **estável** se a ordem relativa dos itens com chaves iguais não se altera durante a ordenação.
- Alguns dos métodos de ordenação mais eficientes não são estáveis.
- A estabilidade pode ser forçada quando o método é não-estável.
- Sedgewick (1988) sugere agregar um pequeno índice a cada chave antes de ordenar, ou então aumentar a chave de alguma outra forma.

## Introdução - Conceitos Básicos

- Ordenar: processo de rearranjar um conjunto de objetos em uma ordem ascendente ou descendente.
- A ordenação visa facilitar a recuperação posterior de itens do conjunto ordenado.
  - Dificuldade de se utilizar um catálogo telefônico se os nomes das pessoas não estivessem listados em ordem alfabética.
- Notação utilizada nos algoritmos:
  - Os algoritmos trabalham sobre os registros de um arquivo.
  - Cada registro possui uma **chave** utilizada para controlar a ordenação.
  - Podem existir outros componentes em um registro.

## Conteúdo do Capítulo

4.1 Ordenação Interna	4.1.7 Ordenação em Tempo Linear
4.1.1 Seleção	* Ordenação por Contagem
4.1.2 Inserção	* Radixsort para Inteiros
4.1.3 Shellsort	* Radixsort para Cadeias de Caracteres
4.1.4 Quicksort	
4.1.5 Heapsort	4.2 Ordenação Externa
* Filas de Prioridades	4.2.1 Intercalação Balanceada de Vários Caminhos
* Heaps	4.2.2 Implementação por meio de Seleção por Substituição
4.1.6 Ordenação Parcial	4.2.3 Considerações Práticas
* Seleção Parcial	4.2.4 Intercalação Polifásica
* Inserção Parcial	4.2.5 Quicksort Externo
* Heapsort Parcial	
* Quicksort Parcial	

## Ordenação\*

Última alteração: 31 de Agosto de 2010

\*Transparências elaboradas por Charles Ornelas Almeida, Israel Guerra e Nivio Ziviani

## Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação.
- Sendo  $n$  o número registros no arquivo, as medidas de complexidade relevantes são:
  - Número de comparações  $C(n)$  entre chaves.
  - Número de movimentações  $M(n)$  de itens do arquivo.
- O uso econômico da memória disponível é um requisito primordial na ordenação interna.
- Métodos de ordenação ***in situ*** são os preferidos.
- Métodos que utilizam listas encadeadas não são muito utilizados.
- Métodos que fazem cópias dos itens a serem ordenados possuem menor importância.

## Introdução - Conceitos Básicos

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort** ou **bucketsort**.
- O método não utiliza comparação entre chaves.
- Uma das dificuldades de implementar este método está relacionada com o problema de lidar com cada monte.
- Se para cada monte nós reservarmos uma área, então a demanda por memória extra pode tornar-se proibitiva.
- O custo para ordenar um arquivo com  $n$  elementos é da ordem de  $O(n)$ .

## Introdução - Conceitos Básicos

- Exemplo de ordenação por distribuição: considere o problema de ordenar um baralho com 52 cartas na ordem:

$A < 2 < 3 < \dots < 10 < J < Q < K$

e

$\clubsuit < \diamondsuit < \heartsuit < \spadesuit$ .

- Algoritmo:

1. Distribuir as cartas em treze montes: ases, dois, três, ..., reis.
2. Colete os montes na ordem especificada.
3. Distribua novamente as cartas em quatro montes: paus, ouros, copas e espadas.
4. Colete os montes na ordem especificada.

## Introdução - Conceitos Básicos

- Classificação dos métodos de ordenação:
  - Interna: arquivo a ser ordenado cabe todo na memória principal.
  - Externa: arquivo a ser ordenado não cabe na memória principal.
- Diferenças entre os métodos:
  - Em um método de ordenação interna, qualquer registro pode ser imediatamente acessado.
  - Em um método de ordenação externa, os registros são acessados seqüencialmente ou em grandes blocos.
- A maioria dos métodos de ordenação é baseada em **comparações** das chaves.
- Existem métodos de ordenação que utilizam o princípio da **distribuição**.

## Ordenação por Seleção (2)

- O método é ilustrado abaixo:

	1	2	3	4	5	6
Chaves iniciais:	<i>O</i>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>A</i>
i = 1	<b><i>A</i></b>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<b><i>O</i></b>
i = 2	<i>A</i>	<b><i>D</i></b>	<b><i>R</i></b>	<i>E</i>	<i>N</i>	<i>O</i>
i = 3	<i>A</i>	<i>D</i>	<b><i>E</i></b>	<b><i>R</i></b>	<i>N</i>	<i>O</i>
i = 4	<i>A</i>	<i>D</i>	<i>E</i>	<b><i>N</i></b>	<b><i>R</i></b>	<i>O</i>
i = 5	<i>A</i>	<i>D</i>	<i>E</i>	<i>N</i>	<b><i>O</i></b>	<b><i>R</i></b>

- As chaves em negrito sofreram uma troca entre si.

## Ordenação Interna

- Tipos de dados e variáveis utilizados nos algoritmos de ordenação interna:

```
type Tipolindice = 0..MAXTAM;
  TipoVetor = array [Tipolindice] of Tipolitem;
var A: TipoVetor;
```

- O índice do vetor vai de 0 até *MaxTam*, devido às chaves **sentinelas**.
- O vetor a ser ordenado contém chaves nas posições de 1 até *n*.

## Ordenação por Seleção (1)

- Um dos algoritmos mais simples de ordenação.
- Algoritmo:
  - Selecione o menor item do vetor.
  - Troque-o com o item da primeira posição do vetor.
  - Repita essas duas operações com os  $n - 1$  itens restantes, depois com os  $n - 2$  itens, até que reste apenas um elemento.

## Ordenação Interna

- Classificação dos métodos de ordenação interna:
  - Métodos simples:
    - Adequados para pequenos arquivos.
    - Requerem  $O(n^2)$  comparações.
    - Produzem programas pequenos.
  - Métodos eficientes:
    - Adequados para arquivos maiores.
    - Requerem  $O(n \log n)$  comparações.
    - Usam menos comparações.
    - As comparações são mais complexas nos detalhes.
    - Métodos simples são mais eficientes para pequenos arquivos.

## Ordenação por Inserção

- O método é ilustrado abaixo:

	1	2	3	4	5	6
Chaves iniciais:	O	R	D	E	N	A
i = 2	<b>O</b>	<b>R</b>	D	E	N	A
i = 3	D	<b>O</b>	<b>R</b>	E	N	A
i = 4	D	E	<b>O</b>	<b>R</b>	N	A
i = 5	D	E	N	<b>O</b>	<b>R</b>	A
i = 6	A	D	E	N	<b>O</b>	<b>R</b>

- As chaves em negrito representam a seqüência destino.

## Ordenação por Seleção

Vantagens:

- Custo linear para o número de movimentos de registros.
- É o algoritmo a ser utilizado para arquivos com registros muito grandes.
- É muito interessante para arquivos pequenos.

Desvantagens:

- O fato de o arquivo já estar ordenado não ajuda em nada, pois o custo continua quadrático.
- O algoritmo não é **estável**.

## Ordenação por Seleção

```

procedure Selecao (var A: TipoVetor; n: TipoIndice);
var i, j, Min: TipoIndice;
    x : TipoItem;
begin
  for i := 1 to n - 1 do
    begin
      Min := i;
      for j := i + 1 to n do
        if A[j].Chave < A[Min].Chave
        then Min := j;
      x := A[Min]; A[Min] := A[i];
      A[i] := x;
    end;
end;
```

- Comparações entre chaves e movimentações de registros:

$$C(n) = \frac{n^2}{2} - \frac{n}{2}$$

$$M(n) = 3(n - 1)$$

- A atribuição  $Min := j$  é executada em média  $n \log n$  vezes, Knuth (1973).

## Ordenação por Inserção

- Seja  $M(n)$  a função que conta o número de movimentações de registros.
- O número de movimentações na  $i$ -ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

- Logo, o número de movimentos é:

Melhor caso :  $M(n) = (3 + 3 + \dots + 3) = 3(n - 1)$

Pior caso :  $M(n) = (4 + 5 + \dots + n + 2) = \frac{n^2}{2} + \frac{5n}{2} - 3$

Caso médio :  $M(n) = \frac{1}{2}(5 + 6 + \dots + n + 3) = \frac{n^2}{4} + \frac{11n}{4} - 3$

## Ordenação por Inserção

- Seja  $C(n)$  a função que conta o número de comparações.
- No anel mais interno, na  $i$ -ésima iteração, o valor de  $C_i$  é:

Melhor caso :  $C_i(n) = 1$

Pior caso :  $C_i(n) = i$

Caso médio :  $C_i(n) = \frac{1}{i}(1 + 2 + \dots + i) = \frac{i+1}{2}$

- Assumindo que todas as permutações de  $n$  são igualmente prováveis no caso médio, temos:

Melhor caso :  $C(n) = (1 + 1 + \dots + 1) = n - 1$

Pior caso :  $C(n) = (2 + 3 + \dots + n) = \frac{n^2}{2} + \frac{n}{2} - 1$

Caso médio :  $C(n) = \frac{1}{2}(3 + 4 + \dots + n + 1) = \frac{n^2}{4} + \frac{3n}{4} - 1$

## Ordenação por Inserção

Considerações sobre o algoritmo:

- O processo de ordenação pode ser terminado pelas condições:
  - Um item com chave menor que o item em consideração é encontrado.
  - O final da seqüência destino é atingido à esquerda.
- Solução:
  - Utilizar um registro **sentinela** na posição zero do vetor.

## Ordenação por Inserção

```
procedure Insercao (var A: TipoVetor; n: TipoIndice);
var i, j: TipoIndice;
    x: Tipoltem;
begin
for i := 2 to n do
begin
    x := A[i];
    j := i - 1;
    A[0] := x; { sentinel }
    while x.Chave < A[j].Chave do
begin
    A[j + 1] := A[j];
    j := j - 1;
end;
    A[j + 1] := x;
end;
end;
```

## Shellsort

- Como escolher o valor de  $h$ :
  - Seqüência para  $h$ :

$$h(s) = 3h(s - 1) + 1, \quad \text{para } s > 1$$

$$h(s) = 1, \quad \text{para } s = 1.$$

- Knuth (1973, p. 95) mostrou experimentalmente que esta seqüência é difícil de ser batida por mais de 20% em eficiência.
- A seqüência para  $h$  corresponde a 1, 4, 13, 40, 121, 364, 1.093, 3.280, ...

## Shellsort

- Os itens separados de  $h$  posições são rearranjados.
- Todo  $h$ -ésimo item leva a uma seqüência ordenada.
- Tal seqüência é dita estar  $h$ -ordenada.
- Exemplo de utilização:

	1	2	3	4	5	6
Chaves iniciais:	O	R	D	E	N	A
$h = 4$		N	A	D	E	O
$h = 2$			D	A	N	E
$h = 1$				A	D	E
					N	O
						R

- Quando  $h = 1$  Shellsort corresponde ao algoritmo de inserção.

## Shellsort

- Proposto por Shell em 1959.
- É uma extensão do algoritmo de ordenação por inserção.
- Problema com o algoritmo de ordenação por inserção:
  - Troca itens adjacentes para determinar o ponto de inserção.
  - São efetuadas  $n - 1$  comparações e movimentações quando o menor item está na posição mais à direita no vetor.
- O método de Shell contorna este problema permitindo trocas de registros distantes um do outro.

## Ordenação por Inserção

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.
- É o método a ser utilizado quando o arquivo está “quase” ordenado.
- É um bom método quando se deseja adicionar uns poucos itens a um arquivo ordenado, pois o custo é linear.
- O algoritmo de ordenação por inserção é **estável**.

## Shellsort

- Vantagens:
  - Shellsort é uma ótima opção para arquivos de tamanho moderado.
  - Sua implementação é simples e requer uma quantidade de código pequena.
- Desvantagens:
  - O tempo de execução do algoritmo é sensível à ordem inicial do arquivo.
  - O método não é **estável**,

## Shellsort: Análise

- A razão da eficiência do algoritmo ainda não é conhecida.
- Ninguém ainda foi capaz de analisar o algoritmo.
- A sua análise contém alguns problemas matemáticos muito difíceis.
- A começar pela própria seqüência de incrementos.
- O que se sabe é que cada incremento não deve ser múltiplo do anterior.
- Conjecturas referente ao número de comparações para a seqüência de Knuth:

$$\text{Conjetura 1} : C(n) = O(n^{1.25})$$

$$\text{Conjetura 2} : C(n) = O(n(\ln n)^2)$$

## Shellsort

- A implementação do Shellsort não utiliza registros **sentinelas**.
- Seriam necessários  $h$  registros sentinelas, uma para cada  $h$ -ordenação.

## Shellsort

```

procedure Shellsort (var A: TipoVetor; n: TipoIndice);
label 999;
var i, j, h: integer;
    x      : Tipolitem;
begin
    h := 1;
    repeat h := 3 * h + 1 until h >= n;
    repeat h := h div 3;
        for i := h + 1 to n do begin
            x := A[i]; j := i;
            while A[j - h].Chave > x.Chave do
                begin A[j] := A[j - h];
                    j := j - h; if j <= h then goto 999;
                end;
            999 : A[j] := x;
        end;
    until h = 1;
end;
```

## Quicksort

- Ilustração do processo de partição:

1	2	3	4	5	6
O	R	<b>D</b>	E	N	A
A	R	<b>D</b>	E	N	O
A	<b>D</b>	R	E	N	O

- O pivô  $x$  é escolhido como sendo  $A[(i + j) \text{ div } 2]$ .
- Como inicialmente  $i = 1$  e  $j = 6$ , então  $x = A[3] = D$ .
- Ao final do processo de partição  $i$  e  $j$  se cruzam em  $i = 3$  e  $j = 2$ .

## Quicksort

- Algoritmo para o particionamento:
  - Escolha arbitrariamente um **pivô**  $x$ .
  - Percorra o vetor a partir da esquerda até que  $A[i] \geq x$ .
  - Percorra o vetor a partir da direita até que  $A[j] \leq x$ .
  - Troque  $A[i]$  com  $A[j]$ .
  - Continue este processo até os apontadores  $i$  e  $j$  se cruzarem.
- Ao final, o vetor  $A[\text{Esq..Dir}]$  está particionado de tal forma que:
  - Os itens em  $A[\text{Esq}], A[\text{Esq} + 1], \dots, A[j]$  são menores ou iguais a  $x$ .
  - Os itens em  $A[i], A[i + 1], \dots, A[\text{Dir}]$  são maiores ou iguais a  $x$ .

## Quicksort

- A parte mais delicada do método é o processo de partição.
- O vetor  $A[\text{Esq..Dir}]$  é rearranjado por meio da escolha arbitrária de um **pivô**  $x$ .
- O vetor  $A$  é particionado em duas partes:
  - A parte esquerda com chaves menores ou iguais a  $x$ .
  - A parte direita com chaves maiores ou iguais a  $x$ .

## Quicksort

- Proposto por Hoare em 1960 e publicado em 1962.
- É o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é o mais utilizado.
- A idéia básica é dividir o problema de ordenar um conjunto com  $n$  itens em dois problemas menores.
- Os problemas menores são ordenados independentemente.
- Os resultados são combinados para produzir a solução final.

## Quicksort: Análise

- Seja  $C(n)$  a função que conta o número de comparações.
- Pior caso:
 
$$C(n) = O(n^2)$$
  - O pior caso ocorre quando, sistematicamente, o pivô é escolhido como sendo um dos extremos de um arquivo já ordenado.
  - Isto faz com que o procedimento Ordena seja chamado recursivamente  $n$  vezes, eliminando apenas um item em cada chamada.
  - O pior caso pode ser evitado empregando pequenas modificações no algoritmo.
  - Para isso basta escolher três itens quaisquer do vetor e usar a **mediana dos três** como pivô.

## Quicksort

- Exemplo do estado do vetor em cada chamada recursiva do procedimento Ordena:

	1	2	3	4	5	6
Chaves iniciais:	<i>O</i>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>A</i>
1		<i>A</i>	<b>D</b>	<i>R</i>	<i>E</i>	<i>N</i>
2			<b>A</b>	<i>D</i>		
3				<b>E</b>	<i>R</i>	<i>N</i>
4					<b>N</b>	<i>R</i>
5						<b>O</b>
						<b>R</b>
	<i>A</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>O</i>	<i>R</i>

## Quicksort

Procedimento Quicksort:

```

procedure QuickSort (var A: Vetor; var n: Indice);
{— Entra aqui o procedimento Particao da transparencia 32 —}
  procedure Ordena (Esq, Dir: Indice);
    var i, j: Indice;
    begin
      particao (Esq, Dir, i, j);
      if Esq < j then Ordena (Esq, j);
      if i < Dir then Ordena (i, Dir);
    end;
    begin
      Ordena (1, n);
    end;
  
```

## Quicksort

Procedimento Particao:

```

procedure Particao (Esq, Dir: Indice; var i, j: Indice);
var x, w: Item;
begin i := Esq; j := Dir;
x := A[(i + j) div 2]; { obtém o pivô x }
repeat while x.Chave > A[i].Chave do i := i + 1;
  while x.Chave < A[j].Chave do j := j - 1;
  if i <= j
  then begin w := A[i]; A[i] := A[j]; A[j] := w;
        i := i + 1; j := j - 1;
      end;
until i > j;
end;
  
```

- O anel interno do procedimento Particao é extremamente simples.
- Razão pela qual o algoritmo Quicksort é tão rápido.

## Heapsort

### Filas de Prioridades

- É uma estrutura de dados onde a chave de cada item reflete sua habilidade relativa de abandonar o conjunto de itens rapidamente.
- Aplicações:
  - SOs usam filas de prioridades, nas quais as chaves representam o tempo em que eventos devem ocorrer.
  - Métodos numéricos iterativos são baseados na seleção repetida de um item com maior (menor) valor.
  - Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.

## Heapsort

- Possui o mesmo princípio de funcionamento da ordenação por seleção.
- Algoritmo:
  1. Selecione o menor item do vetor.
  2. Troque-o com o item da primeira posição do vetor.
  3. Repita estas operações com os  $n - 1$  itens restantes, depois com os  $n - 2$  itens, e assim sucessivamente.
- O custo para encontrar o menor (ou o maior) item entre  $n$  itens é  $n - 1$  comparações.
- Isso pode ser reduzido utilizando uma fila de prioridades.

## Quicksort

- Vantagens:
  - É extremamente eficiente para ordenar arquivos de dados.
  - Necessita de apenas uma pequena pilha como memória auxiliar.
  - Requer cerca de  $n \log n$  comparações em média para ordenar  $n$  itens.
- Desvantagens:
  - Tem um pior caso  $O(n^2)$  comparações.
  - Sua implementação é muito delicada e difícil:
    - \* Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
  - O método não é **estável**.

## Quicksort: Análise

- Melhor caso:
 
$$C(n) = 2C(n/2) + n = n \log n - n + 1$$
- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.
- Caso médio de acordo com Sedgewick e Flajolet (1996, p. 17):
 
$$C(n) \approx 1,386n \log n - 0,846n,$$
- Isso significa que em média o tempo de execução do Quicksort é  $O(n \log n)$ .

## Heapsort

### Filas de Prioridades - Algoritmos de Ordenação

- As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação.
- Basta utilizar repetidamente a operação Insere para construir a fila de prioridades.
- Em seguida, utilizar repetidamente a operação Retira para receber os itens na ordem reversa.
- O uso de listas lineares não ordenadas corresponde ao método da seleção.
- O uso de listas lineares ordenadas corresponde ao método da inserção.
- O uso de *heaps* corresponde ao método Heapsort.

## Heapsort

### Filas de Prioridades - Representação

- A melhor representação é através de uma estruturas de dados chamada *heap*:
  - Neste caso, Constrói é  $O(n)$ .
  - Insere, Retira, Substitui e Altera são  $O(\log n)$ .
- Observação:**  
Para implementar a operação Ajunta de forma eficiente e ainda preservar um custo logarítmico para as operações Insere, Retira, Substitui e Altera é necessário utilizar estruturas de dados mais sofisticadas, tais como árvores binomiais (Vuillemin, 1978).

## Heapsort

### Filas de Prioridades - Representação

- Representação através de uma lista linear ordenada:
  - Neste caso, Constrói leva tempo  $O(n \log n)$ .
  - Insere é  $O(n)$ .
  - Retira é  $O(1)$ .
  - Ajunta é  $O(n)$ .
- Representação é através de uma lista linear não ordenada:
  - Neste caso, Constrói tem custo linear.
  - Insere é  $O(1)$ .
  - Retira é  $O(n)$ .
  - Ajunta é  $O(1)$  para apontadores e  $O(n)$  para arranjos.

## Heapsort

### Filas de Prioridades - Tipo Abstrato de Dados

- Operações:
  - Constrói uma fila de prioridades a partir de um conjunto com  $n$  itens.
  - Informa qual é o maior item do conjunto.
  - Retira o item com maior chave.
  - Insere um novo item.
  - Aumenta o valor da chave do item  $i$  para um novo valor que é maior que o valor atual da chave.
  - Substitui o maior item por um novo item, a não ser que o novo item seja maior.
  - Altera a prioridade de um item.
  - Remove um item qualquer.
  - Ajunta duas filas de prioridades em uma única.

## Heap

- Na representação do *heap* em um arranjo, a maior chave está sempre na posição 1 do vetor.
- Os algoritmos para implementar as operações sobre o *heap* operam ao longo de um dos caminhos da árvore.
- Um algoritmo elegante para construir o *heap* foi proposto por Floyd em 1964.
- O algoritmo não necessita de nenhuma memória auxiliar.
- Dado um vetor  $A[1], A[2], \dots, A[n]$ .
- Os itens  $A[n/2 + 1], A[n/2 + 2], \dots, A[n]$  formam um *heap*:
  - Neste intervalo não existem dois índices  $i$  e  $j$  tais que  $j = 2i$  ou  $j = 2i + 1$ .

## Heap

- As chaves na árvore satisfazem a condição do *heap*.
- A chave em cada nó é maior do que as chaves em seus filhos.
- A chave no nó raiz é a maior chave do conjunto.
- Uma árvore binária completa pode ser representada por um array:

1	2	3	4	5	6	7
S	R	O	E	N	A	D

- A representação é extremamente compacta.
- Permite caminhar pelos nós da árvore facilmente.
- Os filhos de um nó  $i$  estão nas posições  $2i$  e  $2i + 1$ .
- O pai de um nó  $i$  está na posição  $i \text{ div } 2$ .

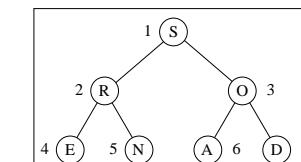
## Heap

- **Árvore binária completa:**
  - Os nós são numerados de 1 a  $n$ .
  - O primeiro nó é chamado raiz.
  - O nó  $\lfloor k/2 \rfloor$  é o pai do nó  $k$ , para  $1 < k \leq n$ .
  - Os nós  $2k$  e  $2k + 1$  são os filhos à esquerda e à direita do nó  $k$ , para  $1 \leq k \leq \lfloor k/2 \rfloor$ .

## Heap

- É uma seqüência de itens com chaves  $c[1], c[2], \dots, c[n]$ , tal que:
 
$$c[i] \geq c[2i],$$

$$c[i] \geq c[2i + 1],$$
 para todo  $i = 1, 2, \dots, n/2$ .
- A definição pode ser facilmente visualizada em uma árvore binária completa:



## Heap

Programa para construir o *heap*:

```
{-- Usa o procedimento Refaz --}
procedure Constroi (var A: TipoVetor; n: TipoIndice);
var Esq: TipoIndice;
begin
  Esq := n div 2 + 1;
  while Esq > 1 do
    begin
      Esq := Esq - 1;
      Refaz (Esq, n, A);
    end;
end;
```

## Heap

Programa para refazer a condição de *heap*:

```
procedure Refaz (Esq, Dir: TipoIndice; var A: TipoVetor);
label 999;
var i: TipoIndice;
j: integer;
x: Tipoltem;
begin i := Esq; j := 2 * i;
x := A[i];
while j <= Dir do begin
  if j < Dir then if A[j].Chave < A[j + 1].Chave then j := j + 1;
  if x.Chave >= A[j].Chave then goto 999;
  A[i] := A[j]; i := j; j := 2 * i;
end;
999 : A[i] := x;
end;
```

## Heap

- A condição de *heap* é violada:
  - O *heap* é refeito trocando os itens D e S.
- O item R é incluído no *heap* ( $Esq = 2$ ), o que não viola a condição de *heap*.
- O item O é incluído no *heap* ( $Esq = 1$ ).
- A Condição de *heap* violada:
  - O *heap* é refeito trocando os itens O e S, encerrando o processo.

O Programa que implementa a operação que informa o item com maior chave:

```
function Max (var A: TipoVetor): Tipoltem;
begin Max := A[1];
end;
```

## Heap

	1	2	3	4	5	6	7
Chaves iniciais:	O	R	D	E	N	A	S
Esq = 3	O	R	<b>S</b>	E	N	A	<b>D</b>
Esq = 2	O	R	S	E	N	A	D
Esq = 1	<b>S</b>	R	<b>O</b>	E	N	A	D

- Os itens de  $A[4]$  a  $A[7]$  formam um *heap*.
- O *heap* é estendido para a esquerda ( $Esq = 3$ ), englobando o item  $A[3]$ , pai dos itens  $A[6]$  e  $A[7]$ .

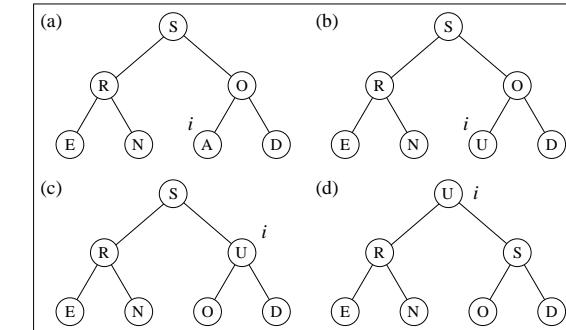
## Heap

Programa que implementa a operação de inserir um novo item no *heap*:

```
const Infinito = MaxInt;
procedure Insere (var x: Item; var A: Vetor; var n: Indice);
begin
  n := n + 1;
  A[n] := x;
  A[n].Chave := -Infinito;
  AumentaChave(n, x.Chave, A);
end;
```

## Heap

- Exemplo da operação de aumentar o valor da chave do item na posição *i*:



- O tempo de execução do procedimento *AumentaChave* em um item do *heap* é  $O(\log n)$ .

## Heap

Programa que implementa a operação de aumentar o valor da chave do item *i*:

```
procedure AumentaChave (i: TipoIndice; ChaveNova: TipoChave; var A: TipoVetor);
var k: integer;
    x: TipItem;
begin
  if ChaveNova < A[i].Chave
  then writeln('Erro: ChaveNova menor que a chave atual')
  else begin A[i].Chave := ChaveNova;
        while (i>1) and (A[i div 2].Chave < A[i].Chave)
        do begin x := A[i div 2]; A[i div 2] := A[i];
              A[i] := x; i := i div 2;
        end;
  end;
end;
```

## Heap

Programa que implementa a operação de retirar o item com maior chave:

```
function RetiraMax (var A: TipoVetor; var n: TipoIndice): TipItem;
begin
  if n < 1
  then writeln('Erro: heap vazio')
  else begin
    RetiraMax := A[1];
    A[1] := A[n];
    n := n - 1;
    Refaz (1, n, A);
  end;
end;
```

## Heapsort

- Vantagens:
  - O comportamento do Heapsort é sempre  $O(n \log n)$ , qualquer que seja a entrada.
- Desvantagens:
  - O anel interno do algoritmo é bastante complexo se comparado com o do Quicksort.
  - O Heapsort não é **estável**.
- Recomendado:
  - Para aplicações que não podem tolerar eventualmente um caso desfavorável.
  - Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap*.

1	2	3	4	5	6	7
S	R	O	E	N	A	D
<b>R</b>	<b>N</b>	O	E	<b>D</b>	A	<b>S</b>
O	N	<b>A</b>	E	D	R	
<b>N</b>	<b>E</b>	A	<b>D</b>	O		
E	D	A	N			
<b>D</b>	<b>A</b>	E				
A	D					

- Exemplo de aplicação do Heapsort:

- O caminho seguido pelo procedimento Refaz para reconstituir a condição do *heap* está em negrito.
- Por exemplo, após a troca dos itens S e D na segunda linha da Figura, o item D volta para a posição 5, após passar pelas posições 1 e 2.

## Heapsort

Programa que mostra a implementação do Heapsort:

```
procedure Heapsort (var A: Vetor; var n: Indice);
var Esq, Dir: Indice;
    x : Item;
{— Entra aqui o procedimento Refaz—}
{— Entra aqui o procedimento Constroi—}
begin
  Constroi(A, n); { constroi o heap }
  Esq := 1; Dir := n;
  while Dir > 1 do { ordena o vetor }
    begin
      x := A[1]; A[1] := A[Dir]; A[Dir] := x;
      Dir := Dir - 1;
      Refaz (Esq, Dir, A);
    end;
  end;
```

### Análise

- O procedimento Refaz gasta cerca de  $\log n$  operações, no pior caso.
- Logo, Heapsort gasta um tempo de execução proporcional a  $n \log n$ , no pior caso.

## Heapsort

- Algoritmo:

1. Construir o *heap*.
2. Troque o item na posição 1 do vetor (raiz do *heap*) com o item da posição  $n$ .
3. Use o procedimento Refaz para reconstituir o *heap* para os itens  $A[1], A[2], \dots, A[n - 1]$ .
4. Repita os passos 2 e 3 com os  $n - 1$  itens restantes, depois com os  $n - 2$ , até que reste apenas um item.

## Comparação entre os Métodos

### Tempo de execução:

- Registros na ordem descendente:

	500	5.000	10.000	30.000
Inserção	40,3	305	575	–
Seleção	29,3	221	417	–
Shellsort	1,5	1,5	1,6	1,6
Quicksort	1	1	1	1
Heapsort	2,5	2,7	2,7	2,9

## Comparação entre os Métodos

- Registros na ordem ascendente:

	500	5.000	10.000	30.000
Inserção	1	1	1	1
Seleção	128	1.524	3.066	–
Shellsort	3,9	6,8	7,3	8,1
Quicksort	4,1	6,3	6,8	7,1
Heapsort	12,2	20,8	22,4	24,6

## Comparação entre os Métodos

### Tempo de execução:

- Observação: O método que levou menos tempo real para executar recebeu o valor 1 e os outros receberam valores relativos a ele.
- Registros na ordem aleatória:

	5.00	5.000	10.000	30.000
Inserção	11,3	87	161	–
Seleção	16,2	124	228	–
Shellsort	1,2	1,6	1,7	2
Quicksort	1	1	1	1
Heapsort	1,5	1,6	1,6	1,6

## Comparação entre os Métodos

### Complexidade:

	Complexidade
Inserção	$O(n^2)$
Seleção	$O(n^2)$
Shellsort	$O(n \log n)$
Quicksort	$O(n \log n)$
Heapsort	$O(n \log n)$

- Apesar de não se conhecer analiticamente o comportamento do Shellsort, ele é considerado um método eficiente).

## Comparação entre os Métodos

### Método da Seleção:

- É vantajoso quanto ao número de movimentos de registros, que é  $O(n)$ .
- Deve ser usado para arquivos com registros muito grandes, desde que o tamanho do arquivo não exceda 1.000 elementos.

## Comparação entre os Métodos

### Influência da ordem inicial dos registros:

	Shellsort			Quicksort			Heapsort		
	5.000	10.000	30.000	5.000	10.000	30.000	5.000	10.000	30.000
Asc	1	1	1	1	1	1	1,1	1,1	1,1
Des	1,5	1,6	1,5	1,1	1,1	1,1	1	1	1
Ale	2,9	3,1	3,7	1,9	2,0	2,0	1,1	1	1

1. Shellsort é bastante sensível à ordenação ascendente ou descendente da entrada.
2. Em arquivos do mesmo tamanho, o Shellsort executa mais rápido para arquivos ordenados.
3. Quicksort é sensível à ordenação ascendente ou descendente da entrada.
4. Em arquivos do mesmo tamanho, o Quicksort executa mais rápido para arquivos ordenados.
5. O Quicksort é o mais rápido para qualquer tamanho para arquivos na ordem ascendente.
6. O Heapsort praticamente não é sensível à ordenação da entrada.

## Comparação entre os Métodos

### Método da Inserção:

- É o mais interessante para arquivos com menos do que 20 elementos.
- O método é estável.
- Possui comportamento melhor do que o método da **bolha** (**Bubblesort**) que também é estável.
- Sua implementação é tão simples quanto as implementações do Bubblesort e Seleção.
- Para arquivos já ordenados, o método é  $O(n)$ .
- O custo é linear para adicionar alguns elementos a um arquivo já ordenado.

## Comparação entre os Métodos

### Observações sobre os métodos:

1. Shellsort, Quicksort e Heapsort têm a mesma ordem de grandeza.
2. O Quicksort é o mais rápido para todos os tamanhos aleatórios experimentados.
3. A relação Heapsort/Quicksort é constante para todos os tamanhos.
4. A relação Shellsort/Quicksort aumenta se o número de elementos aumenta.
5. Para arquivos pequenos (500 elementos), o Shellsort é mais rápido que o Heapsort.
6. Se a entrada aumenta, o Heapsort é mais rápido que o Shellsort.
7. O Inserção é o mais rápido se os elementos estão ordenados.
8. O Inserção é o mais lento para qualquer tamanho se os elementos estão em ordem descendente.
9. Entre os algoritmos de custo  $O(n^2)$ , o Inserção é melhor para todos os tamanhos aleatórios experimentados.

## Comparação entre os Métodos

### Heapsort:

- É um método de ordenação elegante e eficiente.
- Apesar de ser cerca de duas vezes mais lento do que o Quicksort, não necessita de nenhuma memória adicional.
- Executa sempre em tempo proporcional a  $n \log n$ ,
- Aplicações que não podem tolerar eventuais variações no tempo esperado de execução devem usar o Heapsort.

## Comparação entre os Métodos

### Quicksort:

- Geralmente se usa a mediana de uma amostra de três elementos para evitar o pior caso.
- Esta solução melhora o caso médio ligeiramente.
- Outra importante melhoria para o desempenho do Quicksort é evitar chamadas recursivas para pequenos subarquivos.
- Para isto, basta chamar um método de ordenação simples nos arquivos pequenos.
- A melhoria no desempenho é significativa, podendo chegar a 20% para a maioria das aplicações (Sedgewick, 1988).

## Comparação entre os Métodos

### Quicksort:

- É o algoritmo mais eficiente que existe para uma grande variedade de situações.
- É um método bastante frágil no sentido de que qualquer erro de implementação pode ser difícil de ser detectado.
- O algoritmo é recursivo, o que demanda uma pequena quantidade de memória adicional.
- Seu desempenho é da ordem de  $\mathcal{O}(n^2)$  operações no pior caso.
- O principal cuidado a ser tomado é com relação à escolha do pivô.
- A escolha do elemento do meio do arranjo melhora muito o desempenho quando o arquivo está total ou parcialmente ordenado.
- O pior caso tem uma probabilidade muito remota de ocorrer quando os elementos forem aleatórios.

## Comparação entre os Métodos

### Shellsort:

- É o método a ser escolhido para a maioria das aplicações por ser muito eficiente para arquivos de tamanho moderado.
- Mesmo para arquivos grandes, o método é cerca de apenas duas vezes mais lento do que o Quicksort.
- Sua implementação é simples e geralmente resulta em um programa pequeno.
- Não possui um pior caso ruim e quando encontra um arquivo parcialmente ordenado trabalha menos.

## Ordenação Parcial

### Algoritmos considerados:

- Seleção parcial.
- Inserção parcial.
- Heapsort parcial.
- Quicksort parcial.

## Ordenação Parcial

- Consiste em obter os  $k$  primeiros elementos de um arranjo ordenado com  $n$  elementos.
- Quando  $k = 1$ , o problema se reduz a encontrar o mínimo (ou o máximo) de um conjunto de elementos.
- Quando  $k = n$  caímos no problema clássico de ordenação.

## Comparação entre os Métodos

### Considerações finais:

- Para registros muito grandes é desejável que o método de ordenação realize apenas  $n$  movimentos dos registros.
- Com o uso de uma **ordenação indireta** é possível se conseguir isso.
- Suponha que o arquivo  $A$  contenha os seguintes registros:  $A[1], A[2], \dots, A[n]$ .
- Seja  $P$  um arranjo  $P[1], P[2], \dots, P[n]$  de apontadores.
- Os registros somente são acessados para fins de comparações e toda movimentação é realizada sobre os apontadores.
- Ao final,  $P[1]$  contém o índice do menor elemento de  $A$ ,  $P[2]$  o índice do segundo menor e assim sucessivamente.
- Essa estratégia pode ser utilizada para qualquer dos métodos de ordenação interna.

## Inserção Parcial

- Pode ser obtido a partir do algoritmo de ordenação por Inserção por meio de uma modificação simples:
  - Tendo sido ordenados os primeiros  $k$  itens, o item da  $k$ -ésima posição funciona como um pivô.
  - Quando um item entre os restantes é menor do que o pivô, ele é inserido na posição correta entre os  $k$  itens de acordo com o algoritmo original.

## Seleção Parcial

```

procedure SelecaoParcial (var A: TipoVetor;
                        n, k: TipoIndice);
var i, j, Min: TipoIndice;
    x: TipoItem;
begin
  for i := 1 to k do
    begin
      Min := i;
      for j := i + 1 to n do
        if A[j].Chave < A[Min].Chave
        then Min := j;
      x := A[Min]; A[Min] := A[i]; A[i] := x;
    end;
end;
  
```

### Análise:

- Comparações entre chaves e movimentações de registros:

$$C(n) = kn - \frac{k^2}{2} - \frac{k}{2}$$

$$M(n) = 3k$$

## Seleção Parcial

- É muito simples de ser obtido a partir da implementação do algoritmo de ordenação por seleção.
- Possui um comportamento espetacular quanto ao número de movimentos de registros:
  - Tempo de execução é linear no tamanho de  $k$ .

## Inserção Parcial: Análise

- O número de movimentações na  $i$ -ésima iteração é:

$$M_i(n) = C_i(n) - 1 + 3 = C_i(n) + 2$$

- Logo, o número de movimentos é:

Melhor caso :  $M(n) = (3 + 3 + \dots + 3) = 3(n - 1)$

Pior caso :  $M(n) = (4 + 5 + \dots + k + 2 + (k + 1)(n - k))$   
 $= kn + n - \frac{k^2}{2} + \frac{3k}{2} - 3$

Caso médio :  $M(n) = \frac{1}{2}(5 + 6 + \dots + k + 3 + (k + 1)(n - k))$   
 $= \frac{kn}{2} + \frac{n}{2} - \frac{k^2}{4} + \frac{5k}{4} - 2$

- O número mínimo de comparações e movimentos ocorre quando os itens estão originalmente em ordem.
- O número máximo ocorre quando os itens estão originalmente na ordem reversa.

## Inserção Parcial: Análise

- No anel mais interno, na  $i$ -ésima iteração o valor de  $C_i$  é:

Melhor caso :  $C_i(n) = 1$

Pior caso :  $C_i(n) = i$

Caso médio :  $C_i(n) = \frac{1}{i}(1 + 2 + \dots + i) = \frac{i+1}{2}$

- Assumindo que todas as permutações de  $n$  são igualmente prováveis, o número de comparações é:

Melhor caso :  $C(n) = (1 + 1 + \dots + 1) = n - 1$

Pior caso :  $C(n) = (2 + 3 + \dots + k + (k + 1)(n - k))$   
 $= kn + n - \frac{k^2}{2} - \frac{k}{2} - 1$

Caso médio :  $C(n) = \frac{1}{2}(3 + 4 + \dots + k + 1 + (k + 1)(n - k))$   
 $= \frac{kn}{2} + \frac{n}{2} - \frac{k^2}{4} + \frac{k}{4} - 1$

## Inserção Parcial: Preserva Restante do Vetor

```

procedure InsercaoParcial2 (var A: TipoVetor; n, k: TipoIndice);
{— Preserva o restante do vetor —}
var i, j: TipoIndice; x: Tipolitem;
begin
  for i := 2 to n do
    begin
      x := A[i];
      if i > k
      then begin j := k; if x.Chave < A[k].Chave then A[i] := A[k]; end
      else j := i - 1;
      A[0] := x; { sentinel }
      while x.Chave < A[j].Chave do
        begin if j < k then A[j + 1] := A[j]; j := j - 1; end;
        if j < k then A[j+1] := x;
      end;
    end;

```

## Inserção Parcial

```

procedure InsercaoParcial (var A: TipoVetor;
n, k: TipoIndice);
{— Não preserva o restante do vetor —}
var i, j: TipoIndice; x: Tipolitem;
begin
  for i := 2 to n do
    begin
      x := A[i];
      if i > k then j := k else j := i - 1;
      A[0] := x; { sentinel }
      while x.Chave < A[j].Chave do
        begin A[j + 1] := A[j]; j := j - 1; end;
        A[j+1] := x;
      end;
    end;

```

- A modificação realizada verifica o momento em que  $i$  se torna maior do que  $k$  e então passa a considerar o valor de  $j$  igual a  $k$  a partir deste ponto.

## Quicksort Parcial

- Assim como o Quicksort, o Quicksort Parcial é o algoritmo de ordenação parcial mais rápido em várias situações.
- A alteração no algoritmo para que ele ordene apenas os  $k$  primeiros itens dentre  $n$  itens é muito simples.
- Basta abandonar a partição à direita toda vez que a partição à esquerda contiver  $k$  ou mais itens.
- Assim, a única alteração necessária no Quicksort é evitar a chamada recursiva `Ordena(i,Dir)`.

## Heapsort Parcial: Análise:

- O HeapsortParcial deve construir um *heap* a um custo  $O(n)$ .
- O procedimento `Refaz` tem custo  $O(\log n)$ .
- O procedimento `HeapsortParcial` chama o procedimento `Refaz`  $k$  vezes.
- Logo, o algoritmo apresenta a complexidade:

$$O(n + k \log n) = \begin{cases} O(n) & \text{se } k \leq \frac{n}{\log n} \\ O(k \log n) & \text{se } k > \frac{n}{\log n} \end{cases}$$

## Heapsort Parcial

```

procedure HeapsortParcial (var A: TipoVetor; n, k: TipoIndice);
{— Coloca menor em A[n], ..., k-ésimo menor em A[n-k] —}
var Esq, Dir: TipoIndice;
    x      : Tipoltem;
    Aux   : integer;
{— Entram aqui os procedimentos Refaz e Constroi das transparencias 50 e 51 —}
begin
    Constroi(A, n); { constroi o heap }
    Aux := 0; Esq := 1; Dir := n;
    while Aux < k do { ordena o vetor }
        begin
            x := A[1]; A[1] := A[n - Aux]; A[n - Aux] := x;
            Dir := Dir - 1; Aux := Aux + 1;
            Refaz (Esq, Dir, A);
        end;
    end;

```

## Heapsort Parcial

- Utiliza um tipo abstrato de dados *heap* para informar o menor item do conjunto.
- Na primeira iteração, o menor item que está em  $A[1]$  (raiz do *heap*) é trocado com o item que está em  $A[n]$ .
- Em seguida o *heap* é refeito.
- Novamente, o menor está em  $A[1]$ , troque-o com  $A[n-1]$ .
- Repita as duas últimas operações até que o  $k$ -ésimo menor seja trocado com  $A[n - k]$ .
- Ao final, os  $k$  menores estão nas  $k$  últimas posições do vetor  $A$ .

## Comparação entre os Métodos de Ordenação Parcial (1)

$n, k$	Seleção	Quicksort	Inserção	Inserção2	Heapsort
$n : 10^1 \ k : 10^0$	1	2,5	1	1,2	1,7
$n : 10^1 \ k : 10^1$	1,2	2,8	1	1,1	2,8
$n : 10^2 \ k : 10^0$	1	3	1,1	1,4	4,5
$n : 10^2 \ k : 10^1$	1,9	2,4	1	1,2	3
$n : 10^2 \ k : 10^2$	3	1,7	1	1,1	2,3
$n : 10^3 \ k : 10^0$	1	3,7	1,4	1,6	9,1
$n : 10^3 \ k : 10^1$	4,6	2,9	1	1,2	6,4
$n : 10^3 \ k : 10^2$	11,2	1,3	1	1,4	1,9
$n : 10^3 \ k : 10^3$	15,1	1	3,9	4,2	1,6
$n : 10^5 \ k : 10^0$	1	2,4	1,1	1,1	5,3
$n : 10^5 \ k : 10^1$	5,9	2,2	1	1	4,9
$n : 10^5 \ k : 10^2$	67	2,1	1	1,1	4,8
$n : 10^5 \ k : 10^3$	304	1	1,1	1,3	2,3
$n : 10^5 \ k : 10^4$	1445	1	33,1	43,3	1,7
$n : 10^5 \ k : 10^5$	$\infty$	1	$\infty$	$\infty$	1,9

## Quicksort Parcial: Análise:

- A análise do Quicksort é difícil.
- O comportamento é muito sensível à escolha do pivô.
- Podendo cair no melhor caso  $O(k \log k)$ .
- Ou em algum valor entre o melhor caso e  $O(n \log n)$ .

## Quicksort Parcial

```

procedure QuickSortParcial (var A: TipoVetor; n, k: TipoIndice);
{— Entra aqui o procedimento Particao —}
procedure Ordena (Esq, Dir, k: TipoIndice);
var i, j: TipoIndice;
begin
  Particao (Esq, Dir, i, j);
  if (j-Esq) >= (k-1)
  then begin if Esq < j then Ordena (Esq, j, k) end
  else begin
    if Esq < j then Ordena (Esq, j, k);
    if i < Dir then Ordena (i, Dir, k);
    end;
  end; { Ordena }
begin
  Ordena (1, n, k);
end;

```

## Quicksort Parcial

Chaves iniciais:	1	2	3	4	5	6
1	<i>O</i>	<i>R</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>A</i>
2		<i>A</i>	<b>D</b>	<i>R</i>	<i>E</i>	<i>N</i>
3			<b>E</b>	<i>R</i>	<i>N</i>	<i>O</i>
4				<b>N</b>	<i>R</i>	<i>O</i>
5					<i>O</i>	<b>R</b>
	<i>A</i>	<i>D</i>	<i>E</i>	<i>N</i>	<i>O</i>	<i>R</i>

- Considere  $k = 3$  e D o pivô para gerar as linhas 2 e 3.
- A partição à esquerda contém dois itens e a partição à direita, quatro itens.
- A partição à esquerda contém menos do que  $k$  itens.
- Logo, a partição direita não pode ser abandonada.
- Considere E o pivô na linha 3.
- A partição à esquerda contém três itens e a partição à direita também.
- Assim, a partição à direita pode ser abandonada.

## Ordenação por Contagem

- Este método assume que cada item do vetor  $A$  é um número inteiro entre 0 e  $k$ .
- O algoritmo conta, para cada item  $x$ , o número de itens antes de  $x$ .
- A seguir, cada item é colocado no vetor de saída na sua posição definitiva.

## Ordenação em Tempo Linear

- Nos algoritmos apresentados a seguir não existe comparação entre chaves.
- Eles têm complexidade de tempo linear na prática.
- Necessitam manter uma cópia em memória dos itens a serem ordenados e uma área temporária de trabalho.

## Comparação entre os Métodos de Ordenação Parcial

- Para valores de  $k$  até 1.000, o método da InserçãoParcial é imbatível.
- O QuicksortParcial nunca ficar muito longe da InserçãoParcial.
- Na medida em que o  $k$  cresce, o QuicksortParcial é a melhor opção.
- Para valores grandes de  $k$ , o método da InserçãoParcial se torna ruim.
- Um método indicado para qualquer situação é o QuicksortParcial.
- O HeapsortParcial tem comportamento parecido com o do QuicksortParcial.
- No entanto, o HeapsortParcial é mais lento.

## Comparação entre os Métodos de Ordenação Parcial (2)

$n, k$	Seleção	Quicksort	Inserção	Inserção2	Heapsort
$n : 10^6 \ k : 10^0$	1	3,9	1,2	1,3	8,1
$n : 10^6 \ k : 10^1$	6,6	2,7	1	1	7,3
$n : 10^6 \ k : 10^2$	83,1	3,2	1	1,1	6,6
$n : 10^6 \ k : 10^3$	690	2,2	1	1,1	5,7
$n : 10^6 \ k : 10^4$	$\infty$	1	5	6,4	1,9
$n : 10^6 \ k : 10^5$	$\infty$	1	$\infty$	$\infty$	1,7
$n : 10^6 \ k : 10^6$	$\infty$	1	$\infty$	$\infty$	1,8
$n : 10^7 \ k : 10^0$	1	3,4	1,1	1,1	7,4
$n : 10^7 \ k : 10^1$	8,6	2,6	1	1,1	6,7
$n : 10^7 \ k : 10^2$	82,1	2,6	1	1,1	6,8
$n : 10^7 \ k : 10^3$	$\infty$	3,1	1	1,1	6,6
$n : 10^7 \ k : 10^4$	$\infty$	1,1	1	1,2	2,6
$n : 10^7 \ k : 10^5$	$\infty$	1	$\infty$	$\infty$	2,2
$n : 10^7 \ k : 10^6$	$\infty$	1	$\infty$	$\infty$	1,2
$n : 10^7 \ k : 10^7$	$\infty$	1	$\infty$	$\infty$	1,7

## Ordenação por Contagem: Análise

- O primeiro **for** tem custo  $O(k)$ .
- O segundo **for** tem custo  $O(n)$ .
- O terceiro **for** tem custo  $O(k)$ .
- O quarto **for** tem custo  $O(n + k)$ .
- Na prática o algoritmo deve ser usado quando  $k = O(n)$ , o que leva o algoritmo a ter custo  $O(n)$ .
- De outra maneira, as complexidades de espaço e de tempo ficam proibitivas. Na seção seguinte vamos apresentar um algoritmo prático e eficiente para qualquer valor de  $k$ .

## Ordenação por Contagem

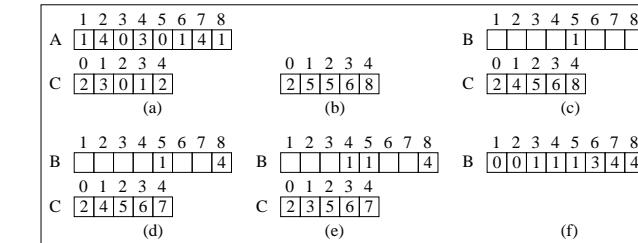
- Os arranjos auxiliares  $B$  e  $C$  devem ser declarados fora do procedimento Contagem para evitar que sejam criados a cada chamada do procedimento.
- No quarto **for**, como podem haver itens iguais no vetor  $A$ , então o valor de  $C[A[j]]$  é decrementado de 1 toda vez que um item  $A[j]$  é colocado no vetor  $B$ . Isso garante que o próximo item com valor igual a  $A[j]$ , se existir, vai ser colocado na posição imediatamente antes de  $A[j]$  no vetor  $B$ .
- O último **for** copia para  $A$  o vetor  $B$  ordenado. Essa cópia pode ser evitada colocando o vetor  $B$  como parâmetro de retorno no procedimento Contagem, como mostrado no Exercício 4.24.
- A ordenação por contagem é um método estável.

## Ordenação por Contagem

```

procedure Contagem (var A: TipoVetor; var n: Indice; var k: integer);
var i: Indice;
    { C: array[Indice] of integer; **Declarar no programa principal** }
    { B: TipoVetor; **Declarar no programa principal** }
begin
    for i := 0 to k do C[i] := 0;
    for i := 1 to n do C[A[i].Chave] := C[A[i].Chave] + 1;
    for i := 1 to k do C[i] := C[i] + C[i-1];
    for i := n downto 1 do
        begin
            B[C[A[i].Chave]] := A[i];
            C[A[i].Chave] := C[A[i].Chave] - 1;
        end;
    for i := 1 to n do A[i] := B[i];
end;

```



- A contém oito chaves de inteiros entre 0 e 4. Cada etapa mostra:
  - (a) o vetor de entrada  $A$  e o vetor auxiliar  $C$  contendo o número de itens iguais a  $i$ ,  $0 \leq i \leq 4$ ;
  - (b) o vetor  $C$  contendo o número de itens  $\leq i$ ,  $0 \leq i \leq 4$ ;
  - (c), (d), (e) os vetores auxiliares  $B$  e  $C$  após uma, duas e três iterações, considerando os itens em  $A$  da direita para a esquerda;
  - (f) o vetor auxiliar  $B$  ordenado.

## Radixsort para Inteiros

- O algoritmo de ordenação por contagem é uma excelente opção para ordenar o vetor  $A$  sobre o dígito  $i$  por ser estável e de custo  $O(n)$ .
- O vetor auxiliar  $C$  ocupa um espaço constante que depende apenas da base utilizada.
  - Por exemplo, para a base 10, o vetor  $C$  armazena valores de  $k$  entre 0 e 9, isto é, 10 posições.
- A implementação a seguir utiliza Base = 256 e o vetor  $C$  armazena valores de  $k$  entre 0 e 255 para representar os caracteres ASCII.
- Nesse caso podemos ordenar inteiros de 32 bits (4 bytes com valores entre 0 e  $2^{32}$ ) em apenas  $d = 4$  chamadas do algoritmo de ordenação por contagem.

## Radixsort para Inteiros

- Radixsort considera o dígito menos significativo primeiro e ordena os itens para aquele dígito.
- Depois repete o processo para o segundo dígito menos significativo, e assim sucessivamente.

07	01	01
33	22	07
18	⇒ 33	⇒ 07
22	07	18
01	07	22
07	18	33
↑	↑	

## Radixsort para Inteiros

Primeiro refinamento:

const

```
M = 8; { Número de bits a considerar a cada passada }
NBITS = 32; { Número de bits da Chave }
BASE :w
= 256;
RadixsortInt (A, n);
for i := 0 to (NBITS div M) - 1 do
  Ordena A sobre o dígito i menos significativo usando um algoritmo estável;
  • O programa recebe o vetor  $A$  e o tamanho  $n$  do vetor.
  • O número de bits da chave (NBITS) e o número de bits a considerar em cada passada ( $m$ ) determinam o número de passadas, que é igual a NBITS div m.
```

## Radixsort para Inteiros

```

procedure RadixsortInt (var A: TipoVetor; n: TipoIndice);
var i, Pass: integer;
begin
  for i := 0 to (NBITS div M) - 1 do
    begin
      Pass := i;
      ContagemInt (A, n, Pass);
    end;
end;

```

## Radixsort para Inteiros

- No Programa, quando qualquer posição  $i$  do vetor  $C$  contém um valor igual a  $n$  significa que todos os  $n$  números do vetor de entrada  $A$  são iguais a  $i$ .
- Isso é verificado no comando `if` logo após o segundo `for` para  $C[0]$ . Nesse caso todos os valores de  $A$  são iguais a zero no *byte* considerado como chave de ordenação e o restante do anel não precisa ser executado.
- Essa situação ocorre com frequência nos *bytes* mais significativos de um número inteiro.
- Por exemplo, para ordenar números de 32 bits que tenham valores entre 0 e 255, os três *bytes* mais significativos são iguais a zero.

## Ordenação por Contagem Alterado

```

procedure ContagemInt (var A: TipoVetor; var n: TipoIndice; var Pass: integer);
var i, j: integer; { Declarar B e C no prog principal }
  function GetBits (x: integer; k: integer; j: integer): integer;
begin GetBits := (x shr k) and not (not 0 shl j); end;
begin
  for i:=0 to BASE - 1 do C[i] := 0;
  for i:=1 to n do
    begin j := GetBits(A[i].Chave, Pass * M, M); C[j] := C[j]+1; end;
  if C[0] < n
  then begin
    for i := 1 to BASE - 1 do C[i] := C[i] + C[i-1];
    for i := n downto 1 do
      begin
        j := GetBits(A[i].Chave, Pass * M, M); B[C[j]]:=A[i]; C[j]:=C[j] - 1;
      end;
    end;
  for i := 1 to n do A[i] := B[i];
end;

```

## Radixsort para Inteiros

- O algoritmo de ordenação por contagem precisa ser alterado para ordenar sobre  $m$  bits de cada chave do vetor  $A$ .
- A função `GetBits` extrai um conjunto contíguo de  $m$  bits do número inteiro.
- Em linguagem de máquina, os bits são extraídos de números binários usando operações `and`, `shl` (*shift left*), `shr` (*shift right*), e `not` (*complementa todos os bits*).
- Por exemplo, os 2 bits menos significativos de um número  $x$  de 10 bits são extraídos movendo os bits para a direita com  $x \text{ shr } 2$  e uma operação `and` com a máscara 0000000011.

## Ordenação Externa

- A ordenação externa consiste em ordenar arquivos de tamanho maior que a memória interna disponível.
- Os métodos de ordenação externa são muito diferentes dos de ordenação interna.
- Na ordenação externa os algoritmos devem diminuir o número de acesso as unidades de memória externa.
- Nas memórias externas, os dados ficam em um arquivo seqüencial.
- Apenas um registro pode ser acessado em um dado momento. Essa é uma restrição forte se comparada com as possibilidades de acesso em um vetor.
- Logo, os métodos de ordenação interna são inadequados para ordenação externa.
- Técnicas de ordenação diferentes devem ser utilizadas.

## Radixsort para Cadeias de Caracteres

```
procedure RadixsortCar (var A: TipoVetor; n: TipoIndice);
var
  i, NCar: integer;
begin
  for i := TAMCHAVE downto 1 do
    begin
      NCar := i;
      ContagemCar (A, n, NCar);
    end;
end; { RadixsortCar }
```

## Radixsort para Cadeias de Caracteres

- O algoritmo de ordenação por contagem precisa ser alterado para ordenar sobre o caractere  $k$  da chave de cada item  $x$  do vetor  $A$ .

```
procedure ContagemCar (var A: TipoVetor; n: TipoIndice; var NCar: integer);
var i, j: integer;
  { C: array[0..BASE-1] of integer; **Declarar no prog principal** }
  { B: TipoVetor; **Declarar no prog principal** }

begin
  for i := 0 to BASE - 1 do C[i] := 0;
  for i := 1 to n do begin j:=ord(A[i].Chave[NCar]); C[j]:=C[j]+1; end;
  for i := 1 to BASE - 1 do C[i] := C[i] + C[i-1];
  for i := n downto 1 do
    begin
      j := ord (A[i].Chave[NCar]); B[C[j]] := A[i]; C[j] := C[j] - 1; end;
  for i := 1 to n do A[i] := B[i];
end; { ContagemCar }
```

## Radixsort para Inteiros: Análise

- Cada passada sobre  $n$  inteiros em ContagemInt custa  $O(n + \text{Base})$ .
- Como são necessárias  $d$  passadas, o custo total é  $O(dn + d\text{Base})$ .
- Radixsort tem custo  $O(n)$  quando  $d$  é constante e  $\text{Base} = O(n)$ .
- Se cada número cabe em uma palavra de computador, então ele pode ser tratado como um número de  $d$  dígitos na notação base  $n$ .
- Para  $A$  contendo 1 bilhão de números de 32 bits (4 dígitos na base  $2^8 = 256$ ), apenas 4 chamadas de Contagem são necessárias.
- Se considerarmos um algoritmo que utiliza o princípio da comparação de chaves, como o Quicksort, então são necessárias  $\approx \log n = 30$  operações por número (considerando que uma palavra de computador ocupa  $O(\log n)$  bits).
- Isso significa que o Radixsort é mais rápido para ordenar inteiros.
- O aspecto negativo é o espaço adicional para  $B$  e  $C$ .

## Intercalação Balanceada de Vários Caminhos

- Considere um arquivo armazenado em uma fita de entrada:

INTERCALAÇÃO BALANCEADA

- Objetivo:
  - Ordenar os 22 registros e colocá-los em uma fita de saída.
- Os registros são lidos um após o outro.
- Considere uma memória interna com capacidade para três registros.
- Considere que esteja disponível seis unidades de fita magnética.

## Ordenação Externa

- O método mais importante é o de ordenação por intercalação.
- Intercalar significa combinar dois ou mais blocos ordenados em um único bloco ordenado.
- A intercalação é utilizada como uma operação auxiliar na ordenação.
- Estratégia geral dos métodos de ordenação externa:
  1. Quebre o arquivo em blocos do tamanho da memória interna disponível.
  2. Ordene cada bloco na memória interna.
  3. Intercale os blocos ordenados, fazendo várias passadas sobre o arquivo.
  4. A cada passada são criados blocos ordenados cada vez maiores, até que todo o arquivo esteja ordenado.

## Ordenação Externa

- Os algoritmos para ordenação externa devem reduzir o número de passadas sobre o arquivo.
- Uma boa medida de complexidade de um algoritmo de ordenação por intercalação é o número de vezes que um item é lido ou escrito na memória auxiliar.
- Os bons métodos de ordenação geralmente envolvem no total menos do que dez passadas sobre o arquivo.

## Ordenação Externa

- Fatores que determinam as diferenças das técnicas de ordenação externa:
1. Custo para acessar um item é algumas ordens de grandeza maior.
  2. O custo principal na ordenação externa é relacionado a transferência de dados entre a memória interna e externa.
  3. Existem restrições severas de acesso aos dados.
  4. O desenvolvimento de métodos de ordenação externa é muito dependente do estado atual da tecnologia.
  5. A variedade de tipos de unidades de memória externa torna os métodos dependentes de vários parâmetros.
  6. Assim, apenas métodos gerais serão apresentados.

## Intercalação Balanceada de Vários Caminhos

- Quantas passadas são necessárias para ordenar um arquivo de tamanho arbitrário?
  - Seja  $n$ , o número de registros do arquivo.
  - Suponha que cada registro ocupa  $m$  palavras na memória interna.
  - A primeira etapa produz  $n/m$  blocos ordenados.
  - Seja  $P(n)$  o número de passadas para a fase de intercalação.
  - Seja  $f$  o número de fitas utilizadas em cada passada.
  - Assim:

$$P(n) = \log_f \frac{n}{m}.$$

No exemplo acima,  $n=22$ ,  $m=3$  e  $f=3$  temos:

$$P(n) = \log_3 \frac{22}{3} = 2.$$

## Intercalação Balanceada de Vários Caminhos

- Fase de intercalação - Primeira passada:
  - O primeiro registro de cada fita é lido.
  - Retire o registro contendo a menor chave.
  - Armazene-o em uma fita de saída.
  - Leia um novo registro da fita de onde o registro retirado é proveniente.
  - Ao ler o terceiro registro de um dos blocos, sua fita fica inativa.
  - A fita é reativada quando o terceiro registro das outras fitas forem lidos.

## Intercalação Balanceada de Vários Caminhos

- Fase de intercalação - Primeira passada:
  - Neste instante um bloco de nove registros ordenados foi formado na fita de saída.
  - Repita o processo para os blocos restantes.
- Resultado da primeira passada da segunda etapa:

fita 4:	<i>A A C E I L N R T</i>
fita 5:	<i>A A A B C C L N O</i>
fita 6:	<i>A A D E</i>

## Intercalação Balanceada de Vários Caminhos

- Fase de criação dos blocos ordenados:

fita 1:	<i>I N T</i>	<i>A C O</i>	<i>A D E</i>
fita 2:	<i>C E R</i>	<i>A B L</i>	<i>A</i>
fita 3:	<i>A A L</i>	<i>A C N</i>	

## Implementação por meio de Seleção por Substituição

Entra	1	2	3
E	I	N	T
R	N	E*	T
C	R	E*	T
A	T	E*	C*
L	A*	E*	C*
A	C*	E*	L*
C	E*	A	L*
A	L*	A	C
O	A	A	C
B	A	O	C
A	B	O	C

Entra	1	2	3
L	C	O	A*
A	L	O	A*
N	O	A*	A*
C	A*	N*	A*
E	A*	N*	C*
A	C*	N*	E*
D	E*	N*	A
A	N*	D	A
A	D	A	
A	D		
D			

- Primeira passada sobre o arquivo exemplo.
- Os asteriscos indicam quais chaves pertencem a blocos diferentes.

## Implementação por meio de Seleção por Substituição

- A implementação do método de intercalação balanceada pode ser realizada utilizando filas de prioridades.
- As duas fases do método podem ser implementadas de forma eficiente e elegante.
- Operações básicas para formar blocos ordenados:
  - Obter o menor dentre os registros presentes na memória interna.
  - Substituí-lo pelo próximo registro da fita de entrada.
- Estrutura ideal para implementar as operações: *heap*.
- Operação de substituição:
  - Retirar o menor item da fila de prioridades.
  - Colocar um novo item no seu lugar.
  - Reconstituir a propriedade do *heap*.

## Intercalação Balanceada de Vários Caminhos

- No exemplo foram utilizadas  $2f$  fitas para uma intercalação-de- $f$ -caminhos.
- É possível usar apenas  $f + 1$  fitas:
  - Encaminhe todos os blocos para uma única fita.
  - Redistribua estes blocos entre as fitas de onde eles foram lidos.
  - O custo envolvido é uma passada a mais em cada intercalação.
- No caso do exemplo de 22 registros, apenas quatro fitas seriam suficientes:
  - A intercalação dos blocos a partir das fitas 1, 2 e 3 seria toda dirigida para a fita 4.
  - Ao final, o segundo e o terceiro blocos ordenados de nove registros seriam transferidos de volta para as fitas 1 e 2.

## Considerações Práticas

- Técnica para obter superposição de E/S e processamento interno:
  - Utilize  $2f$  áreas de entrada e  $2f$  de saída.
  - Para cada unidade de entrada ou saída, utiliza-se duas áreas de armazenamento:
    1. Uma para uso do processador central
    2. Outra para uso do processador de entrada ou saída.
  - Para entrada, o processador central usa uma das duas áreas enquanto a unidade de entrada está preenchendo a outra área.
  - Depois a utilização das áreas é invertida entre o processador de entrada e o processador central.
  - Para saída, a mesma técnica é utilizada.

## Considerações Práticas

- As operações de entrada e saída de dados devem ser implementadas eficientemente.
- Deve-se procurar realizar a leitura, a escrita e o processamento interno dos dados de forma simultânea.
- Os computadores de maior porte possuem uma ou mais unidades independentes para processamento de entrada e saída.
- Assim, pode-se realizar processamento e operações de E/S simultaneamente.

## Implementação por meio de Seleção por Substituição

- Exemplo:

Entra	1	2	3
A	A	C	I
L	A	C	I
E	C	L	I
R	E	L	I
N	I	L	R
	L	N	R
T	N	R	
	R	T	
		T	

- Para  $f$  pequeno não é vantajoso utilizar seleção por substituição para intercalar blocos:
  - Obtém-se o menor item fazendo  $f - 1$  comparações.
- Quando  $f$  é 8 ou mais, o método é adequado:
  - Obtém-se o menor item fazendo  $\log_2 f$  comparações.

## Implementação por meio de Seleção por Substituição

- Fase de intercalação dos blocos ordenados obtidos na primeira fase:
  - Operação básica: obter o menor item dentre os ainda não retirados dos  $f$  blocos a serem intercalados.

Algoritmo:

- Monte uma fila de prioridades de tamanho  $f$ .
- A partir de cada uma das  $f$  entradas:
  - Substitua o item no topo da fila de prioridades pelo próximo item do mesmo bloco do item que está sendo substituído.
  - Imprima em outra fita o elemento substituído.

## Intercalação Polifásica

- Problema com a intercalação balanceada de vários caminhos:
  - Necessita de um grande número de fitas.
  - Faz várias leituras e escritas entre as fitas envolvidas.
  - Para uma intercalação balanceada de  $f$  caminhos são necessárias  $2f$  fitas.
  - Alternativamente, pode-se copiar o arquivo quase todo de uma única fita de saída para  $f$  fitas de entrada.
  - Isso reduz o número de fitas para  $f + 1$ .
  - Porém, há um custo de uma cópia adicional do arquivo.
- Solução:
  - **Intercalação polifásica.**

## Considerações Práticas

- Escolha da ordem de intercalação  $f$ :
  - Para fitas magnéticas:
    - \*  $f$  deve ser igual ao número de unidades de fita disponíveis menos um.
    - \* A fase de intercalação usa  $f$  fitas de entrada e uma fita de saída.
    - \* O número de fitas de entrada deve ser no mínimo dois.
  - Para discos magnéticos:
    - \* O mesmo raciocínio acima é válido.
    - \* O acesso seqüencial é mais eficiente.
  - Sedegwick (1988) sugere considerar  $f$  grande o suficiente para completar a ordenação em poucos passos.
  - Porém, a melhor escolha para  $f$  depende de vários parâmetros relacionados com o sistema de computação disponível.

## Considerações Práticas

- Solução para os problemas:
  - Técnica de previsão:
    - \* Requer a utilização de uma única área extra de armazenamento durante a intercalação.
    - \* Superpõe a entrada da próxima área que precisa ser preenchida com a parte de processamento interno do algoritmo.
    - \* É fácil saber qual área ficará vazia primeiro.
    - \* Basta olhar para o último registro de cada área.
    - \* A área cujo último registro é o menor, será a primeira a ser esvaziada.

## Considerações Práticas

- Problemas com a técnica:
  - Apenas metade da memória disponível é utilizada.
  - Isso pode levar a uma ineficiência se o número de áreas for grande.  
Ex: Intercalação-de- $f$ -caminhos para  $f$  grande.
  - Todas as  $f$  áreas de entrada em uma intercalação-de- $f$ -caminhos se esvaziando aproximadamente ao mesmo tempo.

## Intercalação Polifásica

- A implementação da intercalação polifásica é simples.
- A parte mais delicada está na distribuição inicial dos blocos ordenados entre as fitas.
- Distribuição dos blocos nas diversas etapas do exemplo:

fita 1	fita 2	fita 3	Total
3	2	0	5
1	0	2	3
0	1	1	2
1	0	0	1

## Intercalação Polifásica

- Exemplo:
  - Blocos ordenados obtidos por meio de seleção por substituição:

fita 1: INRT      ACEL      AABCLO
fita 2: AACEN    AAD
fita 3:

- Configuração após uma intercalação-de-2-caminhos das fitas 1 e 2 para a fita 3:

fita 1: AABCLO
fita 2:
fita 3: AACEINNRT    AAAACDEL

## Intercalação Polifásica

- Exemplo:
    - Depois da intercalação-de-2-caminhos das fitas 1 e 3 para a fita 2:
- |                          |
|--------------------------|
| fita 1:                  |
| fita 2: AAAABCCCEILNNORT |
| fita 3: AAAACDEL         |
- Finalmente:
    - A intercalação é realizada em muitas fases.
    - As fases não envolvem todos os blocos.
    - Nenhuma cópia direta entre fitas é realizada.

## Intercalação Polifásica

- Os blocos ordenados são distribuídos de forma desigual entre as fitas disponíveis.
- Uma fita é deixada livre.
- Em seguida, a intercalação de blocos ordenados é executada até que uma das fitas esvazie.
- Neste ponto, uma das fitas de saída troca de papel com a fita de entrada.

## Quicksort Externo

- Para o partionamento é utilizada uma área de armazenamento na memória interna.
- Tamanho da área:  $\text{TamArea} = j - i - 1$ , com  $\text{TamArea} \geq 3$ .
- Nas chamadas recursivas deve-se considerar que:
  - Primeiro deve ser ordenado o subarquivo de menor tamanho.
  - Condição para que, na média,  $O(\log n)$  subarquivos tenham o processamento adiado.
  - Subarquivos vazios ou com um único registro são ignorados.
  - Caso o arquivo de entrada  $A$  possua no máximo  $\text{TamArea}$  registros, ele é ordenado em um único passo.

## Quicksort Externo

- Seja  $R_i$ ,  $1 \leq i \leq n$ , o registro que se encontra na  $i$ -ésima posição de  $A$ .
- Algoritmo:
  - Particionar  $A$  da seguinte forma:  
 $\{R_1, \dots, R_i\} \leq R_{i+1} \leq R_{i+2} \leq \dots \leq R_{j-2} \leq R_{j-1} \leq \{R_j, \dots, R_n\}$ ,
  - chamar recursivamente o algoritmo em cada um dos subarquivos  
 $A_1 = \{R_1, \dots, R_i\}$  e  $A_2 = \{R_j, \dots, R_n\}$ .

## Quicksort Externo

- Foi proposto por Monard em 1980.
- Utiliza o paradigma de **divisão e conquista**.
- O algoritmo ordena ***in situ*** um arquivo  $A = \{R_1, \dots, R_n\}$  de  $n$  registros.
- Os registros estão armazenados consecutivamente em memória secundária de acesso randômico.
- O algoritmo utiliza somente  $O(\log n)$  unidades de memória interna e não é necessária nenhuma memória externa adicional.

## Intercalação Polifásica

### Análise:

- A análise da intercalação polifásica é complicada.
- O que se sabe é que ela é ligeiramente melhor do que a intercalação balanceada para valores pequenos de  $f$ .
- Para valores de  $f > 8$ , a intercalação balanceada pode ser mais rápida.

## Quicksort Externo: Procedimentos Auxiliares

```

procedure EscreveMax (var ArqLEs: TipoArq; R: TipoRegistro; var Es: Integer);
begin seekUpdate(ArqLEs, Es - 1);
  write(ArqLEs, R);
  Es := Es - 1;
end;

procedure EscreveMin (var ArqEi: TipoArq; R: TipoRegistro; var Ei: Integer);
begin write (ArqEi, R);
  Ei := Ei + 1;
end;

procedure RetiraMax (var Area: TipoArea; var R: TipoRegistro; var NArea: Integer );
begin RetiraUltimo (Area, R);
  NArea := ObterNumCelOcupadas (Area);
end;

procedure RetiraMin (var Area: TipoArea; var R: TipoRegistro; var NArea: Integer );
begin RetiraPrimeiro (Area, R);
  NArea := ObterNumCelOcupadas (Area);
end;

```

```

procedure LeSup (var ArqLEs: TipoArq; var UltLido: TipoRegistro;
  var Ls: Integer; var OndeLer: Boolean);
begin
  seekUpdate (ArqLEs, Ls - 1);
  read (ArqLEs, UltLido);
  Ls := Ls - 1;
  OndeLer := false;
end;

procedure LeInf (var ArqLi: TipoArq; var UltLido: TipoRegistro;
  var Li: Integer; var OndeLer: Boolean);
begin read (ArqLi, UltLido);
  Li := Li + 1; OndeLer := true;
end;

procedure InserirArea (var Area: TipoArea; var UltLido: TipoRegistro;
  var NArea: Integer );
begin {Insere UltLido de forma ordenada na Area}
  Inserelitem (UltLido, Area);
  NArea := ObterNumCelOcupadas (Area);
end;

```

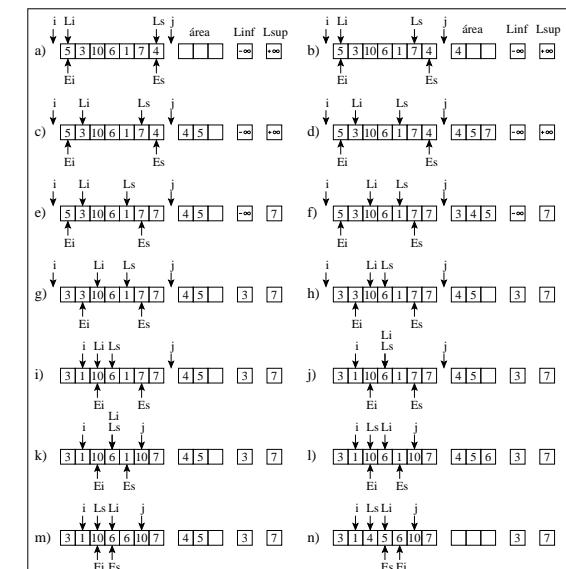
## Quicksort Externo

```

procedure QuicksortExterno (var ArqLi, ArqEi, ArqLEs: TipoArq; Esq, Dir: integer);
var i : integer; j : integer;
  Area: TipoArea; { Área de armazenamento interna}
begin
  if Dir - Esq >= 1
  then begin
    FAVazia (Area);
    Particao (ArqLi, ArqEi, ArqLEs, Area, Esq, Dir, i, j);
    if i - Esq < Dir - j
    then begin { ordene primeiro o subarquivo menor }
      QuicksortExterno (ArqLi, ArqEi, ArqLEs, Esq, i);
      QuicksortExterno (ArqLi, ArqEi, ArqLEs, j, Dir);
    end
    else begin
      QuicksortExterno (ArqLi, ArqEi, ArqLEs, j, Dir);
      QuicksortExterno (ArqLi, ArqEi, ArqLEs, Esq, i);
    end;
  end;
end; { QuicksortExterno }

```

## Quicksort Externo



## Quicksort Externo: Análise

- Seja  $n$  o número de registros a serem ordenados.
- Seja  $b$  o tamanho do bloco de leitura ou gravação do Sistema operacional.
- Melhor caso:  $O(\frac{n}{b})$ 
  - Por exemplo, ocorre quando o arquivo de entrada já está ordenado.
- Pior caso:  $O(\frac{n^2}{\text{TamArea}})$ 
  - ocorre quando um dos arquivos retornados pelo procedimento Particao tem o maior tamanho possível e o outro é vazio.
  - A medida que  $n$  cresce, a probabilidade de ocorrência do pior caso tende a zero.
- Caso Médio:  $O(\frac{n}{b} \log(\frac{n}{\text{TamArea}}))$ 
  - É o que tem a maior probabilidade de ocorrer.

## Quicksort Externo: Procedimento Particao

```

if UltLido.Chave > Lsup
then begin j := Es; EscreveMax (ArqLEs, UltLido, Es); end
else if UltLido.Chave < Linf
then begin i := Ei; EscreveMin (ArqEi, UltLido, Ei); end
else begin
    InserirArea (Area, UltLido, NRArea);
    if Ei - Esq < Dir - Es
    then begin
        RetiraMin (Area, R, NRArea);
        EscreveMin (ArqEi, R, Ei); Linf := R.Chave;
    end
    else begin
        RetiraMax (Area, R, NRArea);
        EscreveMax (ArqLEs, R, Es); Lsup := R.Chave;
    end
end;
end;
while (Ei <= Es) do
begin RetiraMin (Area, R, NRArea); EscreveMin (ArqEi, R, Ei); end
end;

```

## Quicksort Externo: Programa Teste

```

program QuicksortExterno;
{— Entra aqui o Programa 3.23 —}
type TipoRegistro = Tipoltem;
  TipoArq = file of TipoRegistro;
var ArqLEs: TipoArq; { Gerencia o Ls e o Es }
  ArqLi : TipoArq; { Gerencia o Li }
  ArqEi : TipoArq; { Gerencia o Ei }
  R : TipoRegistro;
{— Entram aqui os Programas K.4, 4.26, 4.27 e 4.28 —}
begin Assign (ArqLi,"teste.dat"); Assign (ArqEi,"teste.dat"); Assign (ArqLEs,"teste.dat");
  SeekWrite (ArqLi,0);
  R := 5; write (ArqLi, R); R := 3; write (ArqLi, R);
  R := 10; write (ArqLi, R); R := 6; write (ArqLi, R);
  R := 1; write (ArqLi, R); R := 7; write (ArqLi, R);
  R := 4; write (ArqLi, R);
  close (ArqLi); QuicksortExterno (ArqLi, ArqEi, ArqLEs, 1, 7);
  close (ArqLi); close (ArqEi); close (ArqLEs);
  SeekUpdate(ArqLi, 0);
  while not eof(ArqLi) do begin read(ArqLi, R); writeln('Registro=', R); end;
  close(ArqLi);
end.

```

## Quicksort Externo: Procedimento Particao

```

procedure Particao (var ArqLi, ArqEi, ArqLEs: TipoArq; Area: TipoArea; Esq, Dir: integer; var i, j: integer);
var Ls, Es, Li, Ei, NRArea, Linf, Lsup: integer;
  UltLido, R: TipoRegistro; OndeLer: boolean;
begin Ls := Dir; Es := Dir; Li := Esq; Ei := Esq;
  Linf := -MaxInt; Lsup := MaxInt; NRArea := 0; OndeLer := true;
  seekUpdate (ArqLi, Li - 1); seekUpdate (ArqEi, Ei - 1);
  i := Esq - 1; j := Dir + 1;
  while (Ls >= Li) do begin
    if NRArea < TamArea - 1
    then begin if OndeLer
        then LeSup (ArqLEs, UltLido, Ls, OndeLer)
        else LeInf (ArqLi, UltLido, Li, OndeLer);
        InserirArea (Area, UltLido, NRArea);
    end
    else begin if Ls = Es then LeSup (ArqLEs, UltLido, Ls, OndeLer)
        else if Li = Ei
            then LeInf (ArqLi, UltLido, Li, OndeLer)
            else if OndeLer
                then LeSup (ArqLEs, UltLido, Ls, OndeLer)
                else LeInf (ArqLi, UltLido, Li, OndeLer);
    end;
  end;
end;

```