Algoritmos e Estruturas de Dados II

Ordenação

Antonio Alfredo Ferreira Loureiro

loureiro@dcc.ufmg.br

http://www.dcc.ufmg.br/~loureiro

Sumário

- Introdução: conceitos básicos
- Ordenação Interna:
  - Seleção
  - Inserção
  - Bolha
  - Shellsort
  - Quicksort
  - Heapsort
- Ordenação parcial:
  - Seleção
  - Inserção
  - Heapsort
  - Quicksort
- Ordenação externa:
  - Intercalação balanceada de vários caminhos
  - Implementação por meio de seleção por substituição
  - Considerações práticas
  - Intercalação polifásica
  - Quicksort externo

Considerações iniciais

- Objetivos:
  - Apresentar os métodos de ordenação mais importantes sob o ponto de vista prático
  - Mostrar um conjunto amplo de algoritmos para realizar uma mesma tarefa, cada um deles com uma vantagem particular sobre os outros, dependendo da aplicação

- Cada método:
  - ilustra o uso de estruturas de dados
  - mostra como a escolha da estrutura influi nos algoritmos

Definição e objetivos da ordenação

- Ordenar corresponde ao processo de rearranjar um conjunto de objetos em uma ordem específica.
- Objetivo da ordenação:
  - facilitar a recuperação posterior de elementos do conjunto ordenado.
- Exemplos:
  - Listas telefônicas
  - Dicionários
  - Índices de livros
  - Tabelas e arquivos

Observações

- Os algoritmos trabalham sobre os registros de um arquivo.
- Apenas uma parte do registro, chamada **chave**, é utilizada para controlar a ordenação.
- Além da chave podem existir outros componentes em um registro, que não têm influência no processo de ordenar, a não ser pelo fato de que permanecem com a mesma chave.
- O tamanho dos outros componentes pode influenciar na escolha do método ou na forma de implementação de um dado método.
- A estrutura de dados registro é a indicada para representar os elementos a serem ordenados.

Notação

- Sejam os os itens $a_1, a_2, \ldots, a_n$.

- **Ordenar** consiste em **permutar** estes itens em uma ordem

$$a_{k_1}, a_{k_2}, \ldots, a_{k_n}$$

  tal que, dada uma função de ordenação f, tem-se a seguinte relação:

$$f(a_{k_1}) \leq f(a_{k_2}) \leq \ldots \leq f(a_{k_n})$$

- Função de ordenação é definida sobre o campo chave.

Notação

- Qualquer tipo de campo chave, sobre o qual exista uma relação de ordem total $<$, para uma dada função de ordenação, pode ser utilizado.

- A relação $<$ deve satisfazer as condições:
  - Apenas um de $a < b$, $a = b$, $a > b$ é verdade
  - Se $a < b$ e $b < c$ então $a < c$

- A estrutura registro é indicada para representar os itens a_i.

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

Observações

- A escolha do tipo da chave como inteiro é arbitrária.
- Qualquer tipo sobre o qual exista uma regra de ordenação bem definida pode ser utilizado.
  - Tipos usuais são o inteiro e seqüência de caracteres.
- Outros componentes representam dados relevantes sobre o item.
- Um método de ordenação é dito estável, se a ordem relativa dos itens com chaves iguais mantém-se inalterada pelo processo de ordenação.
- Exemplo:
  - Se uma lista alfabética de nomes de funcionários de uma empresa é ordenada pelo campo salário, então um método estável produz uma lista em que os funcionários com mesmo salário aparecem em ordem alfabética.

Classificação dos métodos de ordenação: Ordenação interna

- Número de registros a serem ordenados é pequeno o bastante para que todo o processo se desenvolva na memória interna (principal)
  - Qualquer registro pode ser acessado em tempo $O(1)$
- Organiza os dados na forma de vetores, onde cada dado é “visível”
- Exemplo:

$$\boxed{12}\ \boxed{7}\ \boxed{10}\ \boxed{5}\ \boxed{4}\ \boxed{15}\ \boxed{9}\ \boxed{8}\ \ldots$$

Ordenação interna

- Requisito predominante:
  - Uso econômico da memória disponível.
  - ➜ Logo, permutação dos itens “in situ”.
- Outro requisito importante:
  - Economia do tempo de execução.
- Medidas de complexidade relevantes:
  - $C(n)$: número de comparações entre chaves
  - $M(n)$: número de movimentos ou trocas de registros
- Observação:
  - A quantidade extra de memória auxiliar utilizada pelo algoritmo é também um aspecto importante.
  - Os métodos que utilizam a estrutura vetor e que executam a permutação dos itens no próprio vetor, exceto para a utilização de uma pequena tabela ou pilha, são os preferidos.

Classificação dos métodos de ordenação: Ordenação externa

- Número de registros a ser ordenado é maior do que o número que cabe na memória interna.
- Registros são armazenados em um arquivo em disco ou fita.
- Registros são acessados seqüencialmente ou em grandes blocos.
  - Apenas o dado de cima é visível.

Métodos para ordenação interna

- Métodos simples ou diretos:
  - Programas são pequenos e fáceis de entender.
  - Ilustram com simplicidade os princípios de ordenação.
  - Pequena quantidade de dados.
  - Algoritmo é executado apenas uma vez ou algumas vezes.
  - Complexidade: $C(n) = O(n^2)$

- Métodos Eficientes:
  - Requerem menos comparações
  - São mais complexos nos detalhes
  - Quantidade maior de dados.
  - Algoritmo é executado várias vezes.
  - Complexidade: $C(n) = O(n \log n)$

➜ Eventualmente, pode ser necessária uma combinação dos dois métodos no caso de uma implementação eficiente

Tipos utilizados na implementação dos algoritmos

- Na implementação dos algoritmos usaremos:

```
type Indice = 0..N;
     Item   = record
                Chave : integer;
                {outros componentes}
              end;
     Vetor  = array [Indice] of Item;

var A : Vetor;
```

- O tipo Vetor é do tipo estruturado arranjo, composto por uma repetição do tipo de dados Item.

- O índice do vetor vai de 0 até n, para poder armazenar chaves especiais chamadas sentinelas.

Métodos de ordenação tratados no curso

- Métodos simples:
  - Ordenação por seleção.
  - Ordenação por inserção.
  - Ordenação por permutação (Método da Bolha).

- Métodos eficientes:
  - Ordenação por inserção através de incrementos decrescentes (Shellsort).
  - Ordenação de árvores (Heapsort).
  - Ordenação por particionamento (Mergesort e Quicksort).

Métodos de ordenação que utilizam o princípio de distribuição

- Métodos que não fazem ordenação baseados em comparação de chaves.
- Exemplo:
  - Considere o problema de ordenar um baralho com 52 cartas não ordenadas.
  - Suponha que ordenar o baralho implica em colocar as cartas de acordo com a ordem.

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

e

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

Métodos de ordenação que utilizam o princípio de distribuição

- Para ordenar por distribuição, basta seguir os passos abaixo:
  1. Distribuir as cartas abertas em 13 montes, colocando em cada monte todos os ases, todos os dois, todos os três, . . . , todos os reis.
  2. Colete os montes na ordem acima (ás no fundo, depois os dois, etc), até o rei ficar no topo.
  3. Distribua novamente as cartas abertas em 4 montes, colocando em cada monte todas as cartas de paus, todas as cartas de ouros, todas as cartas de copas e todas as cartas de espadas.
  4. Colete os montes na ordem indicada acima (paus, ouros, copas e espadas).

Métodos de ordenação que utilizam o princípio de distribuição

- Métodos baseados no princípio de distribuição são também conhecidos como ordenação digital, *radixsort* ou *bucketsort*.
- Exemplos:
  - Classificadoras de cartões perfurados utilizam o princípio da distribuição para ordenar uma massa de cartões.
  - Carteiro no momento de distribuir as correspondências por rua ou bairro
- Dificuldades de implementar este método:
  - Está relacionada com o problema de lidar com cada monte.
  - Se para cada monte é reservada uma área, então a demanda por memória extra pode se tornar proibitiva.
- O custo para ordenar um arquivo com n elementos é da ordem de $O(n)$, pois cada elemento é manipulado algumas vezes.

Ordenação por Seleção

- Um dos algoritmos mais simples.
- Princípio:
  1. Seleciona o item com a menor chave;
  2. Troca este item com o item A[1];
  3. Repita a operação com os $n-1$ itens restantes, depois com os $n-2$ restantes, etc.
- Exemplo (chaves sublinhadas foram trocadas de posição):

	1	*2*	*3*	*4*	*5*	*6*
Chaves iniciais	44	12	55	42	94	18
$i = 1$	<u>12</u>	<u>44</u>	55	42	94	18
$i = 2$	12	<u>18</u>	55	42	94	<u>44</u>
$i = 3$	12	18	<u>42</u>	<u>55</u>	94	44
$i = 4$	12	18	42	<u>44</u>	94	<u>55</u>
$i = 5$	12	18	42	44	<u>55</u>	<u>94</u>

Algoritmo Seleção

```
procedure Selecao (var A : Vetor);
var i, j, Min : Indice;
    T         : Item;
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;
    T := A[Min];
    A[Min] := A[i];
    A[i] := T;
  end;
end;  {Selecao}
```

Análise do algoritmo Seleção

- O comando de atribuição `Min := j`
  - É executado em média cerca de $n \log n$ vezes (Knuth, 1973).
  - Este valor depende do número de vezes que c_j é menor do que todas as chaves anteriores $c_1, c_2, \ldots, c_{j-1}$, quando estamos percorrendo as chaves $c_1, c_2, \ldots, c_n$.

- Análise:
  - $C(n) = \frac{n^2}{2} - \frac{n}{2} = O(n^2)$.
  - $M(n) = 3(n - 1) = O(n)$.

- $C(n)$ e $M(n)$ independem da ordem inicial das chaves.

Observações sobre o algoritmo de Seleção

- Vantagens:
  - ↑ É um dos métodos mais simples de ordenação existentes.
  - ↑ Para arquivos com registros muito grandes e chaves pequenas, este deve ser o método a ser utilizado.
  - ↑ Com chaves do tamanho de uma palavra, este método torna-se bastante interessante para arquivos pequenos.

- Desvantagens;
  - ↓ O fato do arquivo já estar ordenado não ajuda em nada pois o custo continua quadrático.
  - ↓ O algoritmo não é estável, pois ele nem sempre deixa os registros com chaves iguais na mesma posição relativa.

Ordenação por Inserção

- Método preferido dos jogadores de cartas.

- Princípio:
  - Em cada passo, a partir de $i = 2$, o i-ésimo elemento da seqüência fonte é apanhado e transferido para a seqüência destino, sendo inserido no seu lugar apropriado.

- Observação: características opostas à ordenação por seleção.

Ordenação por Inserção

- Exemplo:

	1	*2*	*3*	*4*	*5*	*6*
Chaves iniciais	44	12	55	42	94	18
$i = 2$	$\underline{12}$	$\underline{44}$	55	42	94	18
$i = 3$	$\underline{12}$	$\underline{44}$	$\underline{55}$	42	94	18
$i = 4$	$\underline{12}$	$\underline{42}$	$\underline{44}$	$\underline{55}$	94	18
$i = 5$	$\underline{12}$	$\underline{42}$	$\underline{44}$	$\underline{55}$	$\underline{94}$	18
$i = 6$	$\underline{12}$	$\underline{18}$	$\underline{42}$	$\underline{44}$	$\underline{55}$	$\underline{94}$

- As chaves sublinhadas representam a seqüência destino.

Ordenação por Inserção

- A colocação do item no seu lugar apropriado na seqüência destino é realizada movendo-se itens com chaves maiores para a direita e então inserindo o item na posição deixada vazia.
- Neste processo de alternar comparações e movimentos de registros existem duas condições distintas que podem causar a terminação do processo:
  - Um item com chave menor que o item em consideração é encontrado.
  - O final da seqüência destino é atingido à esquerda.
- A melhor solução para a situação de um anel com duas condições de terminação é a utilização de um registro sentinela:
  - Na posição zero do vetor colocamos o próprio registro em consideração.
  - Para isso, o índice do vetor tem que ser definido para a seqüência $0..n$.

Algoritmo de Inserção

```
procedure Insercao (var A : Vetor);
var i, j : Indice;
    T    : Item;
begin
  for i:=2 to n do
  begin
    T    := A[i];
    j    := i-1;
    A[0] := T;   {Sentinela à esquerda}
    while T.Chave < A[j].Chave do
    begin
      A[j+1] := A[j];
      j := j - 1;
    end;
    A[j+1] := T;
  end;
end;  {Insercao}
```

Análise de Inserção: Comparações

Anel mais interno (`while`): i-ésimo deslocamento:

Melhor caso: $C_i = 1$

Pior caso: $C_i = i$

Caso médio: $C_i = \frac{1}{i} \sum_{j=1}^{i} j = \frac{1}{i}(\frac{i(i+1)}{2}) = \frac{i+1}{2}$

Análise de Inserção: Comparações

- Considerações:
  - $C(n) = \sum_{i=2}^{n} C_i$.
  - Vamos assumir que para o caso médio todas as permutações de n são igualmente prováveis.

- Cada caso é dado por:

  Melhor caso: $\sum_{i=2}^{n} 1 = n - 1$

  Pior caso: $\sum_{i=2}^{n} i = \sum_{i=1}^{n} i - 1 = \frac{n(n+1)}{2} - 1 = \frac{n^2}{2} + \frac{n}{2} - 1$

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

Análise de Inserção: Movimentações

O número de movimentações na i-ésima iteração é:

$$M_i = C_i - 1 + 3 = C_i + 2 \quad \text{(incluindo a sentinela)}$$

Temos que $M(n) = \sum_{i=2}^{n} M_i$, onde cada caso é dado por:

Melhor caso: $\sum_{i=2}^{n}(1+2) = \sum_{i=2}^{n} 3 = 3(n-1) = 3n-3$

Pior caso: $\sum_{i=2}^{n}(i+2) = \sum_{i=2}^{n} i + \sum_{i=2}^{n} 2 =$

$\frac{n(n+1)}{2} - 1 + 2(n-1) = \frac{n(n+1)}{2} + 2n - 3 = \frac{n^2}{2} + \frac{5n}{2} - 3$

Caso médio: $\sum_{i=2}^{i}(\frac{i+1}{2} + 2) = \sum_{i=2}^{n} \frac{i+1}{2} + \sum_{i=2}^{n} 2 = \frac{n^2}{4} + \frac{11n}{4} - 1$

Inserção: Pior caso

	0	*1*	*2*	*3*	*4*	*5*	*6*	# de comp.
		6	5	4	3	2	1	
$i = 2$	5	5	6	4	3	2	1	2
$i = 3$	4	4	5	6	3	2	1	3
$i = 4$	3	3	4	5	6	2	1	4
$i = 5$	2	2	3	4	5	6	1	5
$i = 6$	1	2	3	4	5	6	1	6
								20 (total)

$$C(n) = \frac{n^2}{2} + \frac{n}{2} - 1 = \frac{36}{2} + \frac{6}{2} - 1 = 18 + 3 - 1 = 20$$

$$M(n) = \frac{n^2}{2} + \frac{5n}{2} - 3 = \frac{36}{2} + \frac{30}{2} - 3 = 18 + 15 - 3 = 30$$

Observações sobre o algoritmo de 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, o que indica um comportamento natural para o algoritmo.
- Para arquivos já ordenados o algoritmo descobre, a um custo $O(n)$, que cada item já está no seu lugar.
- Utilização:
  - O método de inserção é o método a ser utilizado quando o arquivo está "quase" ordenado.
  - É também um bom método quando se deseja adicionar poucos itens a um arquivo já ordenado e depois obter um outro arquivo ordenado, com custo linear.
- O método é estável pois deixa os registros com chaves iguais na mesma posição relativa.

Inserção $\times$ Seleção

- Arquivos já ordenados:
  - Inserção: algoritmo descobre imediatamente que cada item já está no seu lugar (custo linear).
  - Seleção: ordem no arquivo não ajuda (custo quadrático).
- Adicionar alguns itens a um arquivo já ordenado:
  - Método da inserção é o método a ser usado em arquivos “quase ordenados”.

Inserção $\times$ Seleção

- Comparações:
  - Inserção tem um número médio de comparações que é aproximadamente a metade da Seleção
- Movimentações:
  - Seleção tem um número médio de comparações que cresce linearmente com n, enquanto que a média de movimentações na Inserção cresce com o quadrado de n.

Comentários sobre movimentações de registros

- Para registros grandes com chaves pequenas, o grande número de movimentações na Inserção é bastante prejudicado.

- No outro extremo, registros contendo apenas a chave, temos como objetivo reduzir o número de comparações, principalmente se a chave for grande.
  - Nos dois casos, o tempo é da ordem de n^2, para n suficientemente grande.

- Uma forma intuitiva de diminuir o número médio de comparações na Inserção é adotar uma pesquisa binária na parte já ordenada do arquivo.
  - Isto faz com que o número médio de comparações em um sub-arquivo de k posições seja da ordem de $\log_2 k$, que é uma redução considerável.
  - Isto resolve apenas metade do problema, pois a pesquisa binária só diminui o número de comparações, já que uma inserção continua com o mesmo custo.

Método da Bolha

- Princípio:
  - Chaves na posição 1 e 2 são comparadas e trocadas se estiverem fora de ordem.
  - Processo é repetido com as chaves 2 e 3, até $n-1$ e n.
- Se desenharmos o vetor de armazenamento verticalmente, com $A[n]$ em cima e $A[1]$ embaixo, durante um passo do algoritmo, cada registro “sobe” até encontrar outro com chave maior, que por sua vez sobe até encontrar outro maior ainda, etc, com um movimento semelhante a uma bolha subindo em um tubo de ensaio.
- A cada passo, podemos limitar o procedimento à posição do vetor que vai de 1 até a posição onde ocorreu a última troca no passo anterior.

Algoritmo da Bolha (*Bubblesort*)

```
var
 Lsup, Bolha, j : integer;
 Aux            : Célula

begin
  Lsup := n;
  repeat
    Bolha := 0;
    for j := 1 to Lsup - 1 do
      if A[j].Chave > A[j+1].Chave
      then begin
             Aux    := A[j];
             A[j]   := A[j+1];
             A[j+1] := Aux;
             Bolha  := j;
           end:
    Lsup := Bolha
  until Lsup <= 1
end;
```

Exemplo do algoritmo da Bolha

16	14	15	15	15	15	15	15	15	15	15	15	15	15
15	4	14	14	14	14	14	14	14	14	14	14	14	14
14	15	4	13	13	13	13	13	13	13	13	13	13	13
13	0	13	4	12	12	12	12	12	12	12	12	12	12
12	2	0	12	4	11	11	11	11	11	11	11	11	11
11	10	2	0	11	4	10	10	10	10	10	10	10	10
10	3	10	2	0	10	4	9	9	9	9	9	9	9
9	12	3	10	2	0	9	4	8	8	8	8	8	8
8	6	12	3	10	2	0	8	4	7	7	7	7	7
7	8	6	11	3	9	2	0	7	4	6	6	6	6
6	13	8	6	9	3	8	2	0	6	4	5	5	5
5	11	11	8	6	8	3	7	2	0	5	4	4	4
4	7	9	9	8	6	7	3	6	2	0	3	3	3
3	1	7	7	7	7	6	6	3	5	2	0	2	2
2	5	1	5	5	5	5	5	5	3	3	2	0	1
1	9	5	1	1	1	1	1	1	1	1	1	1	0
`Lsup`		15	13	12	11	10	9	8	7	6	5	2	1
Passos		*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*

Comentários sobre o método da Bolha

- “Parece” com o algoritmo de Seleção.
- Este método faz tantas trocas que o tornam o mais ineficiente de todos os métodos simples ou diretos.
- Melhor caso:
  - Ocorre quando o arquivo está completamente ordenado, fazendo $n - 1$ comparações e nenhuma troca, em apenas um passo.
- Pior caso:
  - Ocorre quando o arquivo está em ordem reversa, ou seja, quando o k-ésimo passo faz $n - k$ comparações e trocas, sendo necessário $n - 1$ passos.
- Quanto “mais ordenado” estiver o arquivo melhor é a atuação do método.
  - No entanto, um arquivo completamente ordenado, com exceção da menor Chave que está na última posição fará o mesmo número de comparações do pior caso.

Análise do caso médio

- Comparações:

$$C(n) = \frac{n^2}{2} - \frac{3n}{4}$$

- Movimentos:

$$M(n) = \frac{3n^2}{4} - \frac{3n}{4}$$

Comentários sobre o método da Bolha

- Método lento:
  - Só compara posições adjacentes.
- Cada passo aproveita muito pouco do que foi “aprendido”sobre o arquivo no passo anterior.
- Comparações redundantes em excesso, devido à linearidade dos algoritmos, e a uma seqüência fixa de comparações.

Comentários sobre o método da Bolha

- Método lento:
  - Só compara posições adjacentes
- Cada passo aproveita muito pouco do que foi “aprendido”sobre o arquivo no passo anterior
- Comparações redundantes em excesso, devido à linearidade dos algoritmos, e a uma seqüência fixa de comparações

Shellsort

- Referência: David L. Shell. A High-speed Sorting Procedure. Communications of the ACM, 2(7):30–32, 1959.

- O método da inserção troca itens adjacentes quando está procurando o ponto de inserção na seqüência destino
  - No primeiro passo se o menor item estiver mais à direita no vetor são necessárias $n - 1$ comparações para achar o seu ponto de inserção.

- Shellsort permite troca de itens distantes uns dos outros
  - Uma extensão do algoritmo de ordenação por inserção

- Shellsort contorna este problema permitindo trocas de registros que estão distantes um do outro
  - Itens que estão separados h posições são rearranjados de tal forma que todo h-ésimo item leva a uma seqüência ordenada.
  - Ordenação por inserção através de incrementos decrescentes.

Shellsort

- Exemplo:

	1	*2*	*3*	*4*	*5*	*6*	
	44	94	55	42	12	18	
$h = 4$	12	18	55	42	44	94	(1↔5, 2↔6)
$h = 2$	12	18	44	42	55	94	(1:3, 2:4, 3↔5, . . .)
$h = 1$	12	18	42	44	55	94	(1:2, 2:3, . . .)

- Comentários:
  - Para $h = 4$, as posições 1 e 5, e 2 e 6 são comparadas e trocadas.
  - Para $h = 2$, as posições 1 e 3, 2 e 4, etc, são comparadas e trocadas se estiverem fora de ordem.
  - Para $h = 1$, corresponde ao algoritmo de inserção, mas nenhum item tem que mover muito!

Shellsort: Seqüências para h

- Referência: Donald E. Knuth. The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd ed. Reading, MA: Addison-Wesley, pp. 83-95, 1998.

$$\begin{array}{ll} h_i = 3h_{i-1} + 1 & \text{para } i > 1 \\ h_1 = 1 & \text{para } i = 1 \end{array}$$

  A seqüência $h = 1, 4, 13, 40, 121, 363, 1093, \ldots$, obtida empiricamente, é uma boa seqüência.

- Robert Sedgewick propôs a seqüência

$$\begin{array}{ll} h_i = 4^i + 3 \cdot 2^{i-1} + 1 & \text{para } i \geq 1 \\ h_0 = 1 & \end{array}$$

  A seqüência $h = 1, 8, 23, 77, 281, 1073, 4193, 16577, \ldots$, segundo Sedgewick, é mais rápida que a proposta por Knuth de 20 a 30%.

Algoritmo Shellsort

```
procedure ShellSort (var A : Vetor);
label 999;
var i, j, h : integer;
    T       : Item;
begin
  h := 1;
  repeat h := 3*h + 1 until h >= n;     {Seqüência proposta por Knuth}
  repeat
    h := h div 3;
    for i := h+1 to n do
    begin
      T := A[i];   j := i;
      while A[j-h].Chave > T.Chave do
      begin
        A[j]  := A[j-h];
        j     := j - h;
        if j <= h then goto 999;
      end;
999:
      A[j]  := T;
    end;
  until h = 1;
end;   {Shellsort}
```

Shellsort: Ordem decrescente

	1	*2*	*3*	*4*	*5*	*6*	C_i	M_i	
	6	5	4	3	2	1			
$h = 4$	2	1	4	3	6	5	2	6	
$h = 2$	2	1	4	3	6	5	4	0	
$h = 1$	1	2	4	3	6	5	1	3	$(i = 2)$
$h = 1$	1	2	4	3	6	5	1	0	$(i = 3)$
$h = 1$	1	2	3	4	6	5	2	3	$(i = 4)$
$h = 1$	1	2	3	4	6	5	1	0	$(i = 5)$
$h = 1$	1	2	3	4	5	6	2	3	$(i = 6)$
							13	15	(Total)

Comentários sobre o Shellsort

- A implementação do Shellsort não utiliza registros sentinelas
  - Seriam necessários h registros sentinelas, uma para cada h-ordenação.
- Porque o Shellsort é mais eficiente?
  - A razão da eficiência do algoritmo ainda não é conhecida.
  - Sabe-se que cada incremento não deve ser múltiplo do anterior.
  - Várias seqüências para h foram experimentadas.
  - Uma que funciona bem (verificação empírica):

    $h_i = 3h_{i-1} + 1$

    $h_1 = 1$

    $h = 1, 4, 13, 40, 121, \ldots$
- Conjecturas referente ao número de comparações para a seqüência de Knuth:
  - Conjectura 1: $C(n) = O(n(\log n)^2)$
  - Conjectura 2: $C(n) = O(n^{1.25})$

Shellsort

- Vantagens:
  - Shellsort é uma ótima opção para arquivos de tamanho moderado ($\pm$ 10000 itens).
  - Sua implementação é simples e requer uma quantidade de código pequena.
  - Implementação simples.

- Desvantagens:
  - O tempo de execução do algoritmo é sensível à ordem inicial do arquivo.
  - O método não é **estável**.

Quicksort

- História:
  - Proposto por C.A.R. Hoare (1959–1960) quando era um *British Council Visiting Student* na Universidade de Moscou.
  - C.A.R. Hoare. Algorithm 63: Partition, Communications of the ACM, 4(7):321, 1961.
  - C.A.R. Hoare. Quicksort, The Computer Journal, 5:10–15, 1962.
- Quicksort é o algoritmo de ordenação interna mais rápido que se conhece para uma ampla variedade de situações.
- Provavelmente é mais utilizado do que qualquer outro algoritmo.
- Idéia básica:
  - Partir o problema de ordenar um conjunto com n itens em dois problemas menores.
  - Ordenar independentemente os problemas menores.
  - Combinar os resultados para produzir a solução do problema maior (neste caso, a combinação é trivial).

Quicksort: Idéia básica

- Fazer uma escolha arbitrária de um item x do vetor chamado pivô.
- Rearranjar o vetor $A[Esq..Dir]$ de acordo com a seguinte regra:
  - Elementos com chaves menores ou iguais a x vão para o lado esquerdo.
  - Elementos com chaves maiores ou iguais a x vão para o lado direito.
  - ➜ Note que uma vez rearranjados os elementos da partição, eles não mudam mais de lado com relação ao elemento x.
- Aplicar a cada uma das duas partições geradas os dois passos anteriores.

➜ Parte delicada do método é o processo de partição.

Quicksort

```
Quicksort(int A[],
          int Esq,
          int Dir)
{
  int i;

  if (Dir > Esq)                    /* Apontadores se cruzaram? */
    {
      i = Partition(Esq, Dir)       /* Particiona de acordo com o pivô */
      Quicksort(A, Esq, i-1);       /* Ordena sub-vetor da esquerda */
      Quicksort(A, i+1, Dir);       /* Ordena sub-vetor da direita */
    }
}
```

Quicksort

- Os parâmetros *Esq* e *Dir* definem os limites dos sub-vetores a serem ordenados

- Chamada inicial:
  - `Quicksort(A, 1, n)`

- O procedimento *Partition* é o ponto central do método, que deve rearranjar o vetor A de tal forma que o procedimento *Quicksort* possa ser chamado recursivamente.

Procedimento *Partition*

- Rearranja o vetor A[*Esq..Dir*] de tal forma que:
  - O pivô (item x) vai para seu lugar definitivo $A[i]$, $1 \leq i \leq n$.
  - Os itens A[*Esq*], A[*Esq*$+1$], $\ldots$, $A[i-1]$ são menores ou iguais a $A[i]$.
  - Os itens $A[i+1]$, $A[i+2]$, $\ldots$, A[*Dir*] são maiores ou iguais a $A[i]$.

- Comentários:
  - Note que ao final da partição não foi feita uma ordenação nos itens que ficaram em cada um dos dois sub-vetores, mas sim o rearranjo explicado acima.
  - O processo de ordenação continua aplicando o mesmo princípio a cada um dos dois sub-vetores resultantes, ou seja, A[*Esq*..$i-1$] e $A[i+1$..*Dir*].

Procedimento *Partition*

- Escolher pivô (Sedgewick):
  - Escolha o item A[*Dir*] (x) do vetor que irá para sua posição final.
  - Observe que este item não está sendo retirado do vetor.

- Partição:
  - Percorra o vetor a partir da esquerda (*Esq*) até encontrar um item $A[i] > x$; da mesma forma percorra o vetor a partir da direita (*Dir*) até encontrar um item $A[j] < x$.
  - Como os dois itens $A[i]$ e $A[j]$ estão fora de lugar no vetor final, eles devem ser trocados.
  - O processo irá parar quando os elementos também são iguais a x (melhora o algoritmo), apesar de parecer que estão sendo feitas trocas desnecessárias.

- Continue o processo até que os apontadores i e j se cruzem em algum ponto do vetor.
  - Neste momento, deve-se trocar o elemento A[*Dir*] com o mais à esquerda do sub-vetor da direita.

Exemplo da partição do vetor

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E

- *Esq* = 1 e *Dir* = 15.
- Escolha do pivô (item x):
  - Item x = A[15] = E.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
$\uparrow$														$\uparrow$
i														j
														$\uparrow$
														Pivô

Exemplo da partição do vetor

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	S									A	M	P	L	E
	↑									↑				↑
	i									j				Pivô

- A varredura a partir da posição 1 pára no item S (S > E) e a varredura a partir da posição 15 pára no item A (A < E), sendo os dois itens trocados.
- Como o vetor ainda não foi todo rearranjado (i e j se cruzarem) o processo deve continuar.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	A									S	M	P	L	E
	↑									↑				↑
	i									j				Pivô

Exemplo da partição do vetor

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	A	O						E	X	S	M	P	L	E
		↑						↑						↑
		i						j						Pivô

- A varredura a partir da posição 2 pára no item O (O > E) e a varredura a partir da posição 11 pára no item E (x = E), sendo os dois itens trocados.
- Como o vetor ainda não foi todo rearranjado (i e j se cruzarem) o processo deve continuar.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	A	E						O	X	S	M	P	L	E
		↑						↑						↑
		i						j						Pivô

Exemplo da partição do vetor

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	A	E	R	T	I	N	G	O	X	S	M	P	L	E
		$\uparrow$	$\uparrow$											$\uparrow$
		j	i											Pivô

- A varredura a partir da posição 3 pára no item R (R > E) e a varredura a partir da posição 9 pára no item E (x = E), sendo que os apontadores se cruzam.
- Pivô deve ser trocado com o R (item mais à esquerda do sub-vetor da direita).

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	S	O	R	T	I	N	G	E	X	A	M	P	L	E
A	A	E	E	T	I	N	G	O	X	S	M	P	L	R

Quicksort: Procedimento Partição

```
Quicksort(int A[], int Esq, int Dir) {
int x,                                        /* Pivô */
    i, j,                                     /* Apontadores para o sub-vetor */
    t;                                        /* Variável auxiliar para troca */

if (Dir > Esq) {                              /* Apontadores se cruzaram? */
   x = A[Dir];                                /* Define o pivô */
   i = Esq - 1;                               /* Inicializa apontador da esq */
   j = Dir;                                   /* Inicializa apontador da dir */
   for (;;) {                                 /* Faz a varredura no vetor */
     while (a[++i] < x);                      /* Percorre a partir da esquerda */
     while (a[--j] > x);                      /* Percorre a partir da direita */
     if (i >= j) break;                       /* Apontadores se cruzaram? */

     t = A[i]; A[i] = A[j]; A[j] = t;         /* Faz a troca entre os elementos */
   }

   t = A[i]; A[i] = A[j]; A[j] = t;           /* Coloca o pivô na posição final */

   Quicksort(A, Esq, i-1);                    /* Ordena sub-vetor da esquerda */
   Quicksort(A, i+1, Dir);                    /* Ordena sub-vetor da direita */
}
}
```

Quicksort: Seqüência de passos recursivos

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	A	E	**E**	T	I	N	G	O	X	S	M	P	L	R
A	A	**E**												
A	A													
				L	I	N	G	O	P	M	**R**	X	T	S
				L	I	G	**M**	O	P	N				
				G	I	L								
					I	L								
								N	P	O				
									O	P				
												S	T	X
													T	**X**
A	A	E	E	G	I	L	M	N	O	P	R	S	T	X

Análise do Quicksort
Considerações iniciais

- Seja $C(n)$ a função que conta o número de comparações.
- 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 *Partition* seja chamado recursivamente n vezes, eliminando apenas um item em cada chamada.
  - Neste caso, se uma partição tem p elementos, então é gerada uma outra partição com $p - 1$ elementos e o pivô vai para a sua posição final.
  - Note que neste caso não é gerada uma segunda partição, ou seja, temos um caso degenerado.
- O pior caso pode ser evitado empregando pequenas modificações no algoritmo:
  - Escolher três elementos ao acaso e pegar o do meio (Mediana de 3).
  - Escolher k elementos ao acaso, ordenar os k elementos e obter o $(\frac{k+1}{2})$-ésimo elemento.

➜ Claro, obter o elemento médio custa, mas compensa!

Outra melhoria para o Quicksort? Partições pequenas

- Utilizar um método simples de ordenação quando o número de elementos numa partição for pequeno.

- Existem algoritmos de ordenação simples, com custo $O(n^2)$ no pior caso, que são mais rápidos que o Quicksort para valores pequenos de n.

- Para partições entre 5 e 20 elementos usar um método $O(n^2)$:
  - Knuth sugere 9.

Outra melhoria para o Quicksort? Trocar espaço por tempo

- Criar um vetor de apontadores para os registros a ordenar.
- Fazemos as comparações entre os elementos apontados, mas não movemos os registros – movemos apenas os apontadores da mesma forma que o Quicksort move registros.
- Ao final os apontadores (lidos da esquerda para a direita) apontam para os registros na ordem desejada.

Qual é o ganho neste caso?

➜ Fazemos apenas n trocas ao invés de $O(n \log n)$, o que faz enorme diferença se os registros forem grandes.

Análise do Quicksort
Melhor caso

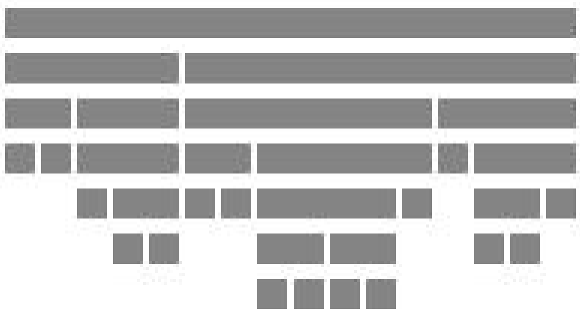
- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.

$$C(n) = 2C(n/2) + n = n \log n - n + 1 = O(n \log n)$$

Análise do Quicksort
Caso médio

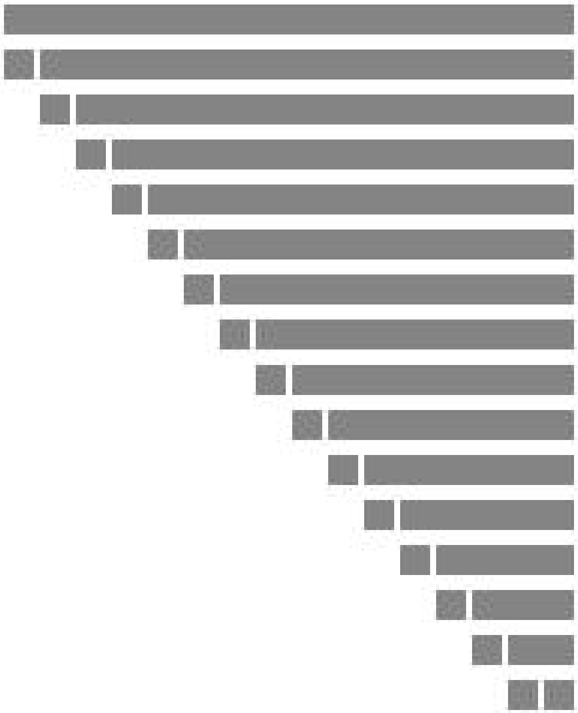
- De acordo com Sedgewick e Flajolet (1996, p. 17), o número de comparações é aproximadamente:

$$C(n) \approx 1,386 n \log n - 0,846 n = O(n \log n).$$



Análise do Quicksort
Pior caso

- Em cada chamada o pior pivô possível é selecionado!
- Por exemplo, o maior elemento de cada sub-vetor sendo ordenado.



Análise do Quicksort
Pior caso

- Seqüência de partições de um conjunto já ordenado!

$$
\begin{aligned}
T(1) &= 2 \\
T(n) &= T(n-1) + n + 2 \\
T(n) &= \sum_{i=2}^{n} i + n + 2 = \frac{n(n+1)}{2} - 1 + n + 2 \\
&= \frac{n^2}{2} + \frac{3n}{2} + 1 \\
T(n) &= O(n^2)
\end{aligned}
$$

Quicksort e a técnica “divisão-e-conquista”

- A técnica divisão-e-conquista consiste basicamente de três passos:
  1. Divide.
  2. Conquista.
  3. Combina.

- No caso do Quicksort, o passo 2 é executado antes do passo 1, ou seja, poderia ser considerado um método de “conquista-e-divisão”.

Quicksort e a técnica "divisão-e-conquista"

Passos do Quicksort considerando o vetor A[*Esq*..*Dir*]:

1. Conquista (rearranja) vetor A[*Esq*..*Dir*] de tal forma que o pivô vai para sua posição final $A[i]$.
   - Observe que a partir deste ponto nenhum elemento à esquerda do pivô é trocado com outro elemento à direita do pivô e vice-versa.

2. Divide A[*Esq*..*Dir*] em dois sub-vetores (partições) A[*Esq*..$i - 1$] e $A[i + 1$..*Dir*].
   - O passo anterior é novamente aplicado a cada partição.

3. Combina não faz nada já que o vetor A, ao final dos dois passos anteriores, está ordenado.

Quicksort: Comentários finais

- 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 deve ser feita com cuidado.
    - ➜ Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados.
  - O método não é estável.

Algumas referências sobre Quicksort

- Knuth, D.E. The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd ed. Reading, MA: Addison-Wesley, pp. 113-122, 1998.
- Sedgewick, R. Quicksort. Ph.D. thesis. Stanford Computer Science Report STAN-CS-75-492. Stanford, CA: Stanford University, May 1975.
- Sedgewick, R. "The Analysis of Quicksort Programs". Acta Informatica 7:327–355, 1977.
- Sedgewick, R. "Implementing Quicksort Programs". Communications of the ACM 21(10):847–857, 1978.

Heapsort

- Mesmo princípio da ordenação por seleção.
- Algoritmo (considere um conjunto de elementos armazenados num vetor):
  1. Selecione o menor elemento do conjunto.
  2. Troque-o com o elemento da primeira posição do vetor.
  3. Repita os passos anteriores para os $n - 1$ elementos restantes, depois para os $n - 2$ elementos restantes, e assim sucessivamente.
- Custo (comparações) para obter o menor elemento entre n elementos é $n-1$.
- Este custo pode ser reduzido?
  – Sim, através da utilização de uma estrutura de dados chamada fila de prioridades.

Fila de prioridades

- Fila:
  - Sugere espera por algum serviço.
  - Indica ordem de atendimento, que é FIFO (FIRST-IN-FIRST-OUT).

- Prioridade:
  - Sugere que serviço não será fornecido com o critério FIFO.
  - Representada por uma *chave* que possui um certo tipo como, por exemplo, um número inteiro.

- Fila de prioridades:
  - É uma fila de elementos onde o próximo item a sair é o que possui a maior prioridade.

Aplicações de fila de prioridades

- Sistemas operacionais:
  - Chaves representam o tempo em que eventos devem ocorrer (por exemplo, o escalonamento de processos).
  - Política de substituição de páginas na memória principal, onde a página a ser substituída é a de menor prioridade (por exemplo, a menos utilizada ou a que está a mais tempo na memória).
- Métodos numéricos:
  - Alguns métodos iterativos são baseados na seleção repetida de um elemento com o maior (menor) valor.
- Sistemas de tempo compartilhado:
  - Projetista pode querer que processos que consomem pouco tempo possam parecer instantâneos para o usuário (tenham prioridade sobre processos demorados).
- Simulação:
  - A ordem do escalonamento dos eventos (por exemplo, uma ordem temporal).

Tipo abstrato de dados: Fila de prioridades

- Comportamento: elemento com maior (menor) prioridade é o primeiro a sair.
- No mínimo duas operações devem ser possíveis:
  - Adicionar um elemento ao conjunto.
  - Extrair um elemento do conjunto que tenha a maior prioridade.

TAD Fila de Prioridades
Principais operações

1. Construir uma fila de prioridades a partir de um conjunto com n elementos.
2. Informar qual é o maior elemento do conjunto.
3. Inserir um novo elemento.
4. Aumentar o valor da chave do elemento i para um novo valor que é maior que o valor atual da chave.
5. Retirar maior (menor) elemento.
6. Substituir o maior elemento por um novo elemento, a não ser que o novo elemento seja maior.
7. Alterar prioridade de um elemento.
8. Remover um elemento qualquer.
9. Fundir duas filas de prioridades em uma única lista.

Observações sobre o TAD fila de prioridades

- A única diferença entre a operação de substituir e as operações de inserir e retirar executadas em seqüência é que a operação de inserir aumenta a fila temporariamente de tamanho.

- Operação "Construir":
  - Equivalente ao uso repetido da operação de Inserir.

- Operação "Alterar"
  - Equivalente a Remover seguido de Inserir.

Representações para filas de prioridades

- Lista linear ordenada
  - Construir: $O(n \log n)$.
  - Inserir: $O(n)$.
  - Retirar: $O(1)$.
  - Ajuntar: $O(n)$.
- Lista linear não ordenada (seqüencial)
  - Construir: $O(n)$.
  - Inserir: $O(1)$.
  - Retirar: $O(n)$.
  - Ajuntar: $O(1)$, no caso de apontadores, ou $O(n)$, no caso de arranjos.
- Heap
  - Construir: $O(n)$
  - Inserir, Retirar, Substituir, Alterar: $O(\log n)$
  - Ajuntar: depende da implementação. Por exemplo, árvores binomiais é eficiente e preserva o custo logarítmico das quatro operações anteriores (J. Vuillemin, 1978).

Questão importante

- Como utilizar as operações sobre fila de prioridades para obter um algoritmo de ordenação?
  - Uso repetido da operação de Inserir.
  - Uso repetido da operação de Retirar.

- Representações para filas de prioridades e os algoritmos correspondentes:
  1. Lista linear ordenada ➜ Inserção.
  2. Lista linear não ordenada (seqüencial) ➜ Seleção.
  3. Heap ➜ Heapsort.

Heap

- Heap:
  – Nome original lançado no contexto do Heapsort.

- Posteriormente:
  – "Garbage-collected storage" em linguagens de programação, tais como Lisp e Turbo Pascal/C/...

- Referência: J.W.J. Williams, Algorithm 232 Heapsort, Communications of the ACM, 7(6):347–348, June 1964.

- Construção do heap "in situ" proposto por Floyd.
  – Referência: Robert W. Floyd, Algorithm 245 Treesort 3, Communications of the ACM, 7(12):701, December 1964.

Heap

- Seqüência de elementos com as chaves

$$A[1], A[2], \ldots, A[n]$$

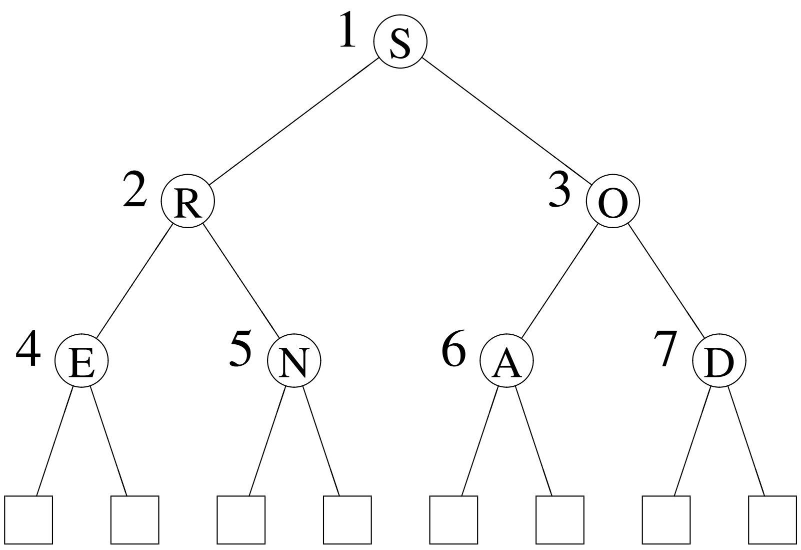
tal que

$$A[i] \geq \begin{cases} A[2i], & \text{e} \\ A[2i+1] \end{cases}$$

para $i = 1, 2, \ldots, \frac{n}{2}$

- Ordem facilmente visualizada se a seqüência de chaves for desenhada em uma árvore binária, onde as linhas que saem de uma chave levam a duas chaves menores no nível inferior:
  - Estrutura conhecida como árvore binária completa

Árvore binária completa

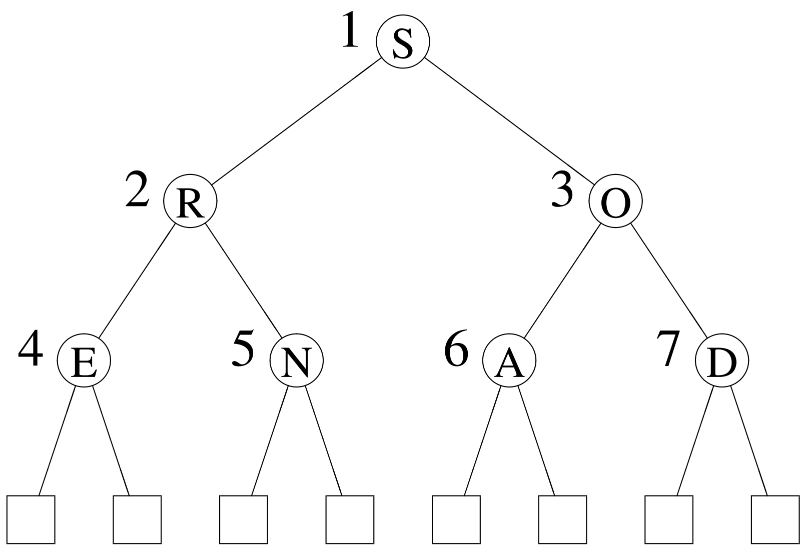


- Nós são numerados por níveis de 1 a n, da esquerda para a direita, onde o nó $\lfloor k/2 \rfloor$ é pai do nó k, para $1 < k \leq n$.
- As chaves na árvore satisfazem a condição do heap:
  - Chave de cada nó é maior que as chaves de seus filhos, se existirem.
  - A chave no nó raiz é a maior chave do conjunto.

Árvore binária completa e Arranjo

Existe uma dualidade entre a representação usando vetor e a representação de árvore.

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



Seja, n o número de nós e i o número de qualquer nó. Logo,

$$\begin{aligned}
\text{Pai}(i) &= \lfloor i/2 \rfloor && \text{para } i \neq 1 \\
\text{Filho-esquerda}(i) &= 2i && \text{para } 2i \leq n \\
\text{Filho-direita}(i) &= 2i + 1 && \text{para } 2i + 1 \leq n
\end{aligned}$$

Árvore binária completa e Arranjo

- Utilização de arranjo:
  - Representação compacta.
  - Permite caminhar pelos nós da árvore facilmente fazendo apenas manipulação de índices.

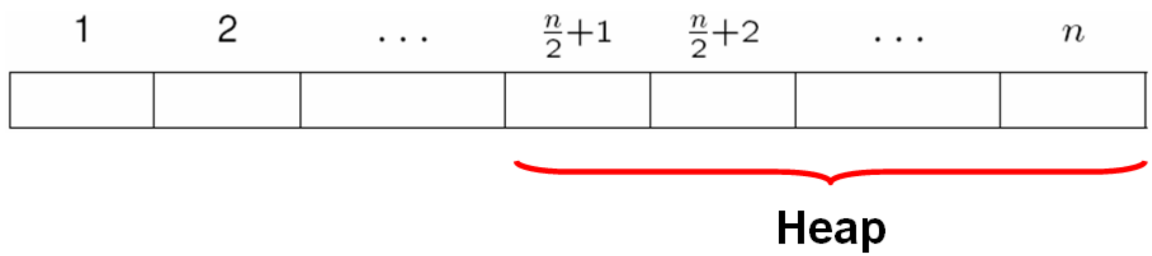
- Heap:
  - É uma árvore binária completa na qual cada nó satisfaz a condição do heap apresentada.
  - No caso de representar o heap por um arranjo, a maior chave está sempre na posição 1 do vetor.

Construção do heap: Invariante

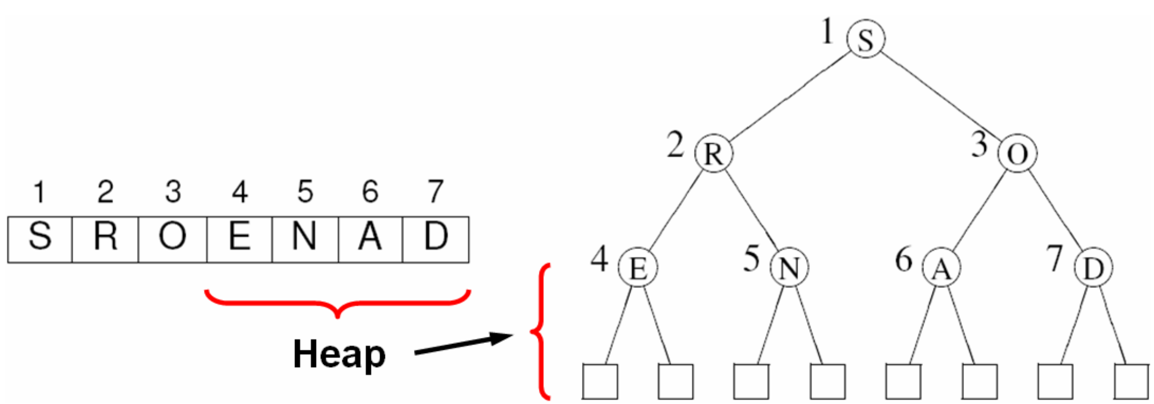
Dado um vetor $A[1], A[2], \ldots, A[n]$, os elementos

$$A[\tfrac{n}{2}+1], A[\tfrac{n}{2}+2], \ldots, A[n]$$

formam um heap, porque neste intervalo do vetor não existem dois índices i e j tais que $j = 2i$ ou $j = 2i + 1$.



Para o arranjo abaixo, temos o seguinte heap:



TAD fila de prioridades usando heap
Maior elemento do conjunto

```
function Max (var A: Vetor): Item;
begin
  Max := A[1];
end;   {Max}
```

TAD fila de prioridades usando heap
Refaz a condição de heap

```
procedure Refaz (    Esq, Dir: Indice;
                 var A       : Vetor);
label 999;
var i: Indice;
    j: integer;
    x: Item;
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  A[j].Chave <= x.Chave then goto 999;
    A[i]  := A[j];
    i     := j;
    j     := 2 * i;
  end;
  999: A[i]  := x;
end;   {Refaz}
```

TAD fila de prioridades usando heap
Constrói o heap

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

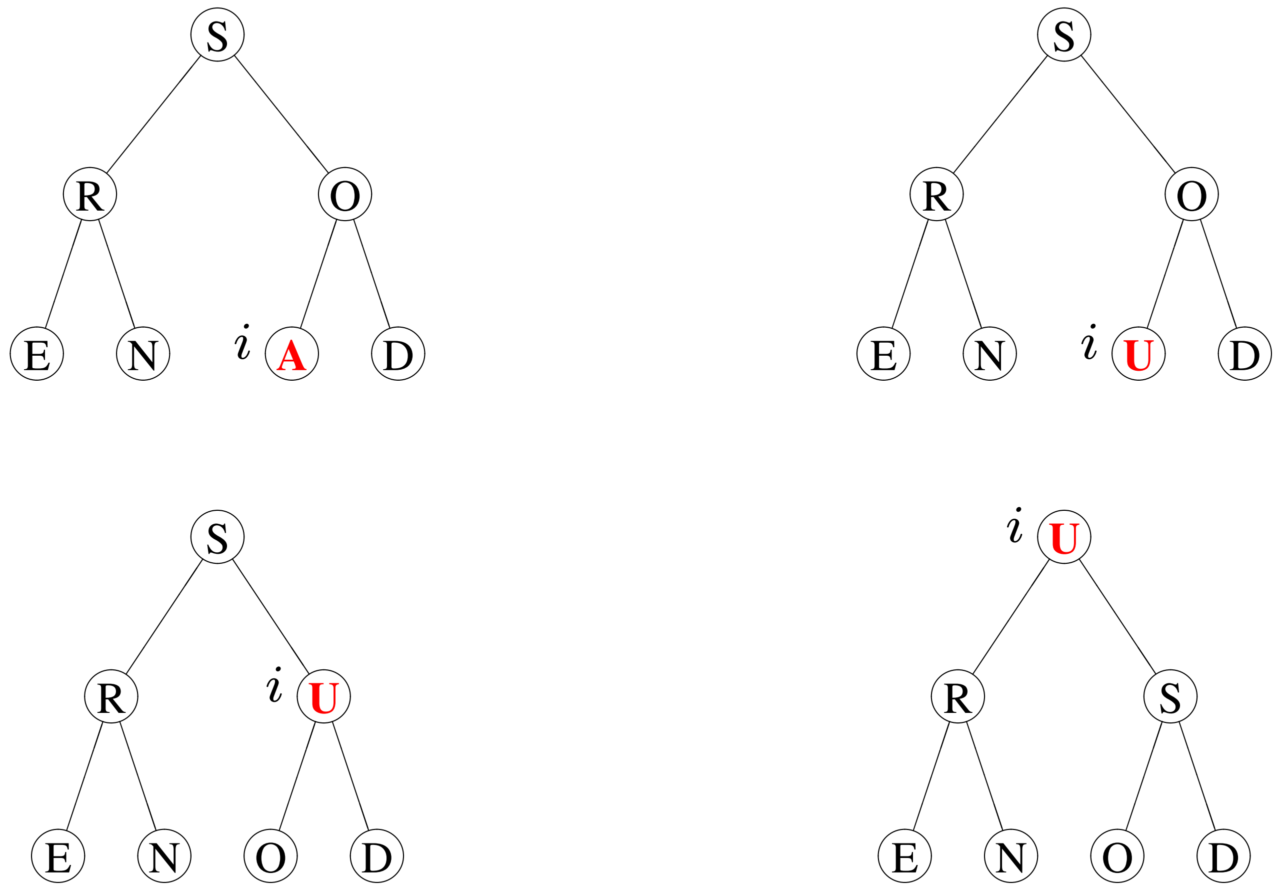
TAD fila de prioridades usando heap
Retira o item com maior chave

```
function RetiraMax (var A: Vetor;
                    var n: Indice): Item;
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;   {RetiraMax}
```

TAD fila de prioridades usando heap
Aumenta o valor da chave do item i

```
procedure AumentaChave (    i         : Indice;
                            ChaveNova: ChaveTipo;
                        var A         : Vetor);
var k: integer;
    x: Item;
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;  {AumentaChave}
```

Operação sobre o TAD fila de prioridades Aumenta o valor da chave do item na posição i



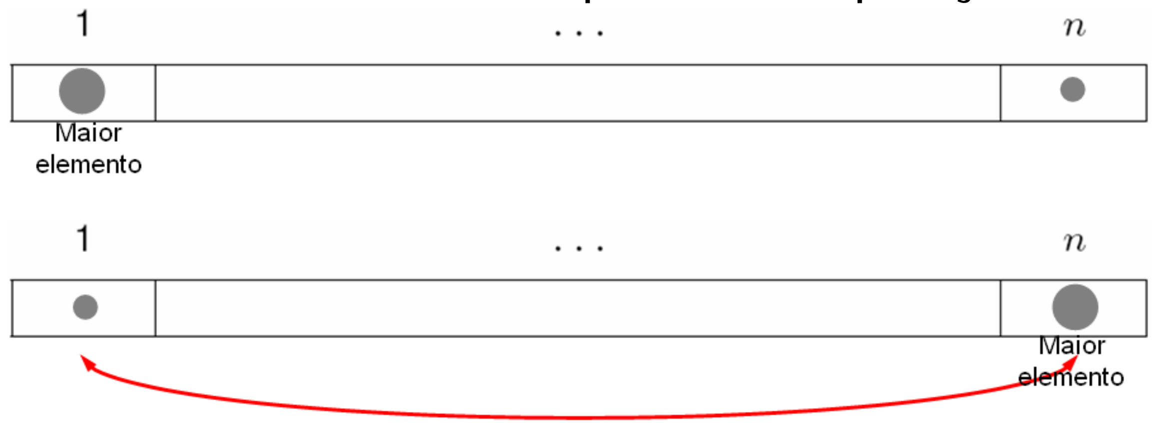
O tempo de execução do procedimento AumentaChave em um item do heap é $O(\log n)$.

TAD fila de prioridades usando heap
Insere 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;  {Insere}
```

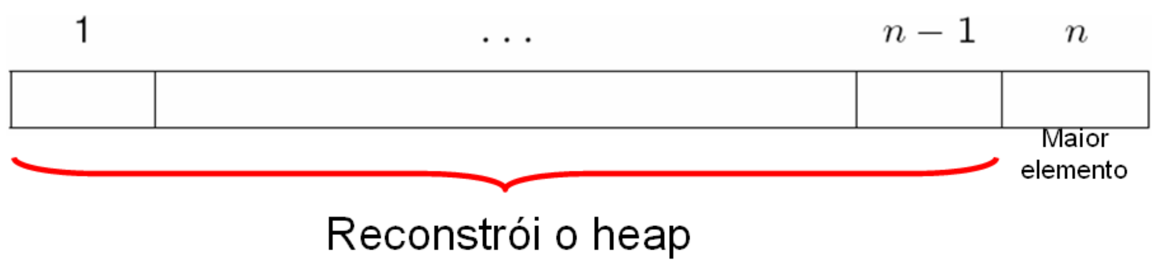
Princípio do Heapsort

1. Construir o heap.
2. A partir do heap construído, pega-se o elemento na posição 1 do vetor (raiz do heap) e troca-se com o elemento que está na posição n do vetor.



3. A seguir, basta reconstruir o heap para os elementos

$$A[1], A[2], \ldots, A[n-1].$$



4. Repita estas duas últimas operações com os $n-1$ elementos restantes, depois com os $n-2$ elementos, até que reste apenas um elemento.

Exemplo de construção do heap

	1	*2*	*3*	*4*	*5*	*6*	*7*
Chaves iniciais	O	R	D	E	N	A	S
Esq = 3	O	R	S	E	N	A	D
Esq = 2	O	R	S	E	N	A	D
Esq = 1	S	R	O	E	N	A	D

- O heap é estendido para a esquerda (Esq = 3), englobando o elemento $A[3]$, pai de $A[6]$ e $A[7]$.
- Agora a condição do heap é violada, elementos D e S são trocados.
- Heap é novamente estendido para a esquerda (Esq = 2) incluindo o elemento R, passo que não viola a condição do heap.
- Heap é estendido para a esquerda (Esq = 1), a condição do heap é violada, elementos O e S são trocados, encerrando o processo.

➜ Método elegante que não necessita de nenhuma memória auxiliar.

Exemplo de ordenação usando Heapsort

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

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

Algoritmo Heapsort

```
procedure Heapsort (var A: Vetor);
var Esq, Dir: Indice;
    x       : Item;

{Entra aqui o procedimento Refaz}
{Entra aqui o procedimento Constroi}

begin
  Constroi(A, n)                     {Constrói 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;  {Heapsort}
```

Análise do Heapsort

- À primeira vista não parece eficiente:
  - Chaves são movimentadas várias vezes.
  - Procedimento Refaz gasta $O(\log n)$ operações no pior caso.
- Heapsort:
  - Tempo de execução proporcional a $O(n \log n)$ no pior caso!
- Heapsort não é recomendado para arquivos com poucos registros porque:
  - O tempo necessário para construir o heap é alto.
  - O anel interno do algoritmo é bastante complexo, se comparado com o anel interno do Quicksort.
- Quicksort é, em média, cerca de duas vezes mais rápido que o Heapsort.
- Entretanto, Heapsort é melhor que o Shellsort para grandes arquivos.

Observações sobre o algoritmo do Heapsort

+ O comportamento do Heapsort é sempre $O(n \log n)$, qualquer que seja a entrada.

+ Aplicações que não podem tolerar eventualmente um caso desfavorável devem usar o Heapsort.

– O algoritmo não é estável, pois ele nem sempre deixa os registros com chaves iguais na mesma posição relativa.

Comparação entre os métodos: Complexidade

Método	Complexidade
Inserção	$O(n^2)$
Seleção	$O(n^2)$
Shellsort[a]	$O(n \log n)$
Quicksort[b]	$O(n \log n)$
Heapsort	$O(n \log n)$

[a] Apesar de não se conhecer analiticamente o comportamento do Shellsort, ele é considerado um método eficiente.

[b] No melhor caso e caso médio.

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

Método	500	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

Registros na ordem ascendente

Método	500	5 000	10 000	30 000
Inserção	1	1	1	1
Seleção	128	1524	3066	–
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

Registros na ordem descendente

Método	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

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 se mantém constante para todos os tamanhos.
4. A relação Shellsort/Quicksort aumenta à medida que o número de elementos aumenta.
5. Para arquivos pequenos (500 elementos), o Shellsort é mais rápido que o Heapsort.
6. Quando o tamanho da entrada cresce, o Heapsort é mais rápido que o Shellsort.
7. O Inserção é o mais rápido para qualquer tamanho 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
Influência da ordem inicial do registros

	Shellsort			Quicksort			Heapsort		
	5000	10000	30000	5000	10000	30000	5000	10000	30000
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	2	1,1	1	1

1. O 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. O 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: 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, que também é estável.
- Sua implementação é tão simples quanto a implementação dos métodos da Bolha 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: 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 1000 elementos.

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.

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 $O(n^2)$ 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: 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: 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 $O(n \log n)$.
- Aplicações que não podem tolerar eventuais variações no tempo esperado de execução devem usar o Heapsort.

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:

$$A[1], A[2], \ldots, A[n].$$

- Seja P um arranjo $P[1], P[2], \ldots, P[n]$ 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.

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.

Ordenação parcial: Aplicações

- Facilitar a busca de informação na Web com as **máquinas de busca**:
  - É comum uma consulta na Web retornar centenas de milhares de documentos relacionados com a consulta.
  - O usuário está interessado apenas nos k documentos mais relevantes.
  - Em geral k é menor do que 200 documentos.
  - Normalmente são consultados apenas os dez primeiros.
  - Assim, são necessários algoritmos eficientes de ordenação parcial.

Ordenação parcial: Algoritmos considerados

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

Seleção parcial

- Um dos algoritmos mais simples.

- Princípio de funcionamento:
  - Selecione o menor item do vetor.
  - Troque-o com o item que está na primeira posição do vetor.
  - Repita estas duas operações com os itens $n-1, n-2 \ldots n-k$.

Seleção parcial

```
procedure SelecaoParcial (var A    : Vetor;
                          var n, k: Indice);
var i, j, Min: Indice;
    x        : Item;
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;  {SelecaoParcial}
```

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

- 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.

Inserção parcial

```
procedure InsercaoParcial (var A    : Vetor;
                           var n, k: Indice);
{Nao  o restante do vetor}
var i, j: Indice;
    x   : Item;
begin
  for i := 2 to n do
  begin
    x := A[i];
    if i > k then j := k else j := i - 1;
    A[0] := x;   {Sentinela}
    while x.Chave < A[j].Chave do
    begin
      A[j + 1] := A[j];
      j        := j - 1;
    end;
    A[j+1] := x;
  end;
end;  {InsercaoParcial}
```

Observações:

1. 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.
2. O algoritmo não preserva o restante do vetor.

Inserção parcial
Algoritmo que preserva o restante do vetor

```
procedure InsercaoParcial2 (var A: Vetor;  var n, k: Indice);
var i, j: Indice;
    x   : Item;
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;  {Sentinela}
    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;  {InsercaoParcial2}
```

Inserção parcial: Análise

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

$$\begin{array}{ll} \text{Melhor caso :} & C_i(n) = 1 \\ \text{Pior caso :} & C_i(n) = i \\ \text{Caso médio :} & C_i(n) = \frac{1}{i}(1 + 2 + \cdots + i) = \frac{i+1}{2} \end{array}$$

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

$$\begin{array}{llll} \text{Melhor caso :} & C(n) & = & (1 + 1 + \cdots + 1) = n - 1 \\ \text{Pior caso :} & C(n) & = & (2 + 3 + \cdots + k + (k+1)(n-k)) \\ & & = & kn + n - \frac{k^2}{2} - \frac{k}{2} - 1 \\ \text{Caso médio :} & C(n) & = & \frac{1}{2}(3 + 4 + \cdots + k + 1 + (k+1)(n-k)) \\ & & = & \frac{kn}{2} + \frac{n}{2} - \frac{k^2}{4} + \frac{k}{4} - 1 \end{array}$$

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:

$$\begin{array}{llcl}
\text{Melhor caso :} & M(n) & = & (3 + 3 + \cdots + 3) = 3(n-1) \\
\text{Pior caso :} & M(n) & = & (4 + 5 + \cdots + k + 2 + (k+1)(n-k)) \\
 & & = & kn + n - \frac{k^2}{2} + \frac{3k}{2} - 3 \\
\text{Caso médio :} & M(n) & = & \frac{1}{2}(5 + 6 + \cdots + k + 3 + (k+1)(n-k)) \\
 & & = & \frac{kn}{2} + \frac{n}{2} - \frac{k^2}{4} + \frac{5k}{4} - 2
\end{array}$$

- 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.

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.

Heapsort parcial

```
procedure HeapsortParcial (var A  : Vetor;
                           var n,k: Indice);
{Coloca menor em A[n], segundo em A[n-1], ...,k-esimo em A[n-k]}
var Esq, Dir: Indice;
    x       : Item;
    Aux     : integer;
{Entram aqui os procedimentos Refaz e Constroi}
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;  {HeapsortParcial}
```

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:

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

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).

Quicksort parcial

Chaves iniciais:	O	R	D	E	N	A
1	A	**D**	R	E	N	O
2	**A**	D				
3			**E**	R	N	O
4				**N**	R	O
5					O	**R**
	A	D	E	N	O	R

- 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 contém 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.

Quicksort parcial

```
procedure QuickSortParcial (var A:    Vetor;
                            var n, k: Indice);
{Entra aqui o procedimento Particao}
  procedure Ordena (Esq, Dir, k: Indice);
    var i, j: Indice;
  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: 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 $O(n \log n)$.

Comparação entre os métodos parciais

n, k	Seleção	Quicksort	Inserção	Inserção2	Heapsort
$n: 10^1 \quad k: 10^0$	1	2,5	1	1,2	1,7
$n: 10^1 \quad k: 10^1$	1,2	2,8	1	1,1	2,8
$n: 10^2 \quad k: 10^0$	1	3	1,1	1,4	4,5
$n: 10^2 \quad k: 10^1$	1,9	2,4	1	1,2	3
$n: 10^2 \quad k: 10^2$	3	1,7	1	1,1	2,3
$n: 10^3 \quad k: 10^0$	1	3,7	1,4	1,6	9,1
$n: 10^3 \quad k: 10^1$	4,6	2,9	1	1,2	6,4
$n: 10^3 \quad k: 10^2$	11,2	1,3	1	1,4	1,9
$n: 10^3 \quad k: 10^3$	15,1	1	3,9	4,2	1,6
$n: 10^5 \quad k: 10^0$	1	2,4	1,1	1,1	5,3
$n: 10^5 \quad k: 10^1$	5,9	2,2	1	1	4,9
$n: 10^5 \quad k: 10^2$	67	2,1	1	1,1	4,8
$n: 10^5 \quad k: 10^3$	304	1	1,1	1,3	2,3
$n: 10^5 \quad k: 10^4$	1445	1	33,1	43,3	1,7
$n: 10^5 \quad k: 10^5$	∞	1	∞	∞	1,9
$n: 10^6 \quad k: 10^0$	1	3,9	1,2	1,3	8,1
$n: 10^6 \quad k: 10^1$	6,6	2,7	1	1	7,3
$n: 10^6 \quad k: 10^2$	83,1	3,2	1	1,1	6,6
$n: 10^6 \quad k: 10^3$	690	2,2	1	1,1	5,7
$n: 10^6 \quad k: 10^4$	∞	1	5	6,4	1,9
$n: 10^6 \quad k: 10^5$	∞	1	∞	∞	1,7
$n: 10^6 \quad k: 10^6$	∞	1	∞	∞	1,8
$n: 10^7 \quad k: 10^0$	1	3,4	1,1	1,1	7,4
$n: 10^7 \quad k: 10^1$	8,6	2,6	1	1,1	6,7
$n: 10^7 \quad k: 10^2$	82,1	2,6	1	1,1	6,8
$n: 10^7 \quad k: 10^3$	∞	3,1	1	1,1	6,6
$n: 10^7 \quad k: 10^4$	∞	1,1	1	1,2	2,6
$n: 10^7 \quad k: 10^5$	∞	1	∞	∞	2,2
$n: 10^7 \quad k: 10^6$	∞	1	∞	∞	1,2
$n: 10^7 \quad k: 10^7$	∞	1	∞	∞	1,7

Comparação entre os métodos de ordenação parcial

1. Para valores de k até 1.000, o método da InserçãoParcial é imbatível.
2. O QuicksortParcial nunca ficar muito longe da InsercaoParcial.
3. Na medida em que o k cresce,o QuicksortParcial é a melhor opção.
4. Para valores grandes de k, o método da InserçãoParcial se torna ruim.
5. Um método indicado para qualquer situação é o QuicksortParcial.
6. O HeapsortParcial tem comportamento parecido com o do QuicksortParcial.
7. No entano, o HeapsortParcial é mais lento.