Estruturas de Dados
Profa. Helena Graziottin Ribeiro

Métodos de Classificação ou de Ordenação Interna de Dados

1. Introdução

Ordenação é um processo comum e bastante estudado (há problemas que não foram resolvidos e algoritmos sendo desenvolvidos) em computação.

Ordenação Interna: ordenação na memória principal.

Para os algoritmos de ordenação que vamos estudar assumimos os seguintes detalhes:

- Os dados estarão armazenados em um vetor.
- cada elemento no vetor é um registro com uma chave de ordenação.
- podemos fazer comparações do tipo: para os registros S e R,
  se (chave(S) < = chave(R)) então ...
- Assumimos a existência da função abaixo que troca os conteúdos de R e S na memória: *troca(ref. R, ref. S)*

A eficiência de algoritmos de ordenação pode ser afetada pelos seguintes fatores:

- número de registros
- tamanho das chaves e dos registros
- até que ponto os registros estão ordenados antes de iniciar o processo de ordenação

Normalmente a eficiência pode ser medida pelo número de comparações e/ou trocas efetuadas.

2. Classificação dos métodos de ordenação

Os métodos de classificação (ou de ordenação) de dados podem ser:

- por inserção (direta, por incrementos decrescentes (*shellsort*))
- por troca (da bolha (*bubblesort*), troca e partição (*quicksort*))
- por seleção (direta, em árvore (*heapsort*))
- por distribuição de chave (*bucket sort*)
- por intercalação (*mergesort*)

3. Algoritmos dos métodos

Três algoritmos simples:

- Ordenação por inserção direta
- Ordenação pela bolha (*Bubble sort)*
- Ordenação por seleção direta

Inserção direta: cada registro é inserido no vetor na ordem correta. Considera-se o vetor dividido em dois segmentos, o primeiro está ordenado, o segundo não.

Inicialmente considera-se que o primeiro registro está na ordem correta (ele faz parte do primeiro segmento). A partir daí, por interações sucessivas pega-se cada registro do segundo segmento e compara-se com os registros do primeiro segmento, para inseri-los na ordem correta no primeiro segmento.

```
Inserção(vet,n)
   i := 2
   enquanto i < n faça
      j := i
      enquanto (j > 1) e ( chave(vet[j]) < chave (vet[j-1] ) faça
         troca (vet[j],vet[j-1])
         j := j – 1
      fim enquanto
      i := i + 1
   fim enquanto
fim
```

Exemplo (**vetor ordenado** | *vetor não-ordenado*)

início: 10 – 8 – 9
1° passo: (**10** | *8 – 9*)
2° passo: (**8 - 10** | *9*)
3° passo: (**8 – 9 - 10** |)

Bubble sort: compara-se cada par sucessivo de elementos (elementos de posição 1 e 2, depois 2 e 3, depois 3 e 4, e assim sucessivamente), trocando-os de posição caso não obedeçam a ordem desejada. Esse procedimento é repetido até que não sejam necessárias novas trocas.

```
BubbleSort(vet,n)
   i := 1
   enquanto i < n faça
      j := n
      enquanto j > i faça
         se chave(vet[j]) < chave(vet[j-1]) então
            troca (vet[j], vet[j-1])
         fim se
         j := j –1
      fim enquanto
```

```
        i := i +1
      fim enquanto
    fim
```

Exemplo (elementos em negrito mostram as trocas ou elementos considerados para trocas):

início: 10 – 8 – 9 - 7
1° passo: (**8 - 10** – *9* – *7*)
(*8* – **9 – 10** – *7*)
(*8* – *9* – **7 – 10**)

2° passo: (**8 – 9** – *7* – *10*)
(*8* – **7 – 9** – *10*)
(*8* – *7* – **9 – 10**)

3° passo: (**7 – 8** – *9* – *10*)
(*7* – **8 – *9*** – *10*)
(*7* – *8* – ***9* – 10**)

Seleção direta: Considera-se o vetor dividido em dois segmentos, o primeiro está ordenado, o segundo não. Inicialmente procura-se o menor registro do vetor, que é colocado na primeira posição (troca-se o menor registro com o registro da primeira posição). A primeira posição passa a ser o primeiro segmento, e o restante do vetor o segundo segmento. Em seguida procura-se o menor registro do segundo segmento, que é colocado na segunda posição do vetor (primeiro segmento), e assim sucessivamente, até que todos os elementos estejam ordenados.

```
    Seleção(vet,n)
      i := 1
      enquanto i < n faça
        menor := i
        j := n
        enquanto j > i faça
          se chave(vet[j]) < chave(vet[menor]) então
            menor := j
          fim se
          j := j –1
        fim enquanto
        troca(vet[i],vet[menor])
        i := i + 1
      fim enquanto
    fim
```

Exemplo (**vetor ordenado** | *vetor não-ordenado*)

início: 10 – 8 – 9
1° passo: (**8** | 10 – *9*)
2° passo: (**8 - 9** | *10*)
3° passo: (**8 – 9 - 10** |)

Comparação dos métodos:

Inserção direta:

- Pior caso: os dados já estão ordenados do menor para o maior.
- Melhor caso: os dados já estão ordenados corretamente.
- No caso de termos um vetor "quase ordenado" o algoritmo de inserção pode ser uma boa escolha.

Bubble sort: (o pior algoritmo!)

- Número de comparações no laço interno é sempre i
- Tempo é mais ou menos o mesmo no melhor e pior caso.
- Número de trocas é semelhante ao da inserção.
- Base para o algoritmo de seleção

Seleção direta:

- Princípio semelhante ao *bubble*, mas não fazemos tantas trocas.
- Vantajoso se trocar registros é um processo ineficiente.

- a complexidade dos três algoritmos é $O(n^2)$, para n registros. Mais especificamente:

	Inserção	**Bubble**	**Seleção**
Comparações:			
- melhor caso	O(n)	$O(n^2)$	$O(n^2)$
- caso médio	$O(n^2)$	$O(n^2)$	$O(n^2)$
- pior caso	$O(n^2)$	$O(n^2)$	$O(n^2)$
Trocas:			
- melhor caso	0	0	O(n)
- caso médio	$O(n^2)$	$O(n^2)$	O(n)
- pior caso	$O(n^2)$	$O(n^2)$	O(n)

Uma solução para trocas + eficientes: armazenar apontadores para os registros no vetor.

Esses algoritmos ordenam comparando registros adjacentes (algoritmos de troca), o que contribui para sua ineficiência. Há algoritmos mais eficientes, que consideram registros não adjacentes.

Exercício: ordenar as seguintes sequências iniciais de números, pelos três métodos básicos:

- 6-5-4-3-2-1
- 4-3-1-2-5-6
- 2-1-3-5-4-6

Shell Sort

Faz comparações e trocas entre elementos não adjacentes.

Explora o melhor caso do algoritmo "inserção direta": procura melhorar a ordenação do vetor para que uma ordenação por inserção direta possa terminar o trabalho (é portanto substancialmente melhor no pior caso da inserção direta).

Quebra a lista em sublistas "conceituais", cada sublista é ordenada por inserção direta. A complexidade do shellsort fica na ordem de $O(n^{1.5})$, no caso médio.

Para tornar o algoritmo exemplo mais simples consideramos que n, o número de valores a serem ordenados, é uma potência de 2.

```
shellsort(vetor, n)
        sublistas := n/2
        enquanto sublistas > 2 faça
                j := 0
                enquanto j < sublistas faça
                        inserção2(vetor, j, sublistas, n)
                        j := j + 1
                fim enquanto
                sublistas := sublistas/2
        fim enquanto
        inserção(vetor, n)
fim


inserção2(ref. vetor, inicio, incremento, n)
        i := inicio
        enquanto i < n faça
                j := i
                enquanto (j > inicio) e (chave(vetor[j]) < chave(vetor[j-incremento]) faça
                        troca(vetor[j], vetor[j-incremento])
                        j:= j - incremento
                fim enquanto
                i := i + incremento
        fim enquanto
fim
```

Exemplo:

- 1° passo: classifica 8 sublistas de tamanho 2 e incremento 8:

59 20 **17** 13 **28** 14 **23** 83 **36** 98 **11** 70 **65** 41 **42** 15

- 2° passo: classifica 4 sublistas de tamanho 4 e incremento 4:
36 20 11 13 28 14 23 15 59 98 17 70 65 41 42 83

- 3° passo: classifica 2 sublistas de tamanho 8 e incremento 2:
28 14 11 13 36 20 17 15 59 41 23 70 65 98 42 83

- 4° passo: classifica 1 sublista de tamanho 16 e incremento 1 (usa inserção direta):
11 13 17 14 23 15 28 20 36 41 42 70 59 83 65 98

- final:
11 13 14 15 17 20 23 28 36 41 42 59 65 70 83 98

Quick Sort

Algoritmo de ordenação mais rápido de propósito geral, aplica o princípio do "dividir para conquistar".

Algoritmo

Seleciona-se um pivô (elemento escolhido aleatoriamente, ou o elemento mediano do vetor), com o qual serão comparados os demais elementos.

Os registros são então re-arranjados em duas "partições":

- Os k valores menores que o pivô são colocados à esquerda dele.
- Os n-k valores maiores que o pivô são colocados à direita dele.
- O pivô é colocado na posição k.

As partições são então ordenadas usando-se q-sort

Melhor caso: partições são do mesmo tamanho.
Pior caso: pivôs são sempre o maior ou o menor valor na partição.

Melhoramentos:

- Usar o valor médio de 3 valores para o pivô.
- *Quick sort* não é muito bom para vetores pequenos (até 9 elementos), portanto:
  - Usar outro algoritmo para partições quando n é pequeno, ou
  - Não fazer nada para n pequeno e, no final, aplicar o algoritmo de inserção.

```
qsort(vet,i,j)
        indicePivo := encontraPivo(vet,i,j)
        troca(vet[indicePivo],vet[j])          /* coloca pivô na última posição do vetor */
        k := partição(vet,i-1,j,chave(vet[j])) /* k é a posição que será ocupada pelo pivô:
                                                  primeira posição do sub-array da direita */
        troca(vet[k],vet[j])                   /* coloca em vet[k] o pivô */
        Se ( (k-i) > 1) então
                qsort(vet,i,k-1)
        Fim Se
        Se ( (j-k) > 1) então
                qsort(vet,k+1,j)
        Fim Se
Fim

encontraPivo(vet,i,j)
        encontraPivo := (i+j)/2
Fim
```

```
partição(vet,e,d,pivo)
        Enquanto e < d faça
                Repita
                        e := e +1
                Até chave(vet[e] >= pivo)
                Repita
                        e := d -1
                Até chave(vet[e] <= pivo)
                troca(vet[e],vet[d])
        Fim Enquanto
        troca(vet[e],vet[d])
        partição := e
Fim
```

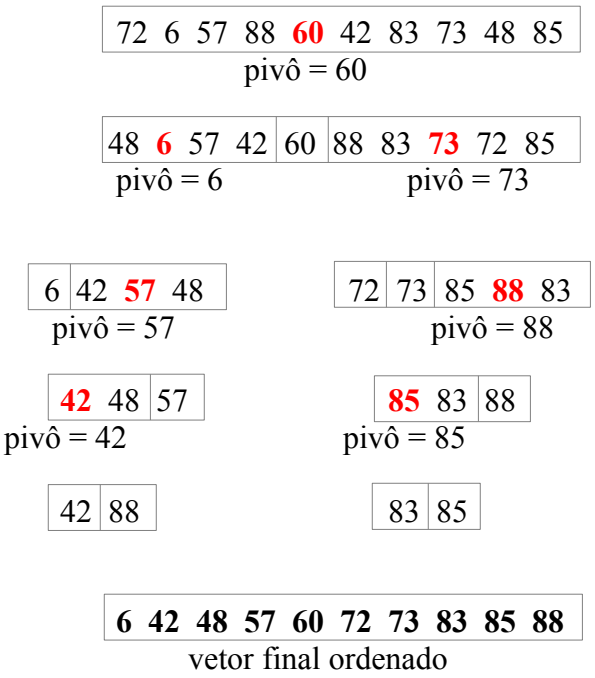
Considerações sobre o desempenho do *quicksort*:

- complexidade:
  - caso médio: O(n log n)
  - pior caso: $O(n^2)$

Exemplo (uma passada: menores que o pivô à esquerda, maiores que o pivô à direita):

inicial	72 6 57 88 85 42 83 73 48 **60** **e** **d**	60 é o pivô, colocado no final
passo 1	72 6 57 88 85 42 83 73 48 60 **e** **d**	
troca 1	**48** 6 57 88 85 42 83 73 **72** 60 **e** **d**	
passo 2	48 6 57 88 85 42 83 73 72 60 **e** **d**	
troca 2	48 6 57 **42** 85 **88** 83 73 72 60 **e** **d**	
passo 3	48 6 57 42 85 88 83 73 72 60 **d** **e**	
troca 3	48 6 57 **85** **42** 88 83 73 72 60 **d** **e**	
reverte troca	48 6 57 **42** **85** 88 83 73 72 60 **d** **e**	

Exemplo completo:



Heap Sort

Baseado na E.D. Heap: árvore binária completa (todo elemento da árvore tem dois filhos, com única exceção possível no último nível - das folhas), que pode ser armazenada em um **vetor** cujos nodos ficam em sequência por níveis: primeiro a raiz, depois os filhos da raiz, depois os filhos dos filhos da raiz, etc.

Funcionamento:

1o.) converter a sequência de elementos a serem ordenados (e que estão num array) num heap (função construirHeap())

2o.) repetidamente remover o maior valor do heap (maxHeap), que está na raiz, e colocá-lo no final do vetor ordenado, reestruturando o heap a cada vez (função removeMaximo()), até que o heap esteja vazio.

Pode ser eficiente porque:

- sua representação usando vetores e eficiente em termos de espaço
- podemos colocar todos os valores na arvore de uma só vez usando a função "construirHeap"
- heap é do tipo maxHeap: o maior elemento está na raiz (propriedade: o valor de um nodo pai é sempre maior que os valores dos seus (dois) filhos.

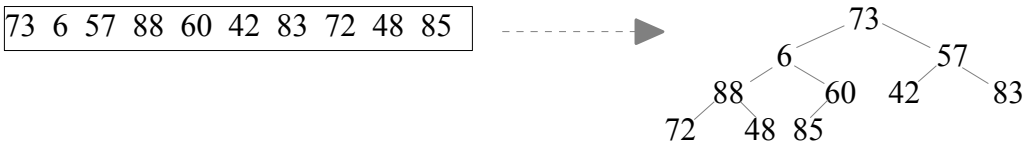
```
construirHeap(Heap)
        i := NumNodos/2 -1
        enquanto i >= 0 faca
                desce(i, Heap)
                i := i -1
        fim enquanto
fim

novoHeap(vet,n,max)
        Heap := vetor[0..max]
        Heap := vetor
        Maximo := max
        NumNodos := n
        construirHeap(Heap)
fim

HeapSort(vetor, n)
        novoHeap(vetor,n,n)          /* constrói o heap */
        i := 0
        enquanto i < n faca
                vetor[((n-1)-i)] := removeMaximo(vetor)
                i := i +1
        fim enquanto
fim
```

- HeapSort é tipicamente mais lento que Quicksort.
- Pode ser útil na situação em que queremos encontrar os k maiores valores em um conjunto de chaves.

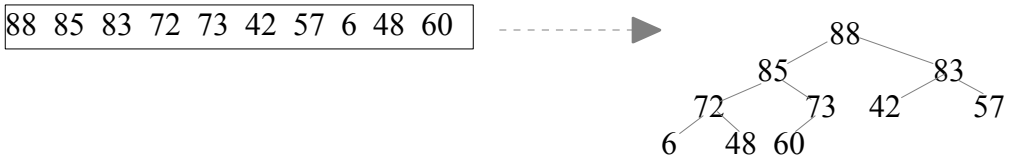
Considerações sobre o desempenho do *heapsort*:
- complexidade:
  melhor, médio e pior caso: O(n log n)

Exemplo:

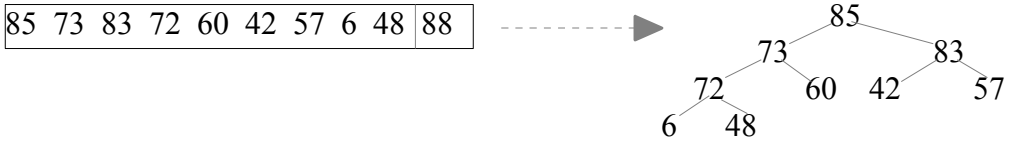
sequência original (no vetor) visualização do heap



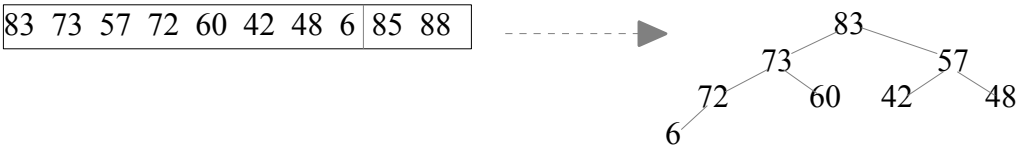
construir o heap



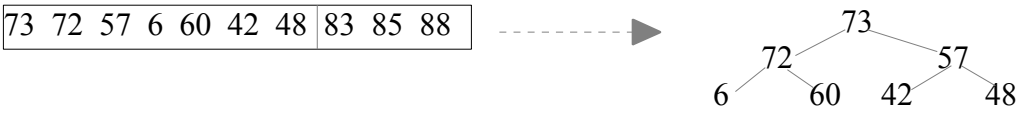
remover 88



remover 85



remover 83



Ordenação por intercalação: *Merge Sort*

Método de ordenação baseado no princípio de dividir os dados em subconjuntos e obter um vetor ordenado resultante pela *intercalação* dos valores dos dois subconjuntos: pega-se o menor valor dos dois subconjuntos, depois o segundo menor e assim por diante, inserindo-os sucessivamente num vetor resultante.

- Uma lista é dividida em duas sublistas que são ordenadas e recombinadas através do método "merge".

```
MergeSort(lst : lista)
   Se comprimento(lst) = 1 Então
      retorna lst
   Fim Se
   lista l1 = metade de lst
   lista l2 = outra metade de lst
   retorna merge(mergeSort(l1),mergeSort(l2))
Fim
```

A função "merge" examina o primeiro elemento de cada sublista, remove o menor deles da sua sublista e o inclui na lista de saída. Esse processo continua até que não haja mais dados de entrada.

O procedimento recursivo causa a sucessiva divisão la lista inicial em sublistas, até obter listas de um elemento. Assim, primeiro é realizado o "merge" das sublistas de 1 elemento, depois das sublistas de 2 elementos, em seguida das sublistas de 4 elementos, e assim por diante.

Considerações sobre o desempenho do *mergesort*:
- complexidade:
  melhor, médio e pior caso: O(n log n)

Exemplo: 36 20 17 13 28 14 23 15

- inicialmente o mergesort recursivamente divide a lista em sublistas de um elemento cada, e recombina as listas, ordenando nos passos seguintes:
36 20 17 13 28 14 23 15

- 1° passo do mergesort: mostra as 4 sublistas de tamanho 2 criadas, cada uma já ordenada e resultante da combinação de duas listas de um elemento cada:
|20 36 | 13 17 | 14 28 | 15 23|

- 2° passo do mergesort: mostra as 2 sublistas de tamanho 4 criadas pelas junções de duas listas de dois elementos cada:
|13 17 20 36 | 14 15 23 28|

- 3° passo do mergesort: mostra a lista final criada pela junção das duas listas de quatro elementos cada:
|13 14 15 17 20 23 28 36|

Ordenação por distribuição de chave: *Bucket Sort (*ou *Radix Sort)*

Método utilizado para ordenar listas cujos elementos são compostos por símbolos que têm valores diferentes segundo sua posição. Um exemplo típico são os números, compostos por dígitos que segundo sua posição podem significar unidade, dezena, centena, etc.

O número de posições a serem consideradas para ordenação define a quantidade de passos que o algoritmo precisa realizar para ordenar a lista. Por exemplo, se os números tiverem apenas unidade e dezena serão necessários dois passos para a ordenação completa.

Em cada passo serão ordenados os símbolos da posição considerada. Por exemplo, inicialmente os números serão ordenados (pelos dígitos de 0 a 9) considerando-se apenas a unidade, depois considerando-se apenas a dezena (já respeitando a ordem definida pelas unidades), e assim sucessivamente.

Considerações sobre o desempenho do *bucketsort*:
- complexidade:
  melhor, médio e pior caso: O(n)

Exemplo: ordenar a sequência 27 91 1 97 17 23 84 28 72 5 67 25

- 1° passo (ordenação da sequência de números pelo dígito mais à direita - unidade):

0
1 → 91 - 1
2 → 72
3 → 23
4 → 84
5 → 5 - 25
6
7 → 27 – 97 – 17 - 67
8 → 28
9

Resultado do 1° passo: 91 1 72 23 84 5 25 27 97 17 67 28

- 2° passo (ordenação da sequência de números pelo dígito mais à esquerda - dezena):

0 → 1 - 5
1 → 17
2 → 23 – 25 – 27 - 28
3
4
5
6 → 67
7 → 72
8 → 84
9 → 91 - 97

Resultado do 2° passo: 1 5 17 23 25 27 28 67 72 84 91 97