# Algoritmos e Estruturas de Dados II
# Ordenação

Material do Prof. Rafael Sachetto Oliveira

*Slides baseados nos disponibilizados pelo livro-texto*

# Ordenação - Conceitos Básicos

- Ordenar: processo de rearranjar um conjunto de objetos em ordem ascendente ou descendente

- Visa a 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:
  - Trabalham sobre os registros de um arquivo
  - Cada registro possui uma chave utilizada para controlar a ordenação
  - Podem existir outros componentes em um registro

# Ordenação - Conceitos Básicos

- Estrutura de um item do arquivo

```
typedef int ChaveTipo;
typedef struct Item {
  ChaveTipo chave;
  /* outros componentes */
} Item;
```

- 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

# Ordenação - Conceitos Básicos

- Classificação dos métodos de ordenação:
  - Ordenação interna: arquivo a ser ordenado cabe todo na memória principal
  - Ordenação 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 - Conceitos Básicos

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

$$A < 2 < 3 < \ldots < 10 < J < Q < K \text{ e}$$
$$\clubsuit < \diamondsuit < \heartsuit < \spadesuit$$

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

# Ordenação - Conceitos Básicos

- Métodos como o ilustrado são também conhecidos como **ordenação digital**, **radixsort** ou **bucketsort**

- O método não utiliza comparação entre chaves

- Dificuldade de implementar (uma das): 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

# Ordenação Interna

- Na escolha de um algoritmo de ordenação interna deve ser considerado o tempo gasto pela ordenação
- Sendo $n$ o número de 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

# 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 Interna

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

```
typedef int Indice;
typedef Item Vetor[MAX_TAM + 1];
Vetor A;
```

- O índice do vetor vai de 0 até $MAX\_TAM + 1$, devido às chaves sentinelas

- O vetor a ser ordenado contém chaves nas posições de 1 até $n$

# Ordenação por Seleção

- 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

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Chaves iniciais: | *O* | *R* | *D* | *E* | *N* | *A* |
| i = 1 | ***A*** | *R* | *D* | *E* | *N* | ***O*** |
| i = 2 | *A* | ***D*** | ***R*** | *E* | *N* | *O* |
| i = 3 | *A* | *D* | ***E*** | ***R*** | *N* | *O* |
| i = 4 | *A* | *D* | *E* | ***N*** | ***R*** | *O* |
| i = 5 | *A* | *D* | *E* | *N* | ***O*** | ***R*** |

# Ordenação por Seleção

```
void selecao(Vetor a, Indice *n) {
  Indice i, j, min;
  Item x;
  for (i = 1; i <= *n - 1; i++) {
    min = i;
    for (j = i + 1; j <= *n; j++) {
      if (a[j].chave < a[min].chave) {
          min = j;
      }
    }
    x = a[min];
    a[min] = a[i];
    a[i] = x;
  }
}
```

# Ordenação por Seleção - Análise

- 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 Seleção

- Vantagens:
  - Custo linear no tamanho da entrada 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

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

# Ordenação por Inserção

- Método preferido dos jogadores de **cartas**

- Algoritmo:
  - Em cada passo a partir de $i = 2$ faça:
    * Selecione o $i$-ésimo item da seqüência fonte
    * Coloque-o no lugar apropriado na seqüência destino de acordo com o critério de ordenação

| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Chaves iniciais: | **O** | *R* | *D* | *E* | *N* | *A* |
| i = 2 | **O** | **R** | *D* | *E* | *N* | *A* |
| i = 3 | **D** | **O** | **R** | *E* | *N* | *A* |
| i = 4 | **D** | **E** | **O** | **R** | *N* | *A* |
| i = 5 | **D** | **E** | **N** | **O** | **R** | *A* |
| i = 6 | **A** | **D** | **E** | **N** | **O** | **R** |

# Ordenação por Inserção

```
void insercao(Vetor a, Indice *n) {
  Indice i, j;
  Item x;
  for (i = 2; i <= *n; i++) {
      x = a[i];
      j = i - 1;
      a[0] = x;    /* sentinela */
      while (x.chave < a[j].chave) {
          a[j+1] = a[j];
          j--;
      }
      a[j+1] = x;
    }
}
```

# 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 - Análise

- Seja C(n) a função que conta o número de comparações

- No laço 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 : $Ci(n) = \frac{1}{i}(1 + 2 + \ldots + i) = \frac{i+1}{2}$

- Assumindo que todas as permutações de $n$ são igualmente prováveis no caso médio, temos:
  melhor caso: $C(n) = (1 + 1 + \ldots + 1) = n - 1$
  pior caso: $C(n) = (2 + 3 + \ldots + n) = \frac{n^2}{2} + \frac{n}{2} - 1$
  caso médio: $C(n) = \frac{1}{2}(3 + 4 + \ldots + n + 1) = \frac{n^2}{4} + \frac{3n}{4} - 1$

# Ordenação por Inserção - Análise

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

$$Mi(n) = C_i(n) - 1 + 3 = Ci(n) + 2$$

- Logo, o número de movimentos é:
  melhor caso: $M(n) = (3 + 3 + \ldots + 3) = 3(n - 1)$
  pior caso: $M(n) = (4 + 5 + \ldots + n + 2) = \frac{n^2}{2} + \frac{5n}{2} - 3$
  caso médio : $M(n) = \frac{1}{2}(5 + 6 + \ldots + n + 3) = \frac{n^2}{4} + \frac{11n}{4} - 3$

# 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

- Proposto por Shell em 1959
- É uma extensão do algoritmo de ordenação por inserção.
- Problema com o algoritmo de ordenação por inserção:
  - Troca a posição dos 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

# Shellsort

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

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

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

# Shellsort

- Como escolher o valor de $h$
  - Sequência para $h$:
    $h(s) = 3h(s-1) + 1$ para $s > 1$
    $h(s) = 1$, para $s = 1$

- Knuth (1973) 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

```
void shellsort(Vetor a, Indice *n);
```

# Shellsort

```
void shellsort(Vetor a, Indice *n) {
 int i, j;
 int h = 1;
 Item x;
 do h = h * 3 + 1; while (h < *n);
 do {
     h = (h-1)/3;
     for (i = h + 1; i <= *n; i++) {
         x = a[i];
         j = i;
         while (a[j - h].chave > x.chave) {
           a[j] = a[j - h];
           j -= h;
           if (j <= h) break;
         }
         a[j] = x;
     }
   } while (h != 1);
}
```

# Shellsort

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

# Shellsort

- 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 sequência 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:
  - Conjectura 1: $C(n) = O(n^{1,25})$
  - Conjectura 2: $C(n) = O(n(\ln n)^2)$

# 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

# 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

- A parte mais delicada do método é o processo de partição
- O vetor $A[Esq..Dir]$ 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

- Algoritmo para o particionamento:
  1. Escolha arbitrariamente um pivô $x$
  2. Percorra o vetor a partir da esquerda até que $A[i] \geq x$
  3. Percorra o vetor a partir da direita até que $A[j] \leq x$
  4. Troque $A[i]$ com $A[j]$
  5. Continue este processo até os apontadores $i$ e $j$ se cruzarem

- Ao final, o vetor $A[Esq..Dir]$ está particionado de tal forma que:
  - Os itens em $A[esq], A[esq+1], \ldots, A[j]$ são menores ou iguais a $x$
  - Os itens em $A[i], A[i+1], \ldots, A[dir]$ são maiores ou iguais a $x$

# Quicksort

- Ilustração do processo de partição:

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| *O* | *R* | **D** | *E* | *N* | *A* |
| *A* | *R* | **D** | *E* | *N* | *O* |
| *A* | **D** | *R* | *E* | *N* | *O* |

- O pivô x é escolhido como sendo $A[(i+j)div2]$
- 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

```
void particionar(Indice esq, Indice dir,
                 Indice *i, Indice *j, Vetor a) {
  Item x, w;
  *i = esq; *j = dir;
  x = a[(*i + *j) / 2];   /* obtem o pivo x */
  do {
      while (x.chave > a[*i].chave) (*i)++;
      while (x.chave < a[*j].chave) (*j)--;
      if (*i <= *j) {
        w = a[*i]; a[*i] = a[*j]; a[*j] = w;
        (*i)++; (*j)--;
      }
    } while (*i <= *j);
}
```

# Quicksort

- O anel interno do procedimento `particionar` é extremamente simples (razão pela qual o algoritmo Quicksort é tão rápido)

# Procedimento Quicksort

```
void ordenar(Indice esq, Indice dir, Vetor a) {
   Indice i, j;
   particionar(esq, dir, &i, &j, a);
   if (esq < j) ordenar(esq, j, a);
   if (i < dir) ordenar(i, dir, a);
 }
 void quicksort(Vetor a, Indice *n) {
   ordenar(1, *n, a);
 }
```

# Procedimento Quicksort

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

| | | | | | | |
|---|---|---|---|---|---|---|
| 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* |

# Quicksort - Análise

- Seja $C(n)$ a função que conta o número de comparações

- Pior caso: $C(n) = O(n^2)$
  - Ocorre quando o pivô é escolhido sempre como sendo um dos extremos de um arquivo já ordenado
  - Isto faz com que o procedimento `ordenar` seja chamado recursivamente $n$ vezes, eliminando apenas um item em cada chamada

- O pior caso pode ser evitado empregando pequenas modificações no algoritmo
  - Basta escolher três itens quaisquer do vetor e usar a **mediana dos três** como pivô

# Quicksort - Análise

- Melhor caso: $C(n) = 2C(n/2) + n = n \log n - n + 1$
  - Ocorre quando cada partição divide o arquivo em duas partes iguais

- Caso médio de acordo com Sedgewick e Flajolet (1996)

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

- Isso significa que, em média, o tempo de execução do Quicksort é $O(n \log n)$

# 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:
  - Pior caso: $O(n^2)$ comparações
  - Implementação é muito delicada e difícil:Um pequeno engano pode levar a efeitos inesperados para algumas entradas de dados
  - Não é estável

# 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

# Heapsort - Filas de Prioridades

- É uma estrutura de dados na qual 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 - Filas de Prioridades - TAD

- Operações:
  1. Construir uma fila de prioridades a partir de um conjunto com $n$ itens
  2. Informar qual é o maior item do conjunto
  3. Retirar o item com maior chave
  4. Inserir um novo item
  5. Aumentar o valor da chave do item $i$ para um novo valor que é maior que o valor atual da chave
  6. Substituir o maior item por um novo item, a não ser que o novo item seja maior
  7. Alterar a prioridade de um item
  8. Remover um item qualquer
  9. Ajuntar duas filas de prioridades em uma única

# Heapsort - Filas de Prioridades - Representação

- Lista linear ordenada:
  - Construir: $O(n \log n)$
  - Inserir: $O(n)$
  - Retirar: $O(1)$
  - Ajuntar: $O(n)$

- Lista linear não-ordenada:
  - Construir: $O(n)$
  - Inserir: $O(1)$
  - Retirar: $O(n)$
  - Ajuntar: $O(1)$ para apontadores e $O(n)$ para arranjos

# Heapsort - Filas de Prioridades - Representação

- A melhor representação é através de uma estruturas de dados chamada heap:
  - Neste caso, Construir é $O(n)$
  - Inserir, Retirar, Substituir e Alterar são $O(logn)$

- Observação: Para implementar a operação Ajuntar de forma eficiente e ainda preservar um custo logarítmico para as operações Inserir, Retirar, Substituir e Alterar é necessário utilizar estruturas de dados mais sofisticadas, tais como árvores binomiais (Vuillemin, 1978).

# Heapsort - Filas de Prioridades - Algoritmos de Ordenação

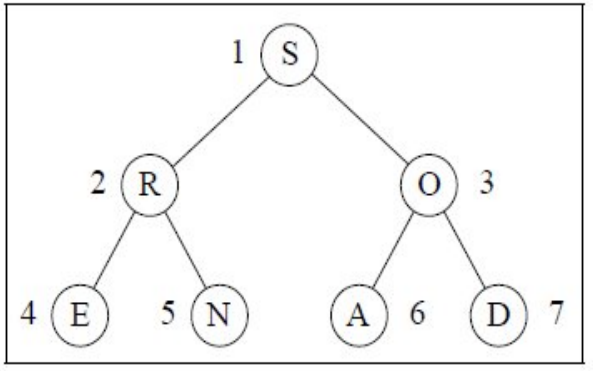
- As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação
- Basta utilizar repetidamente a operação Insere para construir a fila de prioridades
- Em seguida, utilizar repetidamente a operação Retira para receber os itens na ordem reversa
- O uso de listas lineares 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 - Heaps

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

$$c[i] \geq c[2i],$$
$$c[i] \geq c[2i+1],$$
para todo $i = 1, 2, \ldots, n/2$

- A definição pode ser facilmente visualizada em uma árvore binária completa:




- 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 $1 \leq k \leq \lfloor n/2 \rfloor$

# Heapsort - Heaps

- 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 $i \; div \; 2$

# Heapsort - Heaps

- 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 $A[1], A[2], \ldots, A[n]$, os itens $A[n/2 + 1], A[n/2 + 2], \ldots, A[n]$ formam um heap:
  - Neste intervalo não existem dois índices $i$ e $j$ tais que $j = 2i$ ou $j = 2i+1$

# Heapsort - Heaps

- Algoritmo:

| | 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* |

- 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 $A[7]$
- A condição de heap é violada:

  - O heap é refeito trocando os itens D e S

- O item R é incluindo no heap ($esq = 2$), o que não viola a condição de heap

- O item O é incluindo no heap ($esq = 1$)

- A Condição de heap violada:
  - O heap é refeito trocando os itens O e S, encerrando o processo

# Heapsort - Heaps

```
void refazer(Indice esq, Indice dir, Vetor a) {
  Indice i = esq;
  int j;
  Item x;
  j = i * 2;
  x = a[i];
  while (j <= dir) {
    if (j < dir) {
       if (a[j].chave < a[j+1].chave) j++;
    }
    if (x.chave >= a[j].chave) break;
    a[i] = a[j];
    i = j; j = i * 2;
  }
  a[i] = x;
}
```

# Heapsort - Heaps

```
 void construir(Vetor a, Indice *n) {
  Indice esq;
  esq = *n / 2 + 1;
  while (esq > 1) {
      esq--;
      refazer(esq, *n, a);
    }
}
```

# Heapsort - Algoritmo

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

# Heapsort - Exemplo

| 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* | |
| ***N*** | ***E*** | *A* | ***D*** | *O* | | |
| ***E*** | ***D*** | *A* | *N* | | | |
| ***D*** | ***A*** | *E* | | | | |
| *A* | *D* | | | | | |

- O caminho seguido pelo procedimento `refazer` 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 - Algoritmo

```
void heapsort(Vetor a, Indice *n) {
  Indice esq, dir;
  Item x;
  construir(a, n);   /* constroi o heap */
  esq = 1; dir = *n;
  while (dir > 1) {
      /* ordena o vetor */
      x = a[1]; a[1] = a[dir]; a[dir] = x;
      dir--;
      refazer(esq, dir, a);
    }
}
```

# Heapsort - Análise

- O procedimento `refazer` 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

- Vantagem:
  - 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
  - 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

# 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 - Tempo de Execução

- Oservação: 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 - Tempo de Execução

- 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

- 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 - Observações

- 1. Shellsort, Quicksort e Heapsort têm a mesma ordem de grandeza
- O Quicksort é o mais rápido para todos os tamanhos aleatórios experimentados
- A relação Heapsort/Quicksort se mantém constante para todos os tamanhos
- A relação Shellsort/Quicksort aumenta à medida que o número de elementos aumenta
- Para arquivos pequenos (500 elementos), o Shellsort é mais rápido que o Heapsort
- Quando o tamanho da entrada cresce, o Heapsort é mais rápido que o Shellsort

- O Inserção é o mais rápido para qualquer tamanho se os elementos estão ordenados
- O Inserção é o mais lento para qualquer tamanho se os elementos estão em ordem descendente
- Entre os algoritmos de custo $O(n^2)$, o Inserção é melhor para todos os tamanhos aleatórios experimentados

# 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 (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 - 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 - 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 proporcional a $nlogn$
- 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 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