COM112 - ALGORITMO E ESTRUTURA DE DADOS II[1]

Pedro Henrique Del Bianco Hokama
UNIFEI

[1]Baseado nos slides elaborados por Charles Ornelas Almeida, Israel Guerra e Nivio Ziviani

Ordenação

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

Ordenação

- Estrutura de um registro:

```
typedef long TipoChave;
typedef struct TipoItem{
  TipoChave Chave;
  /* Outros componentes */
} TipoItem;
```

- 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.

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

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

Ordenação por distribuição

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

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

desempatando por

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

- Algoritmo:
  1. Distribuir as cartas em treze montes: ases, dois, três, . . ., reis.
  2. Colete os montes na ordem contraria, de forma que o K fique em cima.
  3. Distribua novamente as cartas em quatro montes: paus, ouros, copas e espadas.
  4. Colete os montes na ordem especificada.

Ordenação por distribuição

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

Ordenação Interna

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

Ordenação Interna

Ziviani classifica os 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 $O(nlogn)$ 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 TipoIndice;
typedef TipoItem TipoVetor[MAXTAM + 1];
/* MAXTAM + 1 por causa da sentinela */
TipoVetor A;
```

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

Ordenação por Seleção

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

Ordenação por Seleção

- O método é ilustrado abaixo:

	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**

- As chaves em negrito sofreram uma troca entre si.
- Vamos ordenar o array das posições 1 até n. A posição 0 não é usada na ordenação por seleção.

Ordenação por Seleção

```
1  void Selecao(TipoItem *A, TipoIndice n){
2    TipoIndice i, j, Min;
3    TipoItem x;
4    for (i = 1; i <= n - 1; i++){
5      Min = i;
6      for (j = i + 1; j <= n; j++){
7        if (A[j].Chave < A[Min].Chave){
8          Min = j;
9        }
10     }
11     x = A[Min];
12     A[Min] = A[i];
13     A[i] = x;
14   }
15 }
```


Ordenação por Seleção

- $C(n) = \frac{n^2}{2} - \frac{n}{2}$
- $M(n) = 3(n-1)$
- A atribuição $Min = j$ da linha 8 é executada em média $n \log n$ vezes, Knuth (1973)
- A complexidade do algoritmo de ordenação por seleção é portanto $O(n^2)$

Ordenação por Seleção

Vantagens:

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

Desvantagens:

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

Ordenação por Inserção

- Método preferido dos jogadores de cartas.
- Algoritmo:
  - Em cada passo a partir de i=2 faça:
    - Selecione o i-ésimo item da sequência fonte.
    - Coloque-o no lugar apropriado na sequência destino de acordo com o critério de ordenação.

Ordenação por Inserção

- O método é ilustrado abaixo:

	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**

- As chaves em negrito representam a sequência destino.

Ordenação por Inserção

```
void Insercao(TipoItem *A, TipoIndice n){
  TipoIndice i, j;
  TipoItem 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 sequência destino é atingido à esquerda.
- Solução:
  - Utilizar um registro sentinela na posição zero do vetor.

Ordenação por Inserção - Complexidade

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

Ordenação por Inserção - Complexidade

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

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

- Logo, o número de movimentos é:
  - Melhor caso: $M(n) = (3 + 3 + \ldots + 3) = 3(n - 1)$
  - 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 - Considerações

- 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.

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 $A[Esq \ldots Dir]$ 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 $A[Esq], A[Esq+1], \dots, A[j]$ são menores ou iguais a x.
    - Os itens em $A[i], A[i+1], \dots, A[Dir]$ 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 $j = 2$.

Partição

```
void Particao (TipoIndice Esq, TipoIndice Dir,
TipoIndice *i, TipoIndice *j, TipoItem *A){
  TipoItem 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);
}
```


Partição

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

Quicksort

```c
void Ordena(TipoIndice Esq, TipoIndice Dir,
            TipoItem *A){
  TipoIndice i, j;
  Particao (Esq, Dir, &i, &j, A);
  if(Esq < j) Ordena(Esq, j, A);
  if(i < Dir) Ordena(i, Dir, A);
}

void QuickSort(TipoItem *A, TipoIndice n){
  Ordena(1, n, A);
}
```


Quicksort: Análise

- Seja C(n) a função que conta o número de comparações.
- Pior caso:

$$C(n) = O(n^2)$$

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

Quicksort: Análise

- Melhor caso:

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

- Esta situação ocorre quando cada partição divide o arquivo em duas partes iguais.
- Caso médio de acordo com Sedgewick e Flajolet (1996, p. 17):

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

• Isso significa que em média o tempo de execução do Quicksort é O(n log n).

Quicksort

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

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. (em uma busca linear)
- Isso pode ser reduzido utilizando uma fila de prioridades.

Filas de Prioridades

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

Filas de Prioridades - TAD

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

Filas de Prioridades - Representação

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

Filas de Prioridades - Representação

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

Filas de Prioridades - Algoritmos de Ordenação

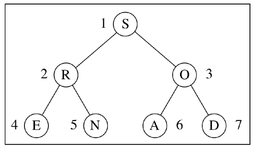
- As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação.
- Basta utilizar repetidamente a operação Insere para construir a fila de prioridades.
- Em seguida, utilizar repetidamente a operação Retira para receber os itens na ordem reversa.
- O uso de listas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

Heap

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

$$
\begin{aligned}
c[i] &\geq c[2i], &(1)\\
c[i] &\geq c[2i+1], &(2)
\end{aligned}
$$

  para todo $i = 1, 2, \ldots, n/2$.
- A definição pode ser facilmente visualizada em uma árvore binária completa:



Heap

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

Heap

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

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

Heap

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 $\lfloor i/2 \rfloor$.

Heap

- Na representação do heap em um arranjo, a maior chave está sempre na posição 1 do vetor.
- Os algoritmos para implementar as operações sobre o heap operam ao longo de um dos caminhos da árvore.
- Um algoritmo elegante para construir o heap foi proposto por Floyd em 1964.
- O algoritmo não necessita de nenhuma memória auxiliar.
- Dado um vetor $A[1], A[2], \ldots, A[n]$.
- 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 $j = 2i + 1$.

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*

- Os itens de $A[4]$ a $A[7]$ estão trivialmente atendendo as condições.
- O heap é estendido para a esquerda ($Esq = 3$), englobando o item $A[3]$, pai dos itens $A[6]$ e $A[7]$

Heap

- 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.

Heap

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

```
TipoItem Max(TipoItem *A){
  return (A[1]);
}
```


Heap

Função para refazer o heap:

```
void Refaz(TipoIndice Esq, TipoIndice Dir,
           TipoItem *A){
  TipoIndice i = Esq;
  int j; TipoItem 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;
}
```


Heap

Função para construir o heap:

```
void Constroi(TipoItem *A, TipoIndice n){
  TipoIndice Esq;
  Esq = n / 2 + 1;
  while (Esq > 1){
    Esq--;
    Refaz(Esq, n, A);
  }
}
```

Heap

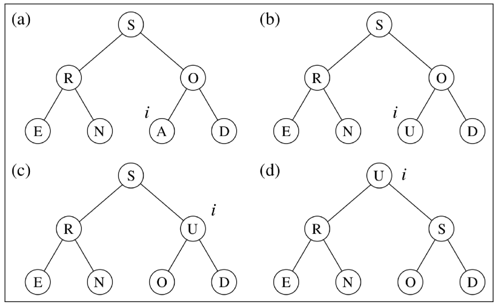
Função que implementa a operação de retirar o item com maior chave:

```
TipoItem RetiraMax(TipoItem *A,
                   TipoIndice *n){
  TipoItem Maximo;
  if (*n < 1)
    printf("Erro: heap vazio \n" );
  else{
    Maximo = A[1];
    A[1] = A[*n];
    (*n)--;
    Refaz(1, *n, A);
  }
  return Maximo;
}
```


Heap

```
void AumentaChave(TipoIndice i,
         TipoChave ChaveNova, TipoItem *A){
  TipoItem x;
  if(ChaveNova < A[i].Chave){
    printf("ChaveNova menor que atual \n");
    return;
  }
  A[i].Chave = ChaveNova;
  while(i > 1 && A[i/2].Chave < A[i].Chave){
    x = A[i/2];
    A[i/2] = A[i];
    A[i] = x;
    i = i/2;
  }
}
```


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



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

Heap

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

```
void Insere (TipoItem *x , TipoItem *A,
             TipoIndice *n){
  (*n)++;
  A[*n] = *x;
  A[*n].Chave = INT_MIN;
  AumentaChave(*n, x->Chave, A);
}
```

Heapsort

- Algoritmo:
  1. Construir o heap.
  2. Troque o item na posição 1 do vetor (raiz do heap) com o item da posição n.
  3. Use o procedimento Refaz para reconstituir o heap para os itens $A[1], A[2], \ldots, A[n-1]$.
  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 de aplicação do 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	
N	**E**	A	**D**	O		
E	**D**	A	N			
D	**A**	E				
A	D					

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

Heapsort - Implementação

```c
void Heapsort(TipoItem *A, TipoIndice n){
  TipoIndice Esq, Dir;
  TipoItem x;
  Constroi(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--;
    Refaz(Esq, Dir, A);
  }
}
```


Heapsort - Análise

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

Heapsort

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

Comparação entre os Métodos

Complexidade:

	Melhor Caso	Caso Médio	Pior Caso
SelectionSort	$O(n^2)$	$O(n^2)$	$O(n^2)$
InsertionSort	$O(n)$	$O(n^2)$	$O(n^2)$
BubleSort	$O(n)$	$O(n^2)$	$O(n^2)$
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$

MergeSort

baseado nos slides do Prof. Rafael Schouery