ESTRUTURAS DE DADOS

prof. Alexandre César Muniz de Oliveira

Sugestão bibliográfica:

- **ESTRUTURAS DE DADOS USANDO C**
Aaron M. Tenenbaum, et alli

- **DATA STRUCTURES, AN ADVANCED APPROACH USING C**
Jeffrey Esakov & Tom Weiss

- **ESTRUTURAS DE DADOS E ALGORITMOS EM JAVA (2ED)**
Michael Godrich & Roberto Tamassia

ORDENAÇÃO

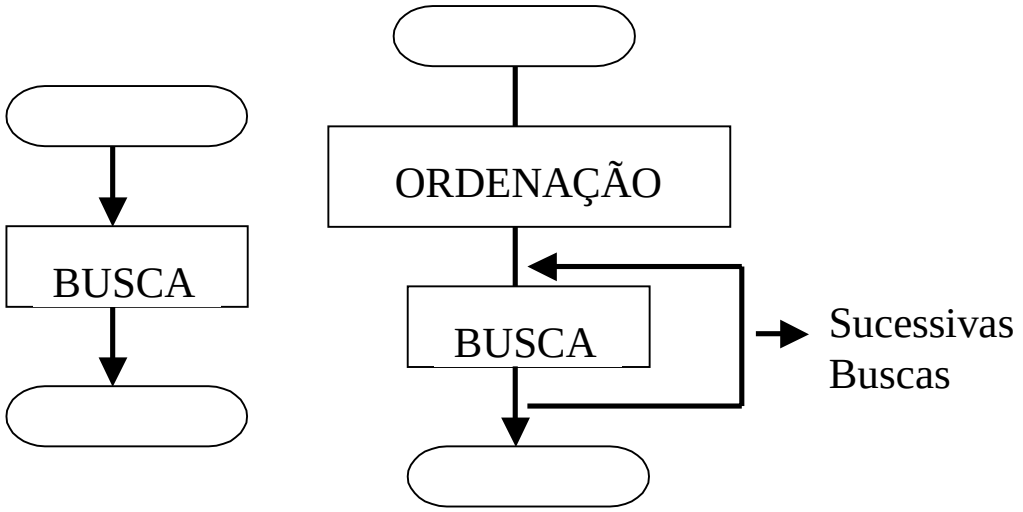
1. Introdução

Informação pode ser considerada como dados estruturados e com significado. Ordenação é uma forma de organizar dados permitindo que eles passem a ter significado e possam ser interpretados por quem os lê.

Um outro benefício da ordenação é permitir que sucessivas operações de busca sejam realizadas com maior eficiência. O tipo de estrutura de dados empregada para armazenar dados define quais algoritmos podem ser empregados para manutenção e busca.

Objetivos da ordenação:

- Classificação como ferramenta para otimizar a busca
- Visualização estruturada de informação



Ordenação e busca são operações sobre um conjunto de dados, designado como arquivo, onde a informação individualizada reside em subconjuntos chamados de registros e cada registro individual contém um ou mais campos de dados.

Uma chave é um campo ou um conjunto deles com especial significado para as operações de ordenação e busca. A chave de ordenação é usada na definição de uma relação de ordem que altera a seqüência em que são acessados os registros em um arquivo. Os registros podem ser ordenados de forma crescente ou decrescente de chave.

ORDENAÇÃO

Exemplo:

Arquivo A : Alunos da UFMA

Registro R_i : Contém informações sobre um aluno *i* específico

Campos C_j : Contêm dados relativos a cada unidade de informação, como código, nome, ano de entrada, ano de formatura, RG, CPF, data de nascimento, etc. $C_j \in R_* = \{ C_1, C_2, ..., C_{m-1}, C_m\}$.

Chave K_i : É um conjunto de campos ($K \supseteq \{C_*\}$) que identificam um dado registro. As operações de ordenação e busca são, em geral, realizadas sobre campos do tipo chave. Ordenando-se as chaves, os registros associados a cada uma delas também ficam ordenados.

Exemplo: O campo código-do-aluno ou ano-de-entrada, por exemplo, podem ser usados para ordenar o arquivo de alunos da UFMA. Observa-se que podem existir vários alunos com mesmo valor de ano-de-entrada. Uma vez ordenados, sucessivas operações de busca podem ser realizadas de forma mais eficiente, desde que executadas sobre os mesmos campos usados na ordenação.

Arquivo A: Alunos da UFMA ordenado pela chave código

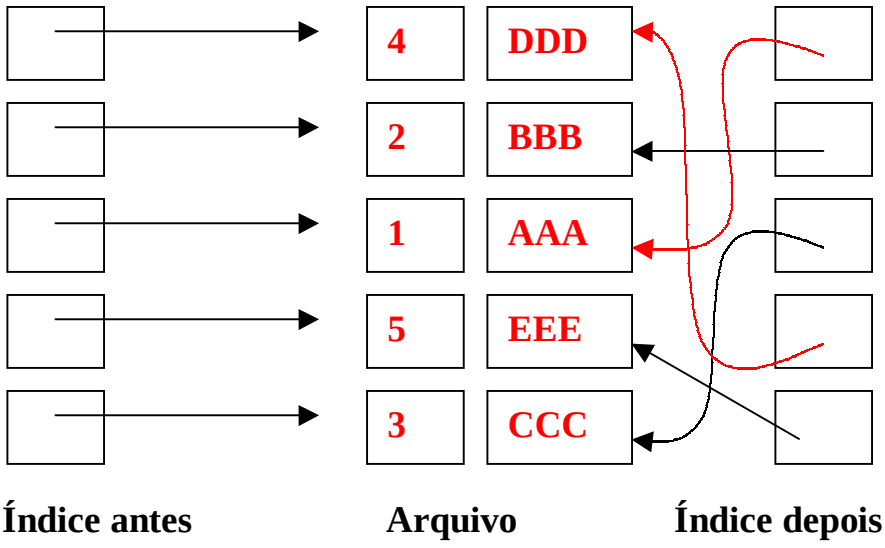
se $a < b \Rightarrow K_a < K_b$

Chave K_i	Campo 1	Campo2	...	Campo n
92101	**Ana**	**Computação**		**...**
92102	**Cláudia**	**Engenharia**		**...**
93101	**Carlos**	**Física**		**...**
93201	**Augusta**	**Química**		**...**
93202	**Baunilha**	**Desenho**		**...**

ORDENAÇÃO

2. Ordenação direta e indireta

Ordenação direta é feita diretamente sobre o conjunto de registros. Por outro lado, a ordenação indireta consegue o mesmo efeito ordenando um arquivo auxiliar chamado de índice, onde estão armazenados apenas as chaves dos registros que se pretende ordenar.



ORDENAÇÃO

2. Eficiência da técnica de ordenação

- Tempo de programação (elaboração) → - Análise do Algoritmo; - Projeto do Programa; - Escrita; - Validação
- Tempo de execução
- Espaço necessário para execução

3.1 Tempo de execução

Varia conforme máquinas, programas, e dados de entrada.

3.2 Análise matemática (complexidade do algoritmo)

- Melhor caso, pior caso, caso médio
- Exemplo (1):

Código	Operações primitivas
tmpMax := A[0]	2
for i:=1 to n-1 do	1+n*(1)+(n-1)*2
if tmpMax < A[i] then	(n-1)*2
tmpMax := A[i]	(n-1)*2
return tmpMax	1

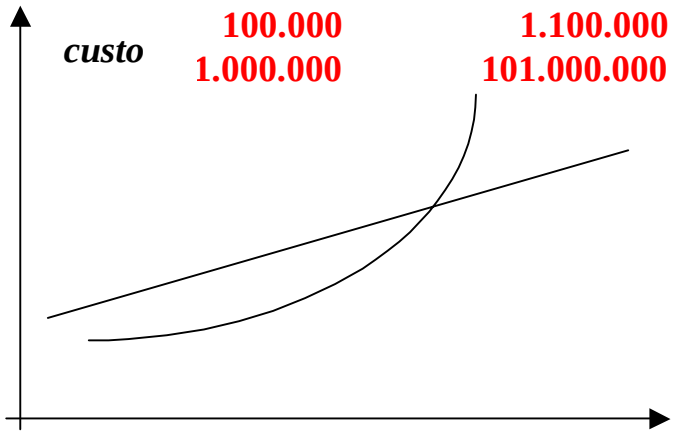
Número de operações primitivas executadas:

T(n) = 2+1+n+(n-1)*2+(n-1)*2+1 = 5n (melhor caso)

T(n) = 2+1+n+(n-1)*2+(n-1)*2+(n-1)*2+1 = 7n-2 (pior caso)

- Exemplo (2):

n	$0{,}01n^2$	$0n$	$0{,}01n^2+10n$
10	**1**	**100**	**101**
100	**100**	**1.000**	**1.100**
1.000	**10.000**	**10.000**	**20.000**
10.000	**1.000.000**	**100.000**	**1.100.000**
100.000	**100.000.000**	**1.000.000**	**101.000.000**
$n \rightarrow \infty$			



Possibilidades: $n < n\log n < n^2$

ORDENAÇÃO

3. Noções de complexidade de algoritmos

Na análise da complexidade de algoritmos é importante concentrar-se na taxa de crescimento do tempo de execução como uma função do tamanho de entrada n, obtendo-se um quadro geral do comportamento. Assim para o exemplo basta saber que o tempo de execução de algoritmo cresce proporcionalmente a n. (O tempo real seria n*factor constante, que depende de software e hardware).

4.1 Comportamento Assintótico de Funções

Comportamento a ser observado em uma função f(n), quando n tende ao infinito. O custo assintótico de uma função f(n) representa o limite do comportamento de custo quando n cresce. Em geral, o custo aumenta com o tamanho n do problema. Para valores pequenos de n, mesmo um algoritmo ineficiente não custa muito para ser executado.

f(n) domina assintoticamente g(n) se existem duas constantes positivas c e n_0 tais que, para $n \geq n_0$, temos $|g(n)| \leq c\,|f(n)|$

Seja $g(n) = n$ e $f(n) = -n^2$

Temos que $|n| \leq |-n^2|$ para todo $n \in N$. Fazendo $c = 1$ e $n_0 = 0$ a definição é satisfeita. Logo, f(n) domina assintoticamente g(n).

4.1 Notação O (*big O* ou *"O" grande*)

Notação trazida da matemática por Knuth (1968):

g(n) = O(f(n)), Lê-se:

- g(n) é de ordem no máximo f(n)
- f(n) domina assintoticamente g(n)
- f(n) é um limite assintótico superior para g(n)

O(f(n)) representa o conjunto de todas as funções que são assintoticamente dominadas por uma dada função f(n).

NOTAÇÃO O

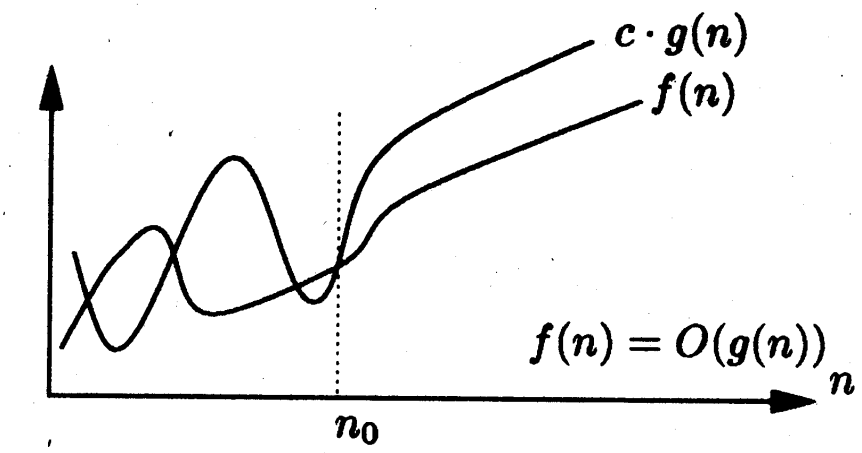
Se $g(n) \in O(f(n))$ então $g(n) = O(f(n))$

Formalmente:

$$\mathbf{g(n) = O(f(n)),\quad \exists\ c > 0\ e\ n_0\quad /\quad 0 \leq g(n) \leq c\ f(n),\ \forall n \geq n_0}$$

ORDENAÇÃO

Graficamente:



A notação O é usada para expressar o limite superior do tempo de execução de cada algoritmo para resolver um problema específico (limite superior de cada algoritmo para um problema).

OPERAÇÕES

- **f(n) = O(f(n))**
- **cO(f(n)) = O(f(n)) c constante**
- **O(f(n)) + O(f(n)) = O(f(n))**
- **O(O(f(n))) = O(f(n))**
- **O(f(n)) + O(g(n)) = O(max(f(n),g(n)))**
- **O(f(n))O(g(n)) = O(f(n)g(n))**
- **f(n)O(g(n)) = O(f(n)g(n))**

Exemplos:

a) Suponha 3 trechos de programas cujos tempos de execução sejam O(n), $O(n^2)$ e O(nlogn). O tempo de execução dos 2 primeiros trechos é O(max(n,n2)) = $O(n^2)$. O tempo de execução de todos os 3 trechos é, então, O(max(n^2, nlogn)), que é O(n2).

b) O produto de O(logn) por O(n) é O(n logn)

c) $g(n) = (n+1)^2$ provar que $g(n) = O(n^2)$?
Se existirem duas constantes positivas c e n_0 tais que, para $n \geq n_0$:
$|(n+1)^2| \leq c\,|n^2| \therefore |(n^2+2n+1)| \leq c\,|n^2| \therefore |(1+2/n+1/n^2)| \leq c$
Para n=1 todo $c \geq 1+2+1 \geq 4$ satisfaz a condição. Para n infinito?

n	*(n+1)²*	*4n²*
0	*1*	*0*
1	*4*	*4*
2	*9*	*16*
3	*16*	*36*
4	*25*	*64*

ORDENAÇÃO

4.2 Notação Ω (*big* Ω ou *"Ω" grande*)

$$g(n) = \Omega(f(n))$$

- Lê-se g(n) é de ordem no mínimo f(n).

- Ω define um limite inferior para a função, por um fator constante.

***f (n) é* um limite assintótico inferior para g(n)**

Se $g(n) \in \Omega(f(n))$ então $g(n) = \Omega(f(n))$

Formalmente:

$$\mathbf{g(n) = \Omega(f(n)), \exists\ \exists\ c > 0\ \ e\ \ n_0\ \ |\ \ 0 \leq cf(n) \leq g(n), \forall\ \forall\ n \geq n_0}$$

A notação Ω é usada para expressar um limite inferior intrínseco ao problema. O **limite inferior** para qualquer algoritmo de **ordenação** é $\Omega(n)$ (vetor já ordenado).

ORDENAÇÃO

4.3 Notação θ (*big* θ ou "θ" *grande*)

$$g(n) = \Theta(f(n))$$

- Lê-se g(n) é da mesma ordem de f(n).
- Θ limita a função por fatores constantes.

f (n) é um limite assintótico *superior e inferior* para g(n)

ou

f(n) é um limite assintótico firme para g(n).

Θ diz que para $n \geq n_o$, o valor de *g(n)* está sempre entre $c_1 f(n)$ e $c_2 f(n)$ inclusive.

Formalmente:

$g(n) = \Theta(f(n))$, $\exists \ \exists\ c_1 > 0$, $c_2 > 0$ e n_o | $0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)$, $\forall \ \forall\ n \geq n_o$

Exemplo: Seja: $1/2n^2 - 3n = \Theta(n^2)$

Prova : Para provar esta afirmação encontre constantes $c_1 > 0$, $c_2 > 0$, $n > 0$, tal que $c_1 n^2 \leq 1/2n^2 - 3n \leq c_2 n^2$ para todo $n \geq n_0$

Ao dividir a expressão acima por n^2 tem-se: $c_1 \leq ½ - 3/n \leq c_2$

A inequação mais a direita será sempre válida para qualquer valor de n ≥ 1 ao escolher $c_2 \geq 1/2$.

A inequação mais a esquerda será sempre válida para qualquer valor de $n \geq 7$ ao escolher $c_1 \geq 1/14$.

Assim: ao escolher $c_1 = 1/14$, $c_2 = 1/2$ e $n_0 = 7$, verifica-se que $1/2n^2 - 3n = \Theta(n^2)$.

ORDENAÇÃO

3.4 Análise assintótica de funções:

Pode-se dizer que:

- O(f(n)) depende do algoritmo (limite superior)
- **Ω**(f(n)) depende do problema (limite inferior)
- (f(n)) depende de ambos ("limite ótimo")

Se f é uma função de complexidade para um algoritmo F, então O(f) é considerada a complexidade assintótica, ou o comportamento assintótico do algoritmo F. A relação de dominação assintótica permite comparar funções de complexidade. Se as funções f e g dominam assintoticamente uma a outra então os algoritmos associados são equivalentes. Nestes casos, o comportamento assintótico não serve para comparar os algoritmos. Por exemplo, sejam F e G dois algoritmos aplicados à mesma classe de problemas. O algoritmo F leva três vezes o tempo de G para ser executado, ou seja, f(n) = 3g(n), sendo que O(f(n) = O(g(n)). Logo o comportamento assintótico não serve para comparar os algoritmos F e G porque eles diferem apenas por uma constante.

3.5 Classes de comportamento assintótico

f (n) = O(1) (complexidade constante)

O uso do algoritmo independe do tamanho de n. Neste caso as instruções do algoritmo são executadas um número fixo de vezes.

f(n)= O(n) (complexidade de linear)

Em geral um pequeno trabalho é realizado sobre cada elemento de entrada.

Esta é a melhor situação possível para um algoritmo que tem que processar n elementos de entrada ou produzir n elementos de saída. Cada vez que n dobra de tamanho o tempo de execução dobra.

f(n) = O(log n) (complexidade logarítmica)

Ocorre tipicamente em algoritmos que resolvem um problema transformando-o em problemas menores. Nestes casos, o tempo de execução pode ser considerado como sendo menor do que uma constante grande. Por exemplo, quando n é um milhão, log2 n ≈ 20.

f(n)= O(nlogn)

Este tempo de execução ocorre tipicamente em algoritmos que resolvem um problema quebrando-o em problemas menores, resolvendo cada um deles independentemente e depois ajuntando as soluções. Por exemplo, quando n é um milhão e a base do logaritmo é 2, nlog2 n é cerca de 20 milhões.

ORDENAÇÃO

f(n)= O(n^2) (complexidade quadrática)

Algoritmos desta ordem de complexidade ocorrem quando os itens de dados são processados aos pares, muitas vezes em um anel dentro de outro. Por exemplo, quando n é mil, o número de operações é da ordem de 1 milhão. Algoritmos deste tipo somente são úteis para resolver problemas de tamanhos relativamente pequenos.

f(n)= O(2^n) (complexidade exponencial)

Algoritmos desta ordem de complexidade geralmente não são úteis sob o ponto de vista prático. Eles ocorrem na solução de problemas quando se usa força bruta para resolvê-los. Por exemplo, quando n é vinte, o tempo de execução é cerca de um milhão. Problemas que somente podem ser resolvidos através de algoritmos exponenciais são ditos *intratáveis*.

3.6 Operações críticas para ordenação e busca

Para ordenação e busca as operações primitivas relevantes (operações críticas) são :

- comparar chave
- trocar elementos

Incrementar um índice, por exemplo, não é considerado uma operação crítica.

5. Classificação por troca de elementos

5.1 Ordenação Bolha

É o método de ordenação mais básico que se conhece: é o primeiro a ser utilizado por estudantes de computação. Ele funciona da seguinte maneira: compara-se X_i a X_{i+1} e, se X_i for maior (ou menor, dependendo da ordem), é feita a troca de posição. A classificação é encerrada quando não há mais trocas.

INTERAÇÃO	
0	25 57 48 37 12 92 86 33
1	25 48 37 12 57 86 33 92
2	25 37 12 48 57 33 86 92
3	25 12 37 48 33 57 86 92
4	12 25 37 33 48 57 86 92
5	12 25 33 37 48 57 86 92
6	12 25 33 37 48 57 86 92
7	12 25 33 37 48 57 86 92

ORDENAÇÃO

A cada alteração um elemento é posicionado corretamente, logo:

Arquivo com n elementos → interações (n-1)

A cada interação i, todos os elementos acima ou igual posição a n-i já estão classificados. Existe a possibilidade de todo o arquivo estar classificado antes das (n-1) interações máximas. Existe uma melhoria: marcar a última posição com troca em que houve troca e não fazer comparações a partir daí.

Algoritmo:

```
Para cada { interação e troca > 0} faça
    Para j de 0 até trocou faça
        Se X[j] > X [j+1]
            trocou ← j;
            troca (X[j], X[j+1]);
        Fim se;
    Fim para;
Fim para;
```

Eficiência:

01. Tempo de programação: simples
02. Espaço para armazenamento: apenas uma variável auxiliar para troca
03. Tempo de execução: f(comparações e trocas)

MELHOR CASO:

- Conjunto ordenado:

n comparações e 0 troca → O(n)

PIOR CASO:

- Conjunto desordenado:

$$(n-1) + (n-2) + (n-3) + ... + (n-k)$$

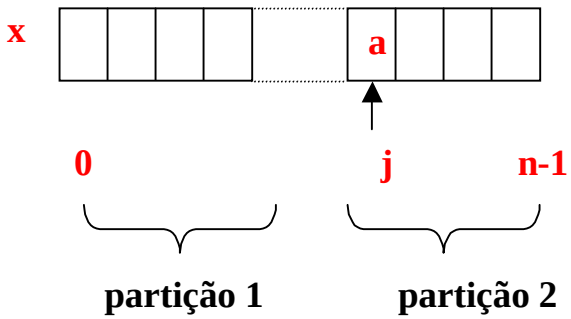
$$(2kn-k^2-k)/2 \Rightarrow k \rightarrow n-1 \rightarrow k \text{ é } O(n)$$

$$\text{logo: } O(n^2)$$

ORDENAÇÃO

5.2 Classificação por troca de partição (*Quicksort*)

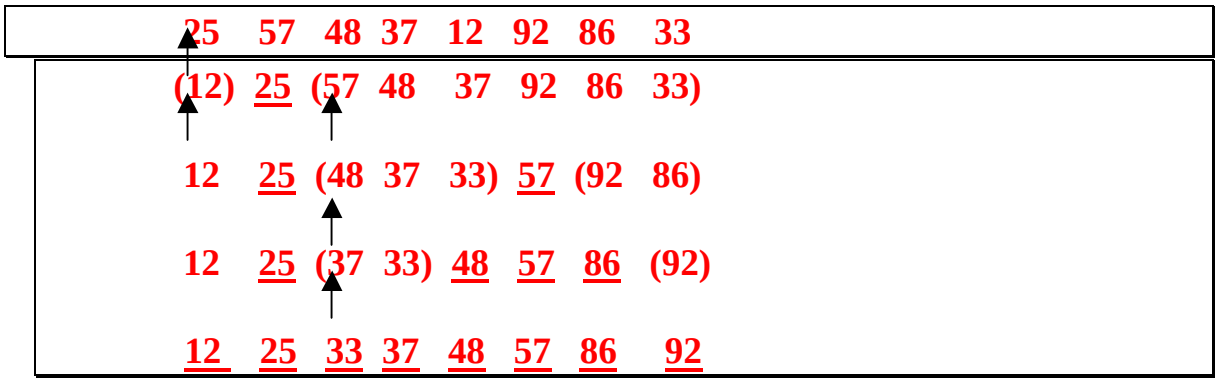
Seja x um conjunto de n elementos
Seja a um elemento dentro de x



- Cada elemento nas posições de 0 até j-1 seja menor ou igual a a.
- Cada elemento nas posições de j+1 até n-1 seja maior ou igual a a.

Nessas condições, a está em sua devida posição dentro de x. Sucessivas interações, em cada uma das partições que vão se formando, tornam x ordenado.

Exemplo(1):

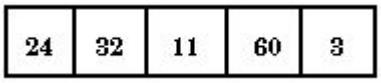


ORDENAÇÃO

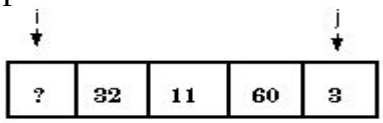
Exemplo(2):

Dado o vetor (24, 32, 11, 60, 3), aplica-se o algoritmo visto anteriormente.

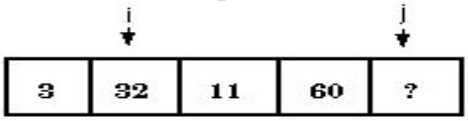
Vetor com dados iniciais:



Inicializa-se *i* e *j* , e o conteúdo de *i* é retirado para que ele seja a chave particionadora:

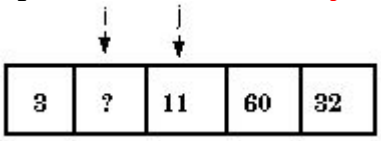


Compara-se o conteúdo de *j* com a chave. Como é menor, 3 é deslocado para *i*. Como houve preenchimento, incrementa-se *i*:

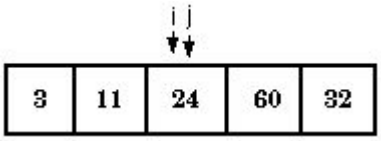


Compara-se o conteúdo da chave com *i* . Como a chave é menor, ocorre o preenchimento de X_i e decrementa-se *j* :

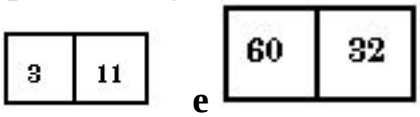
Mais uma vez, compara-se a chave com Xj e, como este é maior que a chave, apenas decrementa-se *j* :



Como X_j é menor do que a chave, desloca-se e incrementa-se *i* , atingindo o ponto mediano:



O processo se repete para os segmentos:



ORDENAÇÃO

ALGORITMO QUICKSORT

```
quicksort ( vetor, limbaixo, limalto )
{se limbaixo >= limalto
        return;
senão {
        part := particione ( vetor, limbaixo,
                             limalto);
        quicksort ( vetor, limbaixo, part-1);
        quicksort ( vetor, part+1, limalto);
}
}
```

```
particione ( vetor, lb, la )
{pivô := vetor[lb];
 up := la; down := lb;
enquanto ( down < up ){
        enquanto ( vetor [ down ] <= pivô
        && down < up ) down++;
        enquanto ( vetor [up] > pivô ) up--;
        se ( down < up )
                vetor [down] ↔ vetor [up];
              vetor [lb] := vetor [up];
              vetor[up] :=pivô;
}
return [up];
 }
```

Simulação do *Quicksort*

pivô ← 25;

25	57	48	37	12	92	86	33

down ud

1. **incrementa down até vetor [down] > pivô**
2. **decrementa up até vetor [up] ≤ pivô**

25	57	48	37	12	92	86	33

down ud

3. **se down < up troca vetor [down] por vetor [up]**

25	12	48	37	57	92	86	33

down ud

4. **até que down ≥ up**

25	12	48	37	57	92	86	33

down ud

➥ *pivô está na posição correta é base para particionar em 2 subconjuntos a serem ordenados.*

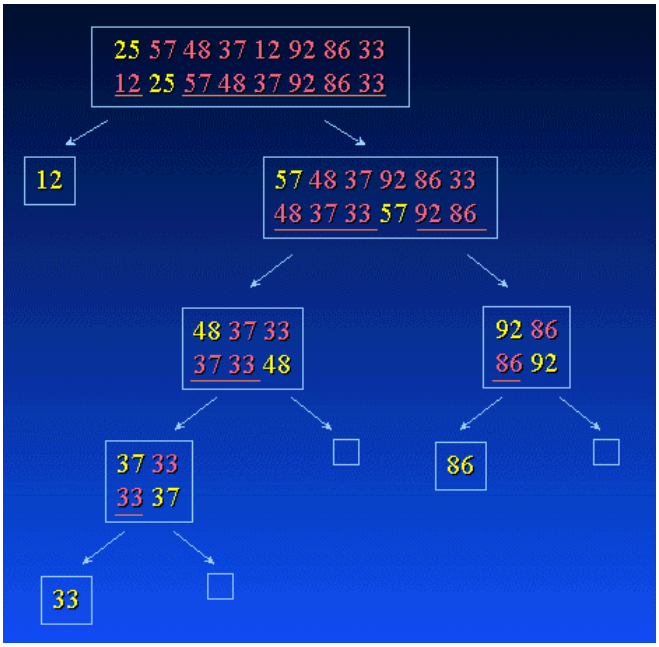
5. **troca o vetor [up] pelo pivô**

12	25	48	37	57	92	86	33

ORDENAÇÃO

VISÃO DA RECURSIVIDADE DO QUICKSORT COMO ÁRVORE

Para evitar o pior caso e casos ruins onde elementos estão em grupos ordenados pode-se utilizar uma estratégia probabilística, selecionando, como pivô, ao invés do primeiro elemento, um elemento aleatório.



Seleção média: pegue ao invés do elemento inicial, sempre o do meio.

EFICIÊNCIA DO QUICKSORT

Seja um arquivo de tamanho n de modo que: $n = 2^m$

- 1ª passagem:

n comparações gerando 2 partições com $n/2$ e $n/2$

- **Ao todo:**

$$n+2\times\left(n/2\right)+4\times\left(n/4\right)+\ldots+n\left(n/n\right)$$

$$n+n+n+n+\ldots+n=m\times n$$

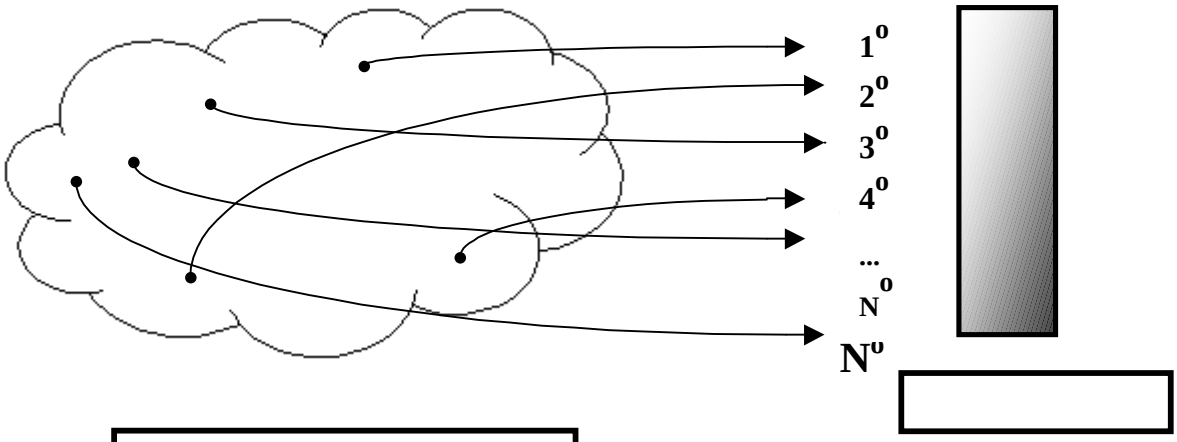
$$O(n\times m)\Rightarrow m=\log_2^n$$

$$O(n\log^n)$$

Para arquivos quase ordenados "Bolha" é bom e "Quicksort" chega a $O(n^2)$. Para arquivos "caóticos" *Quicksort* é 0 ($n \log_2(n)$)

ORDENAÇÃO

6. Classificação por seleção e por árvore



CONJUNTO
"PRÉ-PROCESSADO"

PRÉ-PROCESSADOR:
- Fila de prioridade
- Lista classificada
- Árvore

Algoritmo Seleção Geral

...

for (i = 0; i < N; i++)
insere X [i] no pré-processador
for (i = 0; i < N; i++)
remove $\overline{X}$ *[i] do pré-processador*

...

TIPOS DE ALGORITMOS:

Direta
Árvore
HeapSort

ORDENAÇÃO

6.1 Classificação Por Seleção Direta

Basicamente:

1. i → posição do maior item.
2. j = 0 até i procurando o maior item.
3. No final coloca o maior item em i.

```
SelectSort ( int X [], int N )
{
        int i, j, posmaior, maior;
        for (i = n-1; i > 0; i--)
        {
                maior = X [0];
                posmaior = 0;
                for (j = n-1; j ≤ i; j++)
                        if ( X [j] > maior)
                        {
                                maior = X [j];
                                posmaior= j;
                        }
                X [posmaior] = X [i];
                X [i] = maior;
        }
}
```

Procura pelo maior

Coloca maior na última posição i

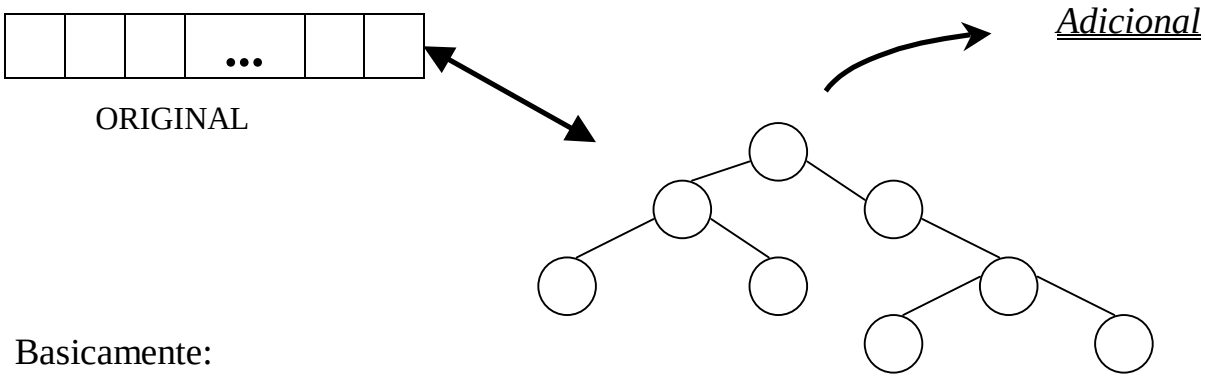
EFICIÊNCIA:

$$(n-1)+(n-2)+...+2+1=n\times(n-1)/2$$

$$O(n^2) \cong \text{bolha}$$

ORDENAÇÃO

6.2 Classificação Por Árvore Binária



Basicamente:

ArvoreSort (int X [], int N)
{

1. Criar árvore;
2. Transferir X→ árvore;
3. Percorrer árvore em ordem;
4. Transferir árvore → X;
5. Liberar árvore;

}

EFICIÊNCIA:

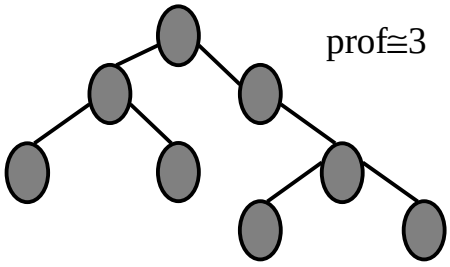
1. Classificado ou em ordem inversa → Árvore ≅ lista;

   Nº de comparações = $2 + 3 + 4 + ... + n = \frac{n \times (n+1)}{2} - 1 = O(n^2)$

2. Totalmente desclassificado (Caso Médio)

   Nº de comparações = função (profundidade)

   Ex.: $N = 8 = 2^3$ → Nº comp≤ prof → Nº comp = O (nlogn)



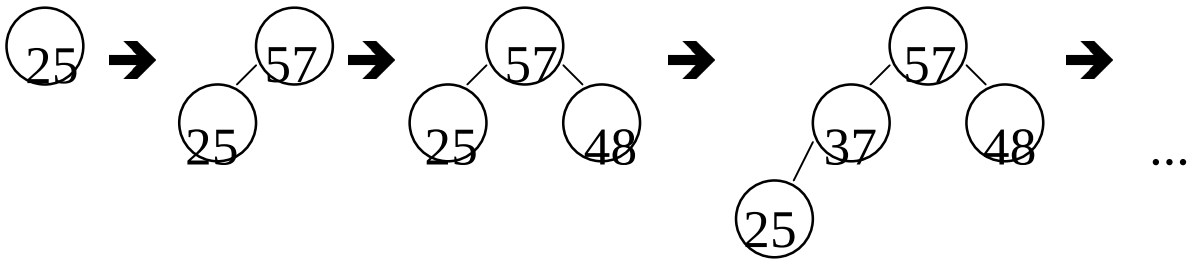
ORDENAÇÃO

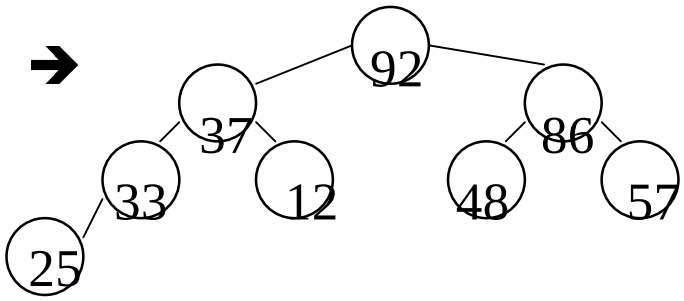
6.3 Classificação Por Heap (*HeapSort*)

Heap = Árvore Binária Completa (folhas em níveis adjacentes)
Ou seja:

i) ∀ folha em níveis adjacentes
ii) ∀ níveis preenchidos, exceto o último: esquerda primeiro
iii) chave (raiz) ⇒ ≥ chave (esq (raiz)) & ≥ chave (dir (raiz))

Construir um heap a partir de: 25, 57, 48, 37, 12, 92, 86, 33

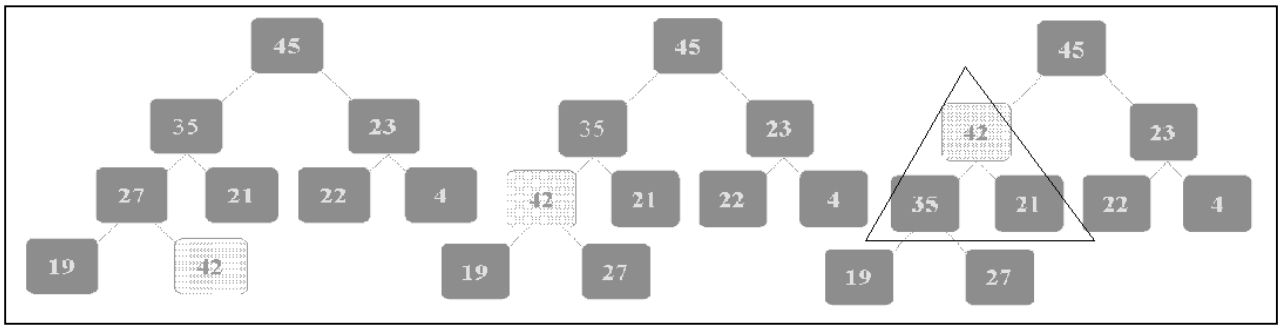




|HEAP| = 8

Inserir novo nó em um heap:

1 Coloque o novo nó na próxima posição disponível i) e ii)
2 Troque com seu "nó-pai" até que o novo nó satisfaça iii)

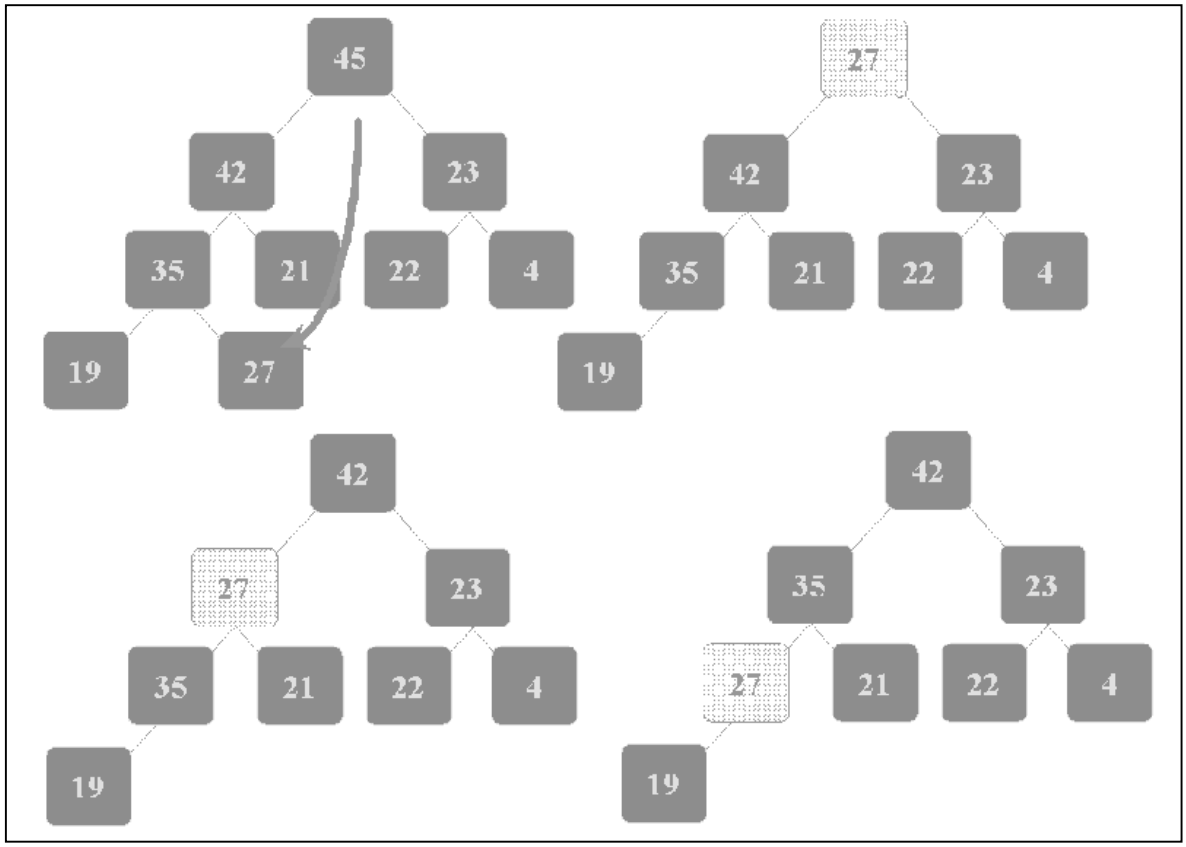


ORDENAÇÃO

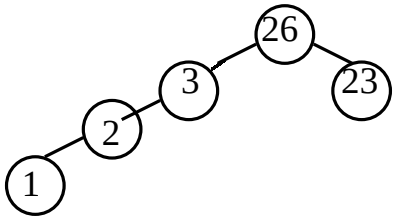
Remover o topo de um heap:

3 Mova o último nó para o nó raiz.

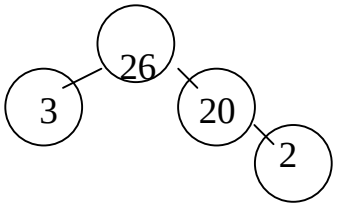
4 Coloque o nó que está fora de lugar um nível abaixo, trocando-o com seu maior "nó-filho" até satisfazer iii)



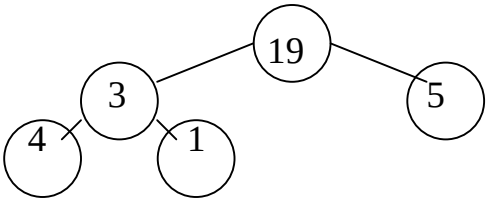
Contra-exemplos



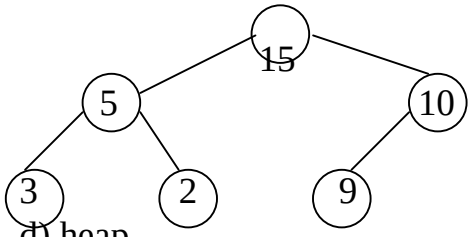
a) não-heap (i)



b) não-heap (ii)



c) não-heap (iii)

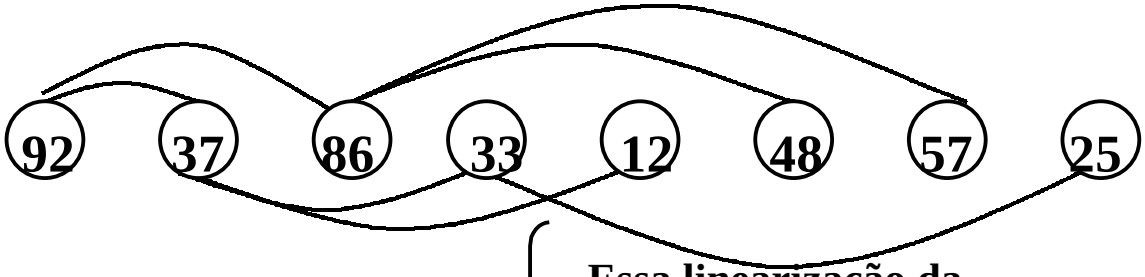


d) heap

ORDENAÇÃO

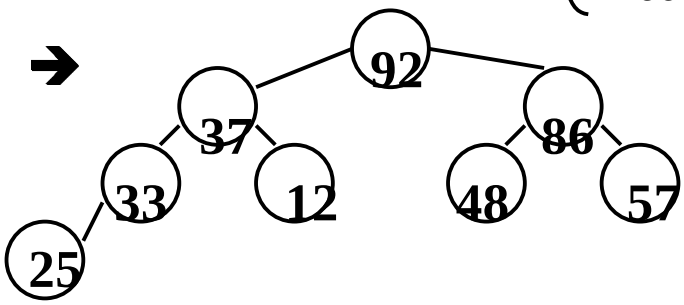
LINEARIZAÇÃO DE HEAPS

Linearizando uma árvore pode-se ter o mesmo HEAP num vetor:



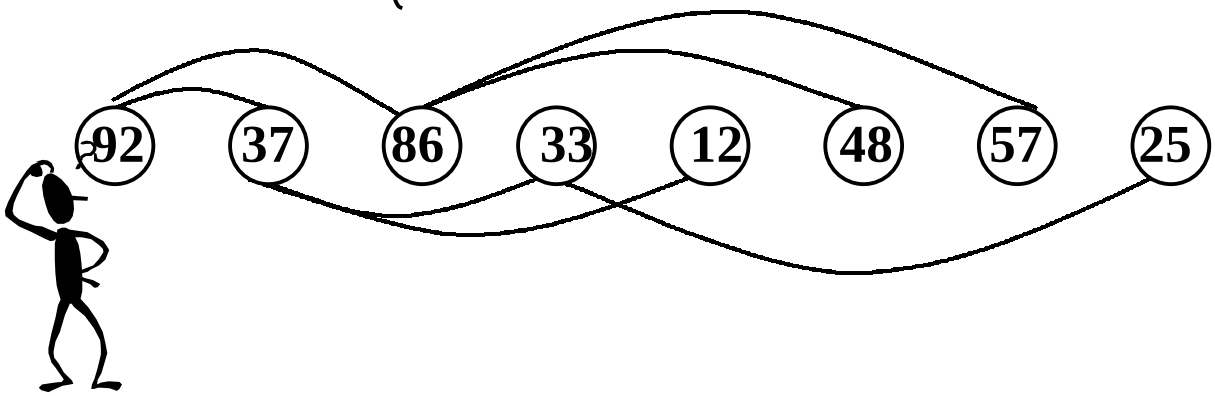
Seja a posição i no vetor:
(i×2)+ 1→ esq (i)
(i×2)+ 2→ dir (i)

Essa linearização da árvore é para implementação do HEAP no vetor e classificar *in loco*.



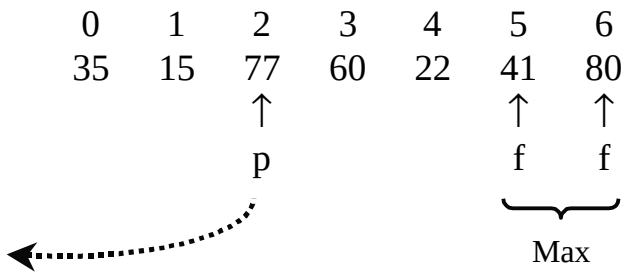
Todo vetor é um heap linearizado, exceto por infringir (eventualmente) iii)

Seja pai
Esq (pai) = pai × 2 + 1;
Dir (pai) = pai × 2 + 2;

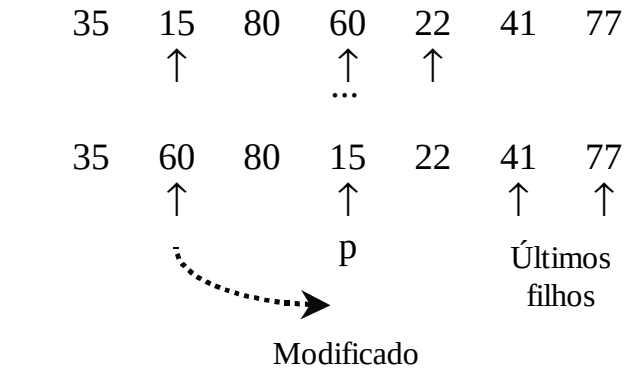


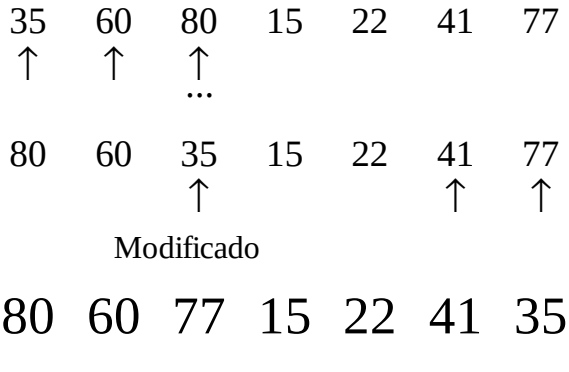
ORDENAÇÃO

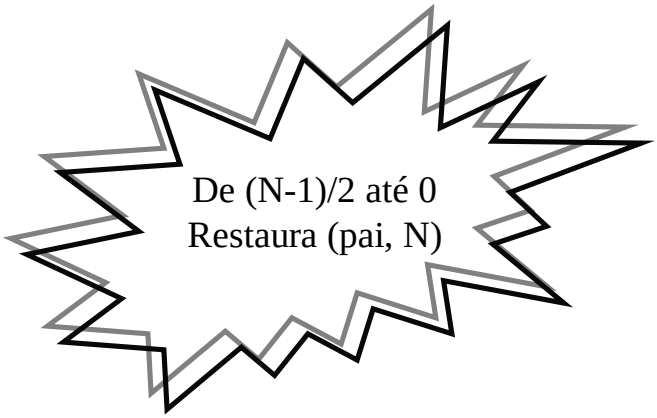
LINEARIZAÇÃO DE HEAPS



? ÚLTIMO PAI $\frac{(N-1)}{2}$

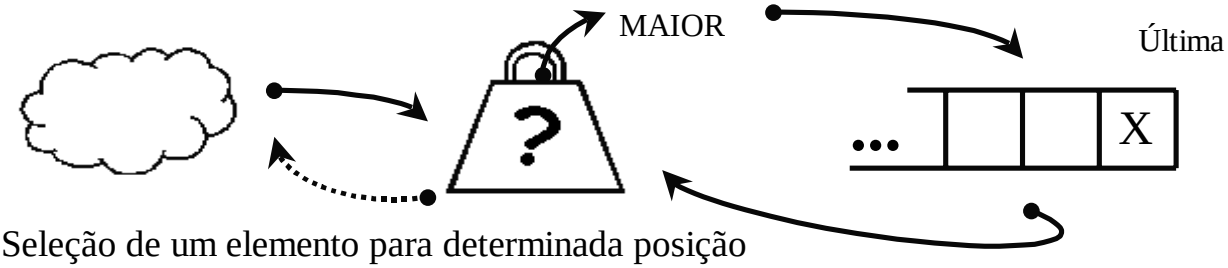






80 60 77 15 22 41 35 = HEAP

ALGORITMO HEAPSORT



Seleção de um elemento para determinada posição (ordenada) através da transformação do conjunto em heap.

ORDENAÇÃO

Exemplo:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
3	4	6	9	7	2	1	8	5
	↑						↑	↑
			↓HEAP					
3	4	6	9	7	2	1	8	5
	↑		↑	↑				
3	9	6	4	7	2	1	8	5
			?				?	?
3	9	6	8	7	2	1	4	5
↑	↑	↑						
9	3	6	8	7	2	1	4	5
	?		?	?				
9	8	6	3	7	2	1	4	5
			?				?	?
9	8	6	5	7	2	1	4	3
								?
			↓TROCA					
3	8	6	5	7	2	1	4	9

Conjunto (N-1)

… (cont.)

0	1	2	3	4	5	6	7
3	8	6	5	7	2	1	4
↑	↑	↑					
			↓RESTAURA				
8	3	6	5	7	2	1	4
	↑		↑	↑			
8	7	6	5	3	2	1	4
				↑			
			↓TROCA				
4	7	6	5	3	2	1	
↑	↑	↑					
			↓RESTAURA				
7	4	6	5	3	2	1	
	↑		↑	↑			
7	5	6	4	3	2	1	
			↑				
			↓TROCA				
1	5	6	4	3	2		
↑	↑	↑					
			↓RESTAURA				
6	5	1	4	3	2		
		↑					
6	5	2	4	3	1		

...

ORDENAÇÃO

ALGORITMOS PARA O HEAPSORT

```
RestauraHeap ( lista, pai, N )
{
        heap = falso;
        filho = 2×pai + 1;
        while ( !heap && filho < N )
        {
                maior = ( lista[filho] > lista[filho+1] ? filho : filho+1 )
                if ( lista[pai] ≥ lista[maior] )
                        heap = verdadeiro;
                else
                {
                        aux = lista[pai];
                        lista[pai] = lista[maior];
                        lista[maior] = aux;
                        pai = maior;
                        filho = pai × 2 +1;
                }
        }
}
MakeHeap ( lista, N )
{
        for ( i=(N-1)/2; i ≥ 0; i-- )
                RestauraHeap ( lista, i, N );
}
HeapSort ( lista, N )
{
        MakeHeap ( lista, N)
        for ( i = N-1; i > 0; i--)
        {
                troca ( lista[0], lista[i]);
                RestauraHeap ( lista, 0, i-1);
        }
}
```

```
Hsort ( vetor, N )
{
        para i = N/2 até 1
                AjusteHeap ( i, N );
        para i = N-1 até 1
        {
                AjusteHeap ( 1, i );
                R = vetor[i];
                Vetor[i+1] = vetor[1];
                Vetor[1] = R;
        }
}
```

ALGORITMO BÁSICO:

1. *Ajusta o vetor X a ser classificado para que ele se torne um HEAP*
2. *Troca X_0 por X_{n-1} (o maior vai para o final)*
3. *Ajusta o vetor X para HEAP de (n-1) posições*
4. *Sucessivamente*

EFICIÊNCIA:

- Caso Médio
  O (n logn) + O(n) >> quicksort O (nlogn)
- Caso Pior
  O (n logn) << quicksort O (n^2)

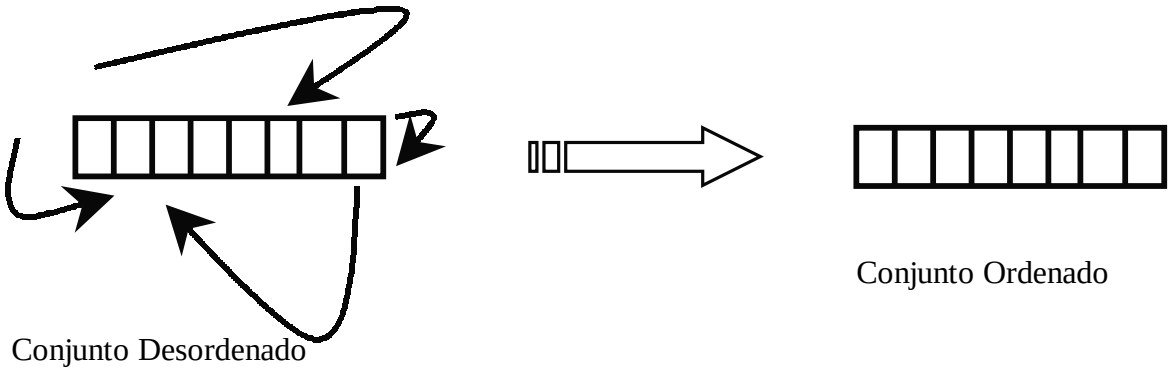
ORDENAÇÃO

7. Ordenação por Inserção

Comparado com a ordenação por seleção:

- Seleção: escolhe o elemento ordenado para a i-ésima posição;
- Inserção: insere o j-ésimo elemento na posição ordenada;

7.1 Inserção simples



Conjunto Desordenado

Conjunto Ordenado

ALGORITMO BÁSICO:

```
Insere ( X[], n )
{
        inicialmente X[0] classificado;
        faça ( k = 1; k < n; k++ )      → aumento progressivo
        {
                salva X[k] → y;
                faça ( i = k-1; i ≥ 0 && y < X[i]; i-- )  → posição onde y inserido
                        X[i+1] ← X[i];    → os maiores, uma posição para a direita
                X[i+1] ← y;  →posição a ser inserido y é X[i]
        }
}
```

EFICIÊNCIA DA INSERÇÃO SIMPLES:

- Melhor Caso: conjunto ordenado
  1 comparação para cada elemento = n
  ➔ O (n)
- Pior Caso: ordem inversa
  (n-1)+(n-2)+(n-3)+...+2+1 =
  = (n-1) × $n/2$
  ➔ O (n^2)
- Caso Médio:
  ➔ O (n^2) melhor que o bolha básico
- Melhoria: Usar busca binária para posicionamento
  ➔ O (nlogn)

- **Comparação entre os métodos já vistos:**

n	Método Apropriado
Até 30	Inserção Simples
Apartir de 30	Quicksort
Apartir de 60	Quicksort / Heapsort

7.2 Ordenação por incrementos decrescentes

Melhoria para o método de inserção que faz o pré-processamento para torná-lo o arquivo no "melhor caso" da inserção simples.

Exemplo: 25 57 48 37 12 92 86 33
Incrementos: {3, 2, 1}

(3)	subarquivo 1:	25 37 86	➔ 25 37 86
	subarquivo 2:	57 12 33	➔ 12 33 57
	subarquivo 3:	48 92	➔ 48 92

↘ 25 12 48 37 33 92 86 57

(2)	subarquivo 1:	25 48 33 86	
	subarquivo 2:	12 37 92 57	

↘ 25 12 33 37 48 57 86 92

(1)	subarquivo 1:	arquivo quase ordenado

Inserção simples opera bem com arquivos pequenos (n^2 < nlogn para n pequeno) e é de ordem O(n) na última interação. Os incrementos são em função do tamanho do arquivo, para tornar os subarquivos suficientemente pequenos. Os incrementos devem ser primos entre si. Ex.: 5, 3, 1

(*) n^2 < nlogn para n pequeno