UNIFESP
UNIVERSIDADE FEDERAL DE SÃO PAULO
1933

Métodos de Ordenação n log n

Prof. Lilian Berton

São José dos Campos, 2018

Algoritmos de ordenação em memória interna

- **Quadráticos $O(n^2)$:**
  - Ordenação por Seleção (Selection Sort)
  - Ordenação por Inserção (Inserction Sort)
  - Ordenação por Bolha (BubbleSort)

- **O(n log n):**
  - Shellsort
  - Quicksort
  - Heapsort

- **Lineares O(n):**
  - Ordenação por contagem
  - Radix sort
  - Bucket sort

ShellSort

- Os itens separados de h posições são rearranjados.
- Todo h-ésimo item leva a uma sequência ordenada. Tal sequência é dita estar h-ordenada.
- Quando h = 1 Shellsort corresponde ao algoritmo de inserção.

- Uma sequência para h segundo Knuth corresponde a

1, 4, 13, 40, 121, 364, 1.093, 3.280, . . .

- O valor ótimo para h não é conhecido.

Exemplo ordenação Shellsort

h = 3

0	1	2	3	4	5	6	7	8	9
22	5	9	1	15	3	14	8	19	10

i= 3
compara 22 e 1
Troca!

1	5	9	22	15	3	14	8	19	10

i= 4
compara 15 e 5

1	5	9	22	15	3	14	8	19	10

i= 5
compara 3 e 9
Troca!

1	5	3	22	15	9	14	8	19	10

i= 6
compara 14 e 22
Troca!
compara 14 e 1

1	5	3	14	15	9	22	8	19	10

i= 7
compara 8 e 15
Troca!
Compara 8 e 5

Exemplo ordenação Shellsort

h = 3

0	1	2	3	4	5	6	7	8	9
1	5	3	14	8	9	22	15	19	10

i= 8
Compara 19 e 9
Compara 9 e 3

1	5	3	14	8	9	22	15	19	10

i= 9
Compara 10 e 22
Troca!
Compara 10 e 14
Troca!
Compara 10 e 1

1	5	3	10	8	9	14	15	19	22

i = n -> finaliza h = 3 e decrementa h

Exemplo ordenação Shellsort

h = 2

0	1	2	3	4	5	6	7	8	9
1	5	3	10	8	9	14	15	19	22

i= 2
compara 3 e 1

1	5	3	10	8	9	14	15	19	22

i= 3
compara 10 e 5

1	5	3	10	8	9	14	15	19	22

i=4
compara 8 e 3
Compara 3 e 1

1	5	3	10	8	9	14	15	19	22

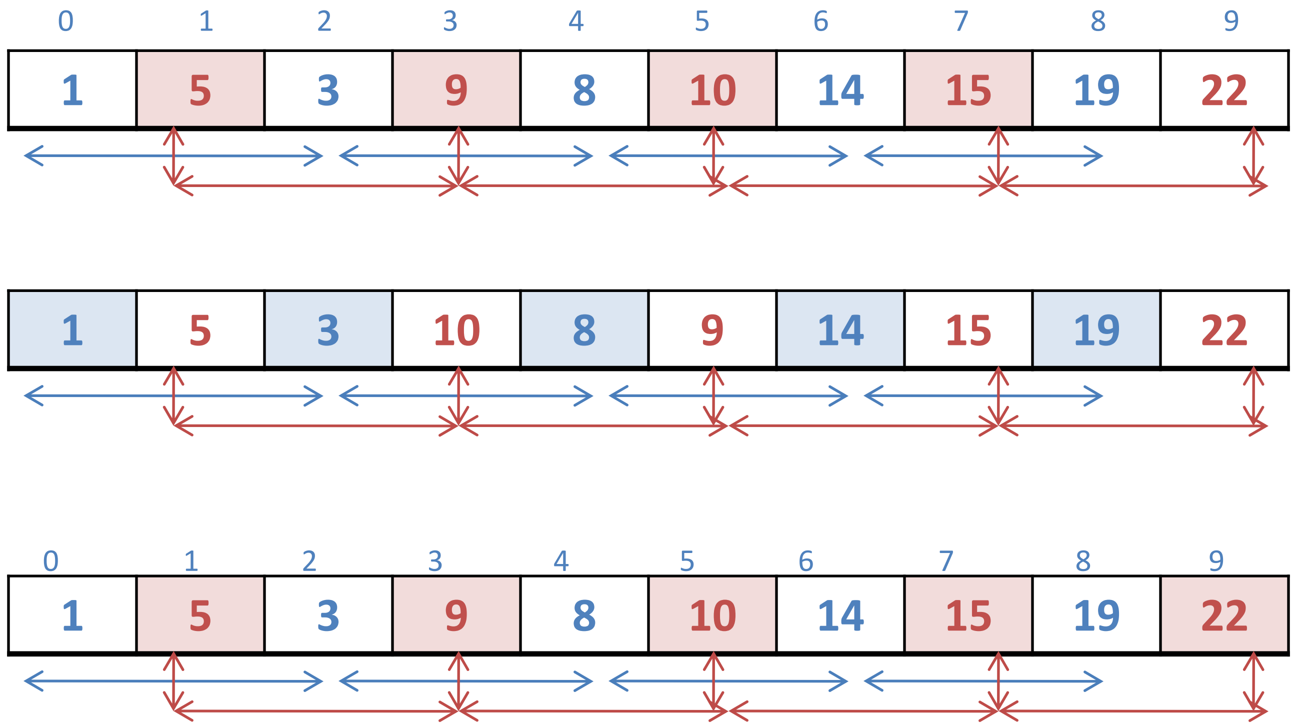
i=5
compara 9 e 10
Troca!
Compara 9 e 5

1	5	3	9	8	10	14	15	19	22

i=6
compara 14 e 8
Compara 8 e 3
Compara 3 e 1

Exemplo ordenação Shellsort

h = 2



i=7
compara 15 e 10
Compara 10 e 9
Compara 9 e 5

i= 8
compara 19 e 14
Compara 14 e 8
Compara 8 e 3
Compara 3 e 1

i=9
Compara 22 e 15
compara 15 e 10
Compara 10 e 9
Compara 9 e 5

i = n -> finaliza h = 2 e decrementa h

Exemplo ordenação Shellsort

Inserção

h =1

0	1	2	3	4	5	6	7	8	9
1	5	3	9	8	10	14	15	19	22

i = 1
Compara 5 e 1

1	5	3	9	8	10	14	15	19	22

i = 2
Compara 3 e 5
Troca!

1	3	5	9	8	10	14	15	19	22

i = 3
Compara 9 e 5

1	3	5	9	8	10	14	15	19	22

i = 4
Compara 8 e 9
Troca!

1	3	5	8	9	10	14	15	19	22

i = 5
Compara 10 e 9

1	3	5	8	9	10	14	15	19	22

i = 6
Compara 14 e 10

Exemplo ordenação Shellsort

Inserção

h =1

0	1	2	3	4	5	6	7	8	9
1	**3**	**5**	**8**	**9**	**10**	**14**	**15**	**19**	**22**

i = 7
Compara 15 e 14

1	**3**	**5**	**8**	**9**	**10**	**14**	**15**	**19**	**22**

i = 8
Compara 19 e 15

1	**3**	**5**	**8**	**9**	**10**	**14**	**15**	**19**	**22**

i = 9
Compara 22 e 19

i = n -> finaliza h = 1 e finaliza algoritmo!

Algoritmo

```
inc[3]={3,2,1};
```

Vetor com os valores para h

```
void shellsort(int v[], int n, int incrementos[], int numinc) {
        int incr, i, j, h, aux;
        for (incr=0; incr<numinc; incr++) {
                h=incrementos[incr];
                for (i=h; i<n; i++) {
                        aux=v[i];
                        for (j=i-h; j>=0 && v[j]>aux; j-=h)
                                v[j+h]=v[j];
                        v[j+h]=aux;
                }
        }
}
```

Análise

- Ninguém ainda foi capaz de analisar o algoritmo. A sua análise contém alguns problemas matemáticos muito difíceis.

- Conjecturas referente ao número de comparações para a sequência de Knuth:
  - Conjetura 1 : $C(n) = O(n^{1,25})$
  - Conjetura 2 : $C(n) = O(n(\ln n)^2)$

- **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.

Algoritmos de ordenação em memória interna

- **Quadráticos $O(n^2)$:**
  - Ordenação por Seleção (Selection Sort)
  - Ordenação por Inserção (Inserction Sort)
  - Ordenação por Bolha (BubbleSort)

- **O(n log n):**
  - Shellsort
  - Quicksort
  - Heapsort

- **Lineares O(n):**
  - Ordenação por contagem
  - Radix sort
  - Bucket sort

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], . . . , A[j] são menores ou iguais a x.
  - Os itens em A[i], A[i + 1], . . . , A[Dir] são maiores ou iguais a x.

Dividir para conquistar

- **Dividir**: O arranjo A[e . . d] é particionado (reorganizado) em dois subarranjos (possivelmente vazios) A[e . . m – 1] e A[m + 1 . . d] tais que cada elemento de A[e . . m – 1] é menor que ou igual a A[m] que, por sua vez, é igual ou menor a cada elemento de A[m + 1 . . d].
- O índice m é calculado como parte desse procedimento de particionamento.

- **Conquistar**: Os dois subarranjos A[e .. m - 1] e A[m + 1 .. d] são ordenados por chamadas recursivas a quicksort.

- **Combinar**: Como os subarranjos são ordenados localmente, não é necessário nenhum trabalho para combiná-los: o arranjo A[e.. d] inteiro agora está ordenado.

Exemplo ordenação por quicksort

i **Pivô = i+j/2** **j**

0	1	2	3	4	5	6	7	8	9
22	**5**	**9**	**1**	**15**	**3**	**14**	**8**	**19**	**10**

1) i = 0 e j = 9; 22 > 15 e 10 < 15 então troca!

0	1	2	3	4	5	6	7	8	9
10	**5**	**9**	**1**	**15**	**3**	**14**	**8**	**19**	**22**

2) i = 1 e j = 9; 5 < 15 então não troca!

3) i = 2 e j = 9; 9 < 15 então não troca!

4) i = 3 e j = 9; 1 < 15 então não troca!

5) i = 4 e j = 9; 15 = 15 então não troca!

6) i = 4 e j = 8; 15 = 15 e 19 > 15 então não troca!

7) i = 4 e j = 7; 15 = 15 e 8 < 15 então troca!

0	1	2	3	4	5	6	7	8	9
10	**5**	**9**	**1**	**8**	**3**	**14**	**15**	**19**	**22**

8) i = 5 e j = 7; 3 < 15 e 15 = 15 então não troca!

8) i = 6 e j = 7; 14 < 15 e 15 = 15 então não troca!

9) i = 7 e j = 7; índices se encontraram, finaliza!

Escolha ruim de pivô!!

Exemplo ordenação por quicksort

i Pivô = i+j/2 j

0	1	2	3	4	5	6	7	8	9
10	**5**	**9**	**1**	**8**	**3**	**14**	**15**	**19**	**22**

1) i = 0 e j = 6; 10 > 1 e 14 > 1 então não troca!

2) i = 0 e j = 5 ; 10 > 1 e 3 > 1 então não troca!

3) i = 0 e j = 4; 10 > 1 e 8 > 1 então não troca!

4) i = 0 e j = 3; 10 > 1 e 1 = 1 então troca!

0	1	2	3	4	5	6	7	8	9
1	**5**	**9**	**10**	**8**	**3**	**14**	**15**	**19**	**22**

5) i = 1 e j = 3; 5 > 1 e 10 > 1 então não troca!

6) i = 2 e j = 3; 9 > 1 e 10 > 1 então não troca!

7) i = 3 e j = 3; índices se encontraram, finaliza!

1	**5**	**9**	**10**	**8**	**3**	**14**	**15**	**19**	**22**

Particionou o subvetor em 1 e n-1 partes!

Escolha boa de pivô!!

Exemplo ordenação por quicksort

i		pivô				j			
0	1	2	3	4	5	6	7	8	9
10	5	9	1	8	3	14	15	19	22

1) i = 0 e j = 6; 10 > 9 e 14 > 9 então não troca!

2) i = 0 e j = 5 ; 10 > 9 e 3 < 9 então troca!

0	1	2	3	4	5	6	7	8	9
3	5	9	1	8	10	14	15	19	22

3) i = 1 e j = 5; 5 < 9 e 10 > 9 então não troca!

4) i = 2 e j = 5; 9 = 9 e 10 < 9 então não troca!

5) i = 2 e j = 4; 9 = 9 e 8 < 9 então troca!

0	1	2	3	4	5	6	7	8	9
3	5	8	1	9	10	14	15	19	22

7) i = 3 e j = 4; 1 < 9 e 9 = 9 então não troca!

8) i = 4 e j = 4; índices se encontraram, finaliza!

3	5	8	1	9	10	14	15	19	22

Particionou o subvetor em duas partes semelhantes!

Escolha boa de pivô!!

Exemplo ordenação por quicksort

i pivô j

0	1	2	3	4	5	6	7	8	9
3	**5**	**8**	**1**	**9**	**10**	**14**	**15**	**19**	**22**

17) i = 0 e j = 3; 3 < 5 e 1 < 5 então não troca!

18) i = 1 e j = 3; 5 = 5 e 1 < 5 então troca!

0	1	2	3	4	5	6	7	8	9
3	**1**	**8**	**5**	**9**	**10**	**14**	**15**	**19**	**22**

19) i = 2 e j = 3; 8 > 5 e 5 = 5 então troca!

0	1	2	3	4	5	6	7	8	9
3	**1**	**8**	**5**	**9**	**10**	**14**	**15**	**19**	**22**

20) i = 3 e j = 3; índices se encontraram, finaliza!

3	**1**	**8**	**5**	**9**	**10**	**14**	**15**	**19**	**22**

Após chamar quicksort para cada subvetor, temos o vetor ordenado!

1	**3**	**5**	**8**	**9**	**10**	**14**	**15**	**19**	**22**

Algoritmo quicksort

```
procedure QuickSort (var A: Vetor; var n: Indice);
{-- Entra aqui o procedimento Particao --}
  procedure Ordena (Esq, Dir: Indice);
  var i, j: Indice;
  begin
    particao (Esq, Dir, i, j);
    if Esq < j then Ordena (Esq, j);
    if i < Dir then Ordena (i, Dir);
  end;
begin
  Ordena (1, n);
end;
```

O(log n)
O(n)

```
procedure Particao (Esq, Dir: Indice;var i,j: Indice);
var x, w: Item;
begin
  i := Esq; j := Dir;
  x := A[(i + j) div 2]; { obtem o pivo x }
  repeat
    while x.Chave > A[i].Chave do i := i + 1;
    while x.Chave < A[j].Chave do j := j - 1;
    if i <= j
    then begin
           w := A[i]; A[i] := A[j]; A[j] := w;
           i := i + 1; j := j - 1;
         end;
  until i > j;
end;
```

O(n)

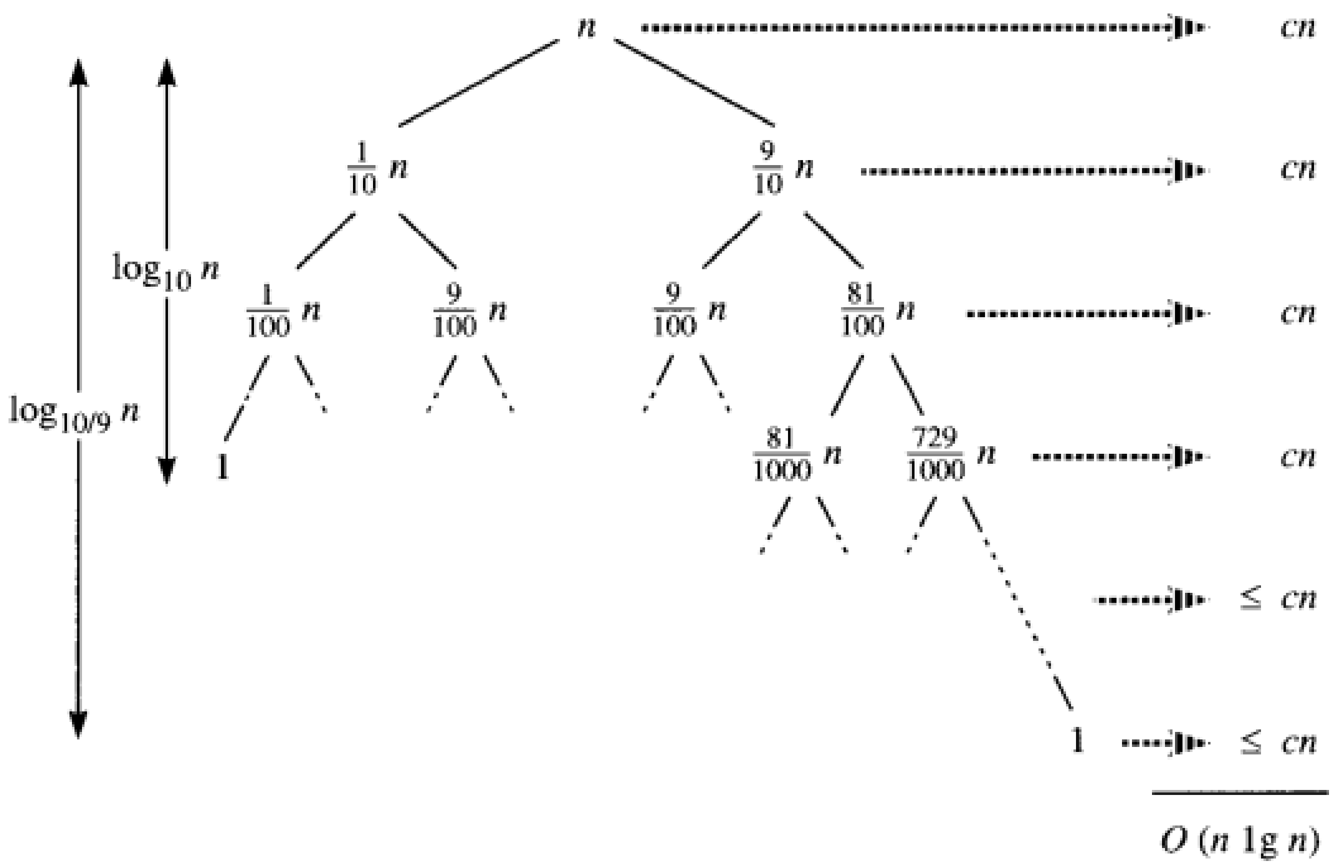
Análise da complexidade

- **Pior caso: $C(n) = O(n^2)$**
  - O pior caso ocorre quando 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 particionamento custa o tempo O(n)
  - Para evitar isso basta escolher três itens quaisquer do vetor e usar a mediana dos três como pivô.
  - $T(n) = T(n-1) + O(n) = O(n^2)$

- **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.
  - $T(n) <= 2T(n/2) + O(n) = O(n \log n)$

- **Caso médio = n log n**
  - Um arranjo de entrada aleatório, é improvável que o particionamento sempre ocorra do mesmo modo em todo nível. Esperamos que algumas divisões sejam razoavelmente bem equilibra-das e que algumas sejam bastante desequilibradas.

Caso médio



Obtida em Cormen et al.

Vantagens e desvantagens

- Vantagens:
  - É eficiente para ordenar arquivos de dados aleatórios.
  - 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 mais delicada e difícil.

- O método não é estável.

Algoritmos de ordenação em memória interna

- **Quadráticos $O(n^2)$:**
  - Ordenação por Seleção (Selection Sort)
  - Ordenação por Inserção (Inserction Sort)
  - Ordenação por Bolha (BubbleSort)

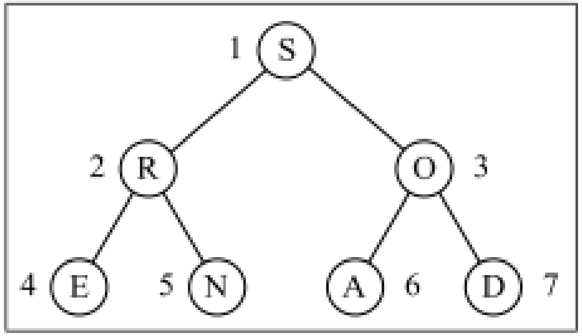
- **O(n log n):**
  - Shellsort
  - Quicksort
  - Heapsort

- **Lineares O(n):**
  - Ordenação por contagem
  - Radix sort
  - Bucket sort

Heap sort

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

- 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 sort

- 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], . . . , 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.

A melhor representação é através de **uma estruturas de dados chamada heap**:

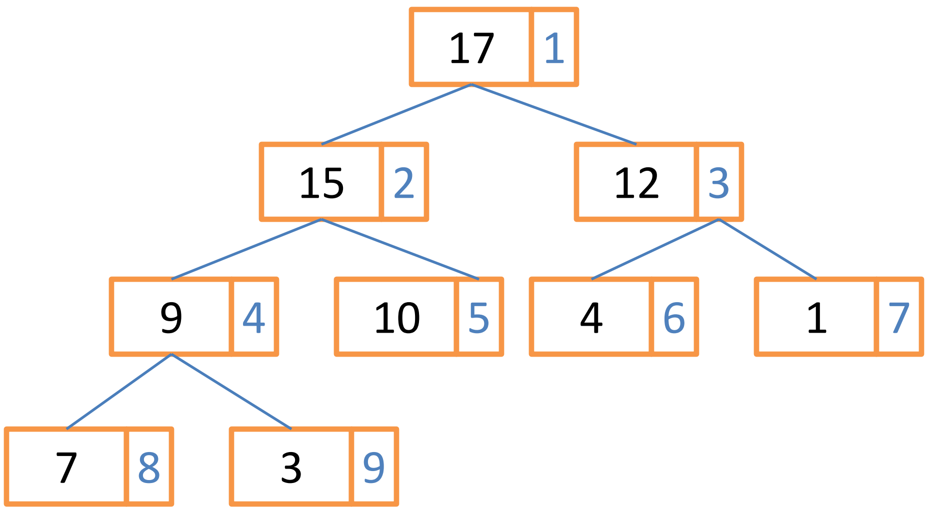
– Neste caso, Constrói é O(n).

– Insere, Retira, Substitui e Altera são O(log n).

- Logo, Heapsort gasta um tempo de execução proporcional a **n log n**, no pior caso.

Heap

- É **uma estrutura de árvore binária** em que cada nó tem uma prioridade maior ou igual a do seus filhos.
- O nó **i** possui dois filhos nas posições: **2i e 2i+1**.
- Custo para inserir e remover é sempre log(n).

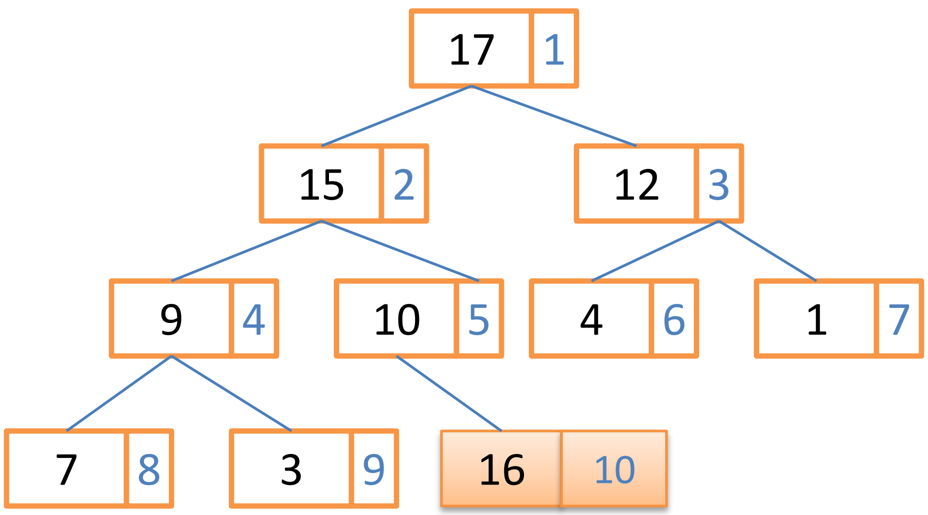


1	2	3	4	5	6	7	8	9
17	15	12	9	10	4	1	7	3

i	2i	2i+1
1	2	3
2	4	5
3	6	7
4	8	9

Heap - Inserção

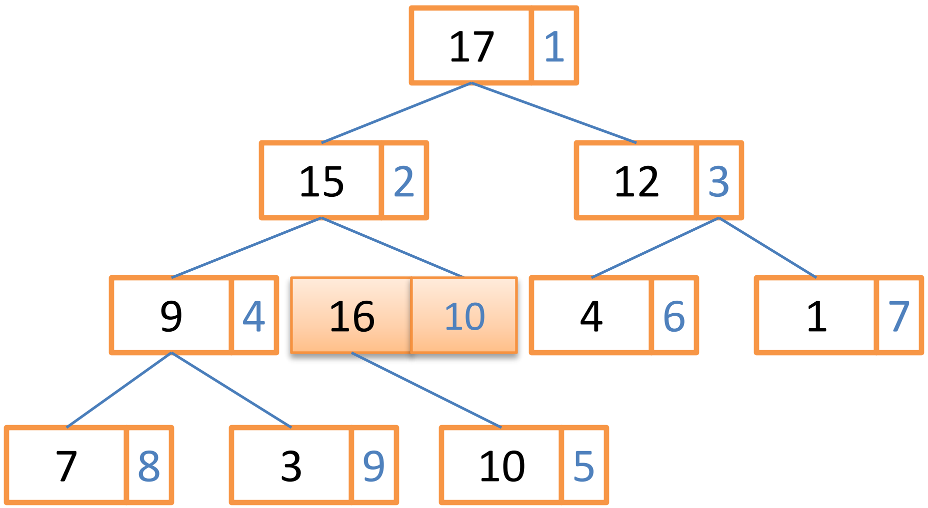
- Para **inserir um novo elemento**, cria-se uma nova posição no fim do array. Se o elemento for maior que seu pai, os dois trocam de lugar. Repete-se esse passo até que ele esteja no local correto.



1	2	3	4	5	6	7	8	9	
17	15	12	9	10	4	1	7	3	16

Heap - Inserção

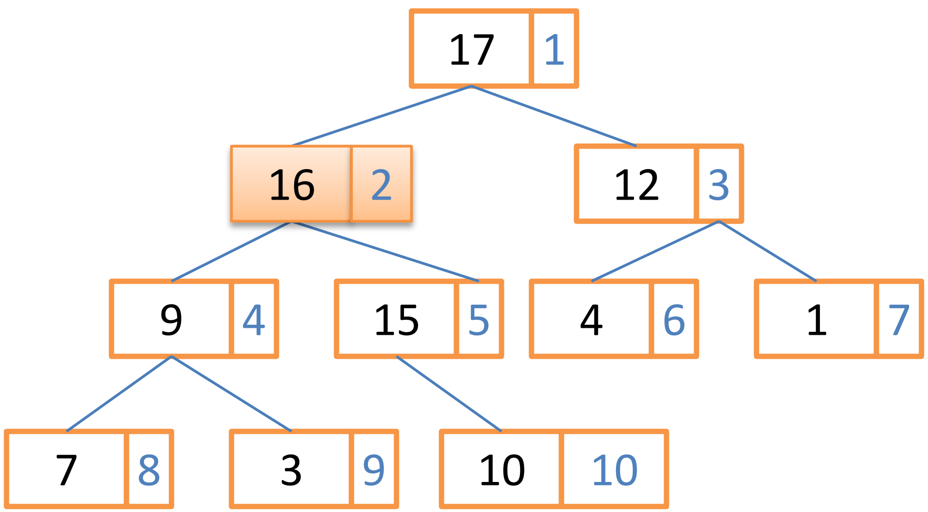
- Para inserir um novo elemento, cria-se uma nova posição no fim do array. Se x for maior que seu pai, os dois trocam de lugar. Repete-se esse passo até que x esteja no local correto.



1	2	3	4	5	6	7	8	9	10
17	15	12	9	16	4	1	7	3	10

Heap - Inserção

- Para inserir um novo elemento, cria-se uma nova posição no fim do array. Se x for maior que seu pai, os dois trocam de lugar. Repete-se esse passo até que x esteja no local correto.



1	2	3	4	5	6	7	8	9	10
17	16	12	9	15	4	1	7	3	10

Implementação

```
void insere(int x, int Heap [], int ult) {
    ult = ult+1;  //ultimo índice do heap antes da inserção (no ex. ult = 9+1=10)
    int i = ult;

    while ((i/2) && (x > Heap[i/2])) {
        Heap[i] = Heap [i/2];
        i = i/2;
    }
    Heap[i] = x;
}
```

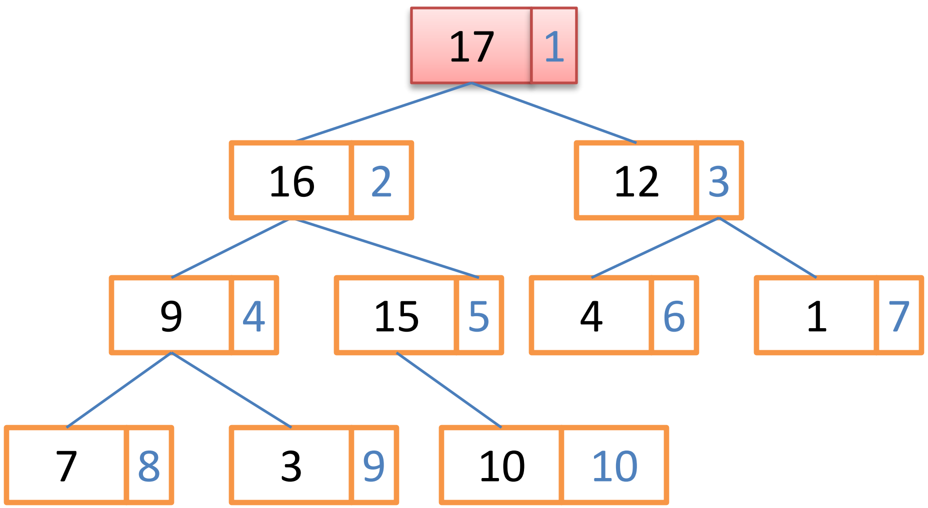
Heap - Remoção

- O elemento a ser removido é sempre o de maior prioridade Heap[1].

- Após a remoção os elementos precisam ser re-arranjados:
  – Colocar em Heap[1] o elemento Heap[ult] e liberar Heap[ult];
  – Se o novo elemento em Heap[1] for menor que seus filhos, então trocar com o maior dos filhos;
  – Repetir esse passo até o elemento ocupar a posição correta.

Heap - Remoção

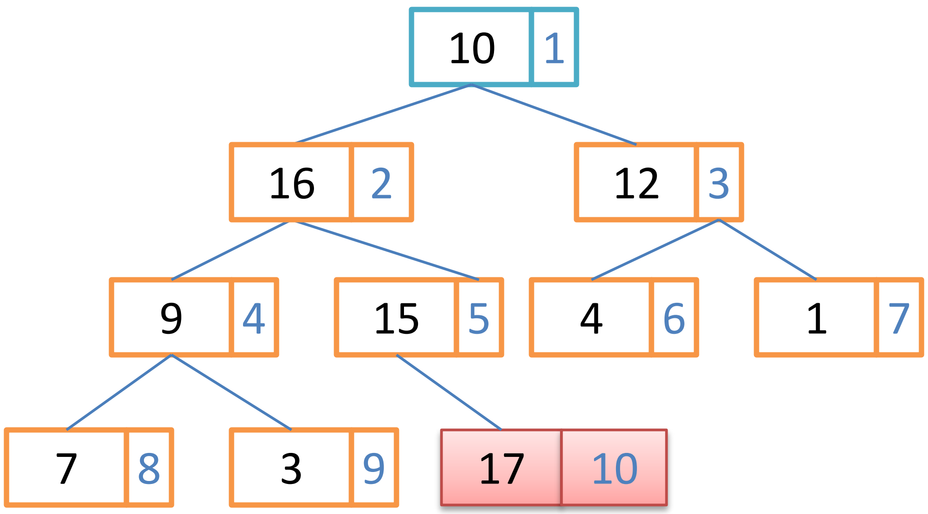
- O elemento a ser removido é sempre o de maior prioridade Heap[1].



1	2	3	4	5	6	7	8	9	10
17	16	12	9	15	4	1	7	3	10

Heap - Remoção

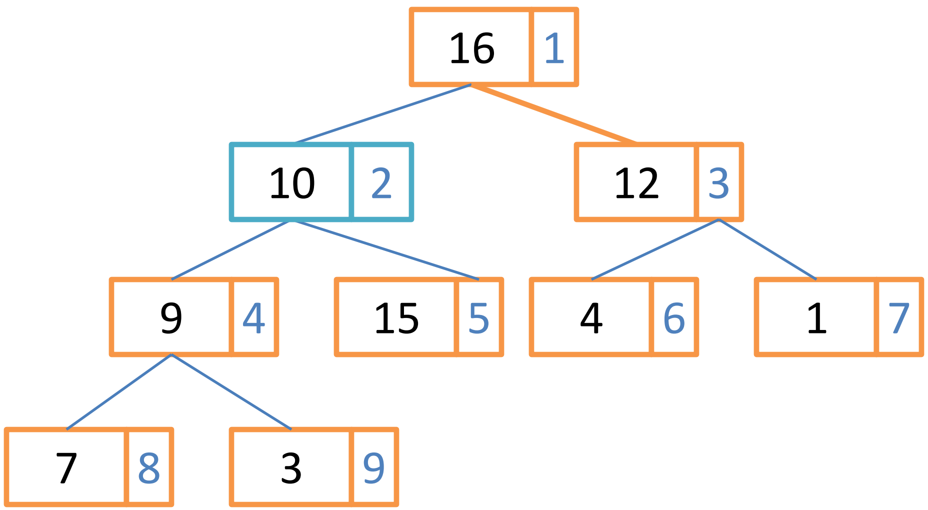
– Colocar em Heap[1] o elemento Heap[ult] e liberar Heap[ult];



1	2	3	4	5	6	7	8	9	10
10	16	12	9	15	4	1	7	3	17

Heap - Remoção

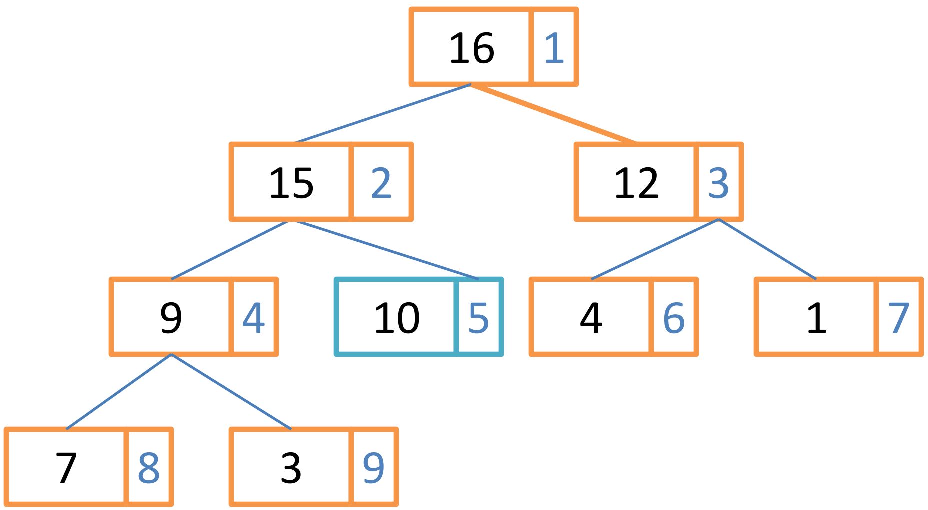
– Se o novo elemento em Heap[1] for menor que seus filhos, então trocar com o maior dos filhos;



1	2	3	4	5	6	7	8	9
16	10	12	9	15	4	1	7	3

Heap - Remoção

– Repetir esse passo até o elemento ocupar a posição correta.



1	2	3	4	5	6	7	8	9
16	15	12	9	10	4	1	7	3

Implementação

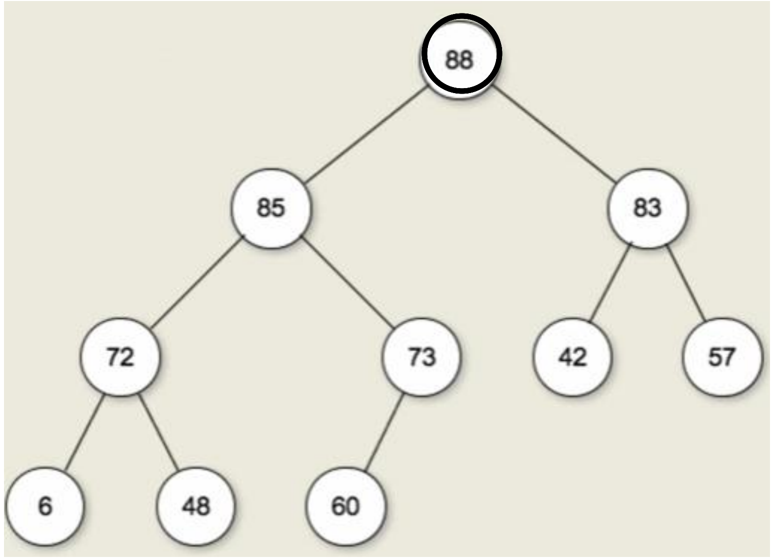
```
void remover(int Heap[], int ult) {
    int remove = Heap[1];
    int x = Heap[ult];
    ult = ult-1;
    int i = 1;
    While ((2i <= ult) && (x < Heap[2i] || x  < Heap[2i+1])) {
        if(Heap[2i] > Heap[2i+1]) {
            Heap[i] = Heap[2i];
            i = 2i;
        } else {
            Heap[i] = Heap[2i+1];
            i=2i+1;
        }
    }
    Heap[i] = x;
}
```

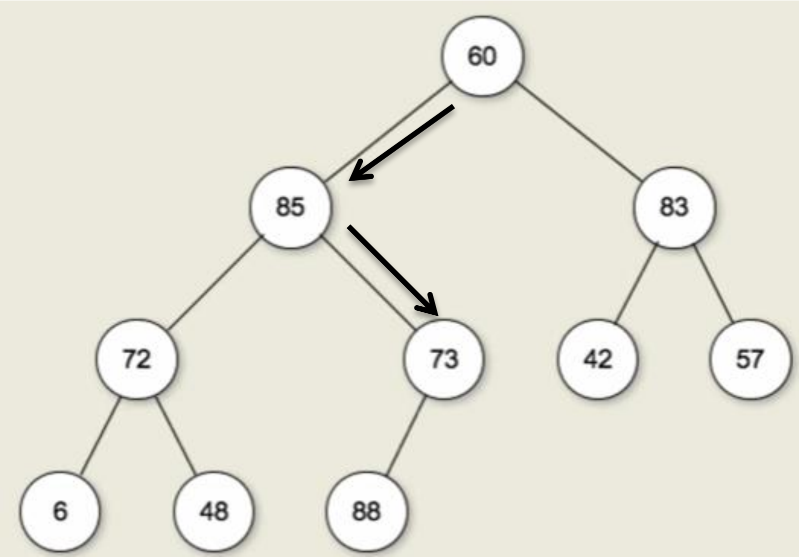
https://www.youtube.com/watch?v=D_B3HN4gcUA

Exemplo Heap sort

88	**85**	**83**	**72**	**73**	**42**	**57**	**6**	**48**	**60**

60	**85**	**83**	**72**	**73**	**42**	**57**	**6**	**48**	**88**

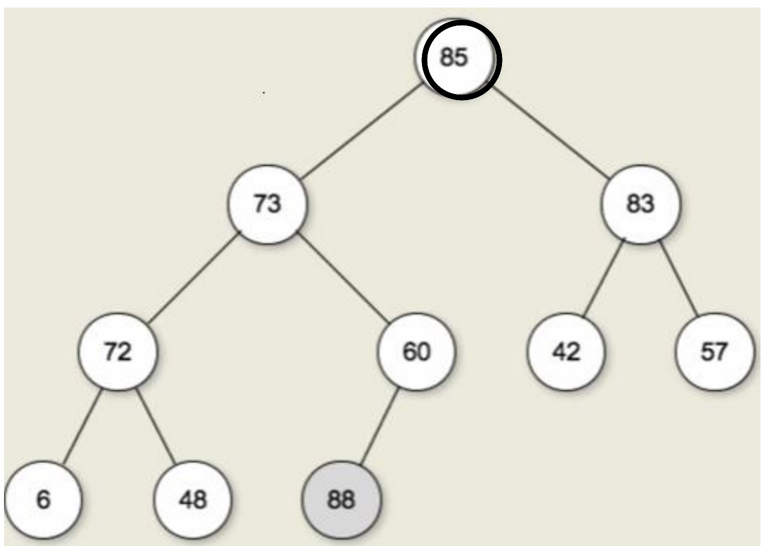


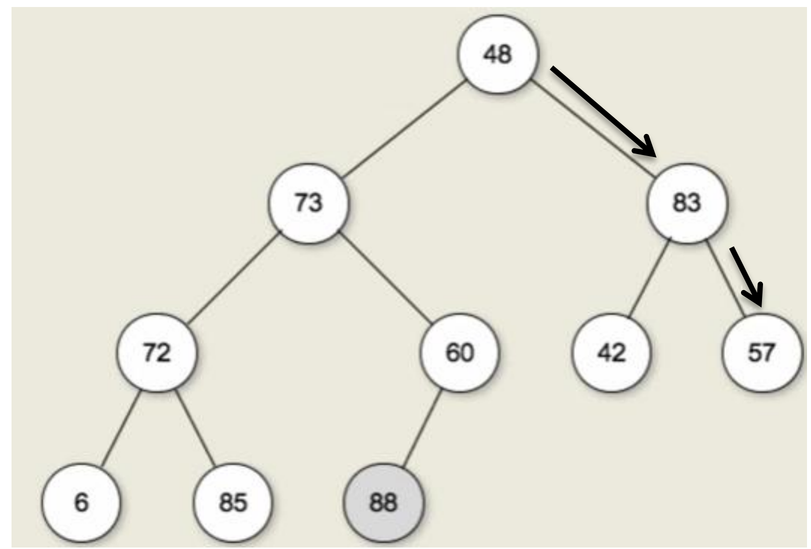


Exemplo Heap sort

85	73	83	72	60	42	57	6	48	88

48	73	83	72	60	42	57	6	85	88

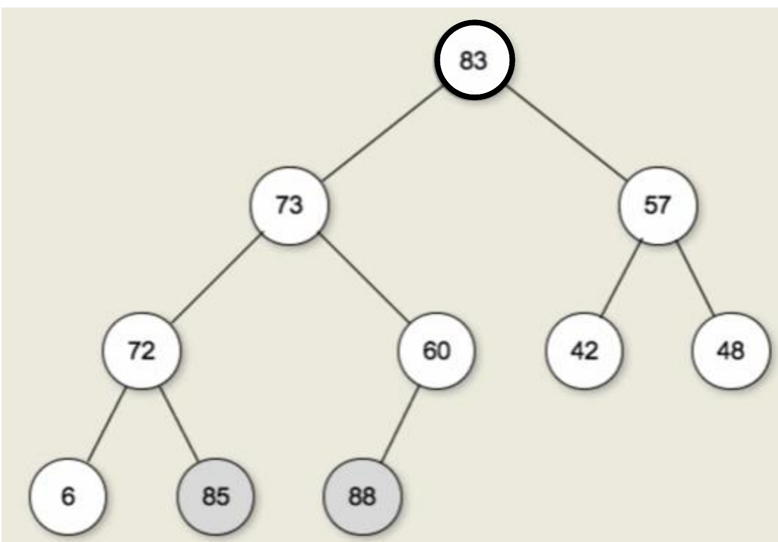


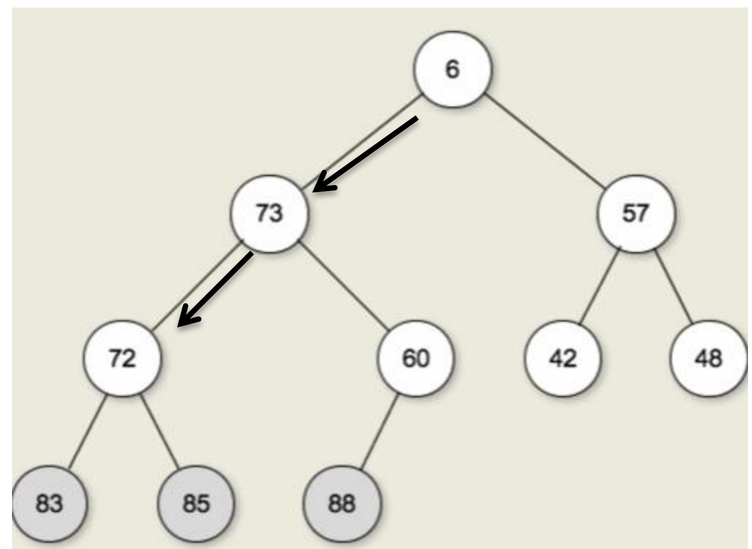


Exemplo Heap sort

83	73	57	72	60	42	48	6	85	88

6	73	57	72	60	42	48	83	85	88

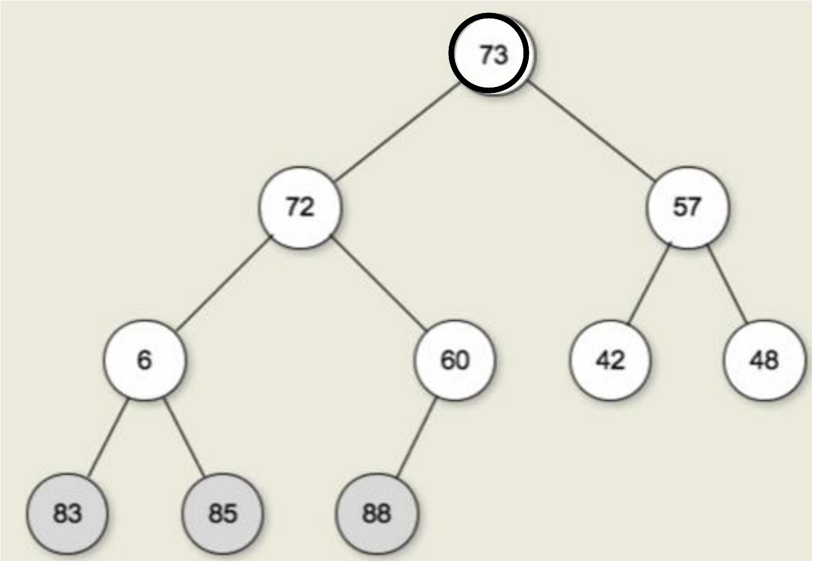


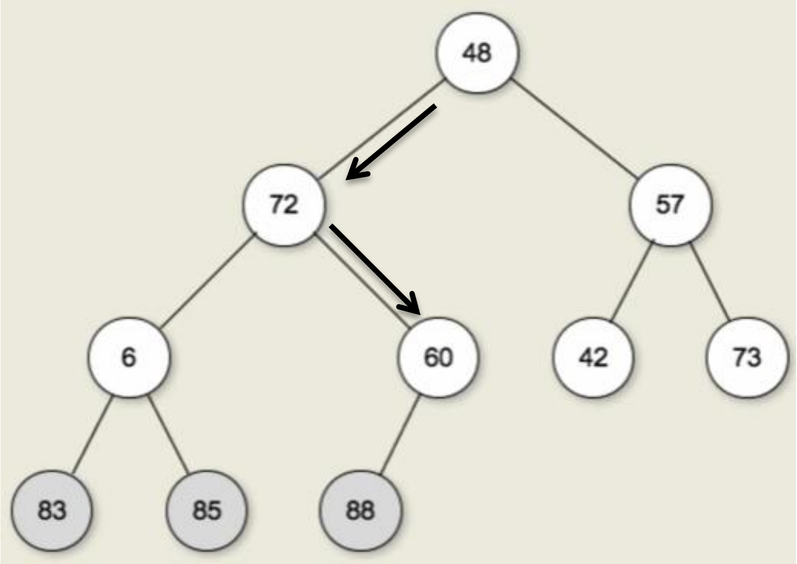


Exemplo Heap sort

73	**72**	**57**	**6**	**60**	**42**	**48**	**83**	**85**	**88**

48	**72**	**57**	**6**	**60**	**42**	**73**	**83**	**85**	**88**

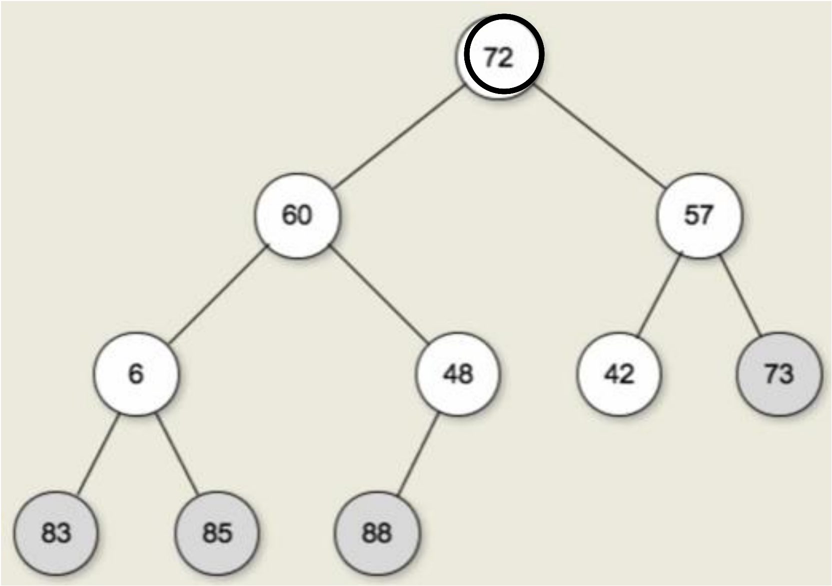


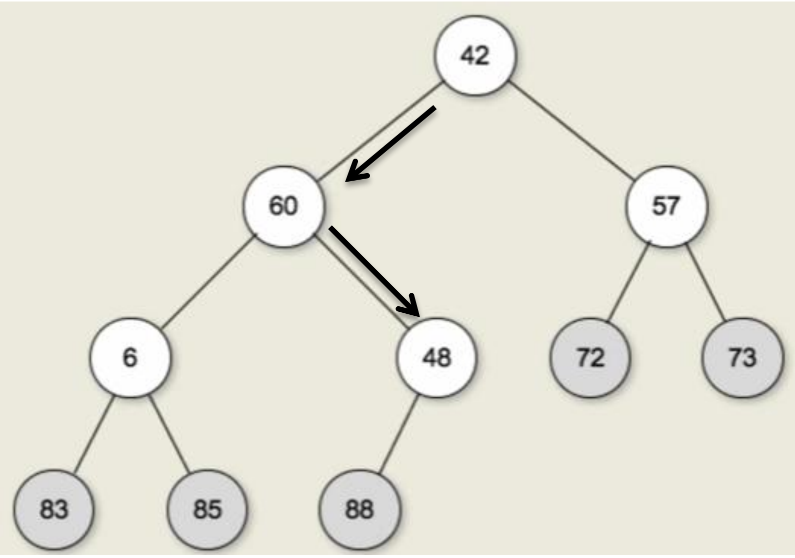


Exemplo Heap sort

72	60	57	6	48	42	73	83	85	88

42	60	57	6	48	72	73	83	85	88

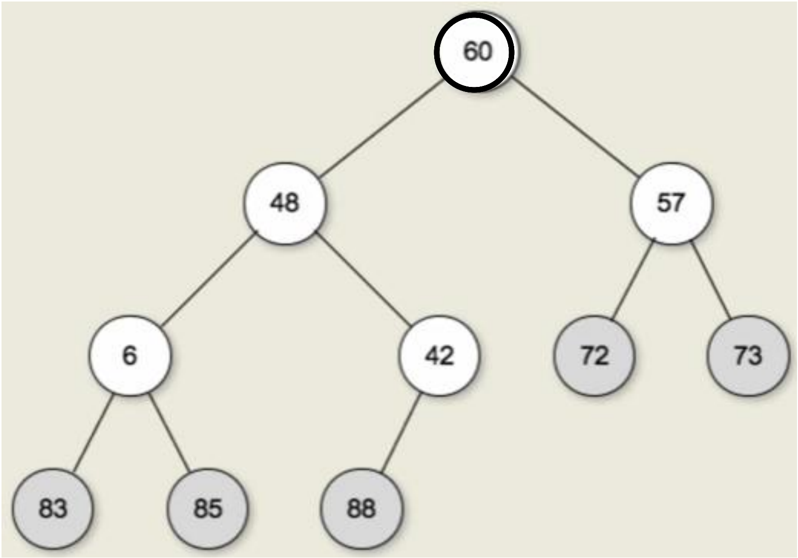


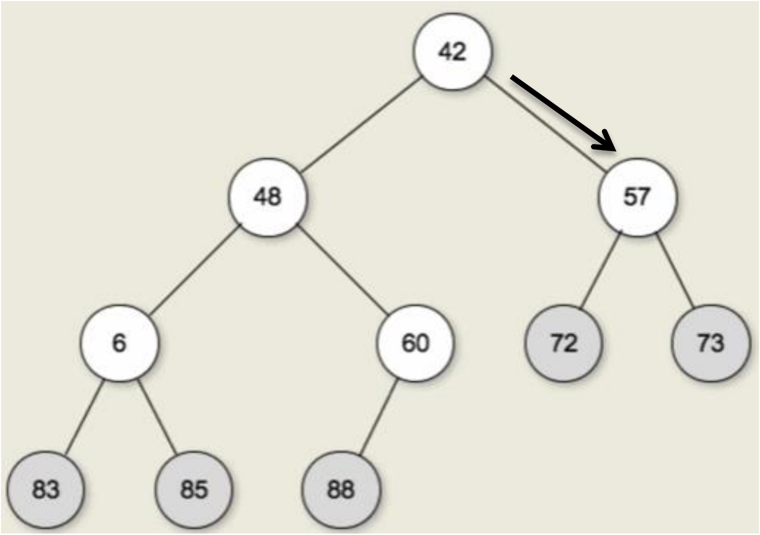


Exemplo Heap sort

60	48	57	6	42	72	73	83	85	88

42	48	57	6	60	72	73	83	85	88

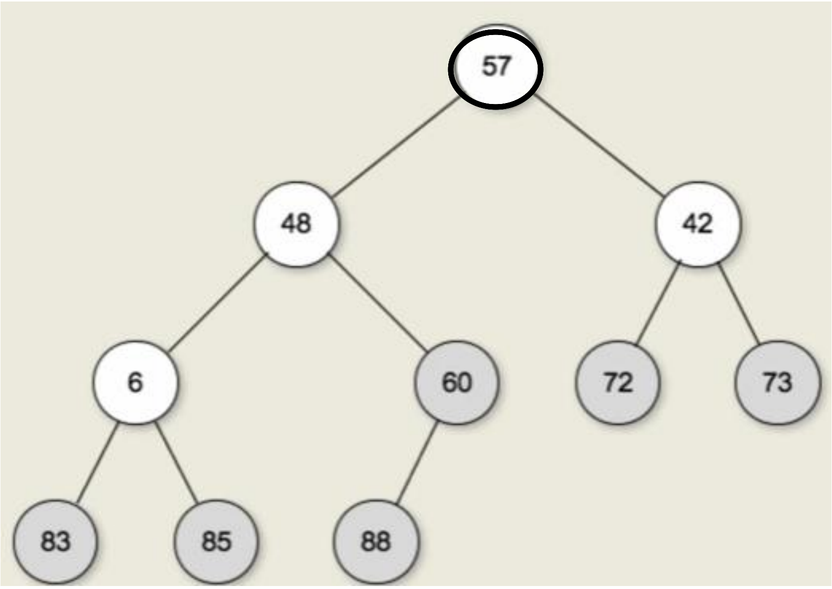


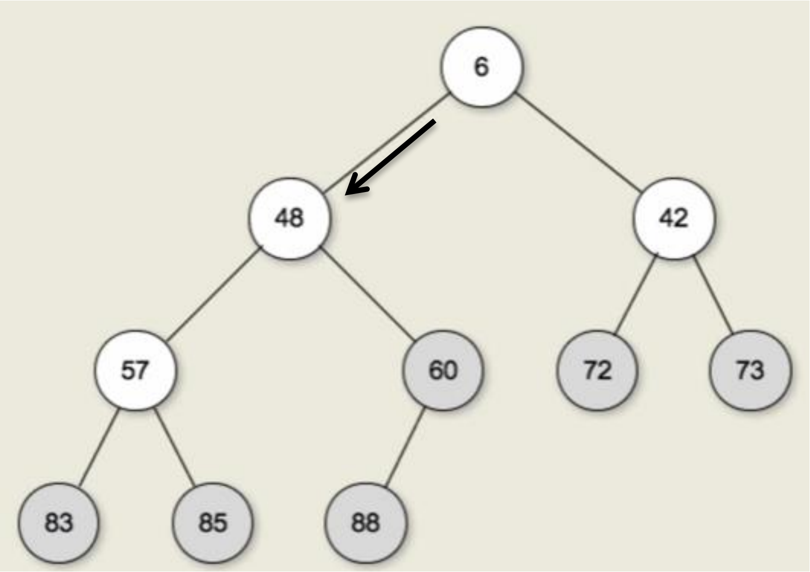


Exemplo Heap sort

42	**48**	**57**	**6**	**60**	**72**	**73**	**83**	**85**	**88**

6	**48**	**42**	**57**	**60**	**72**	**73**	**83**	**85**	**88**

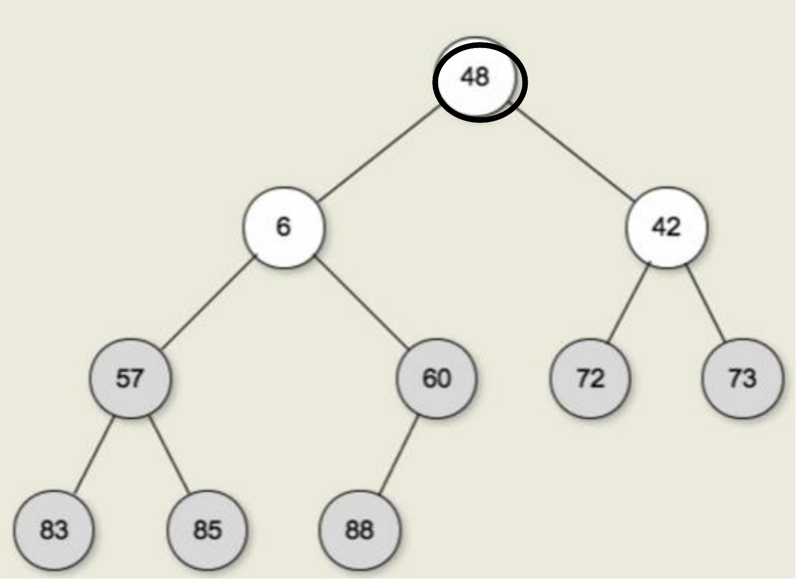


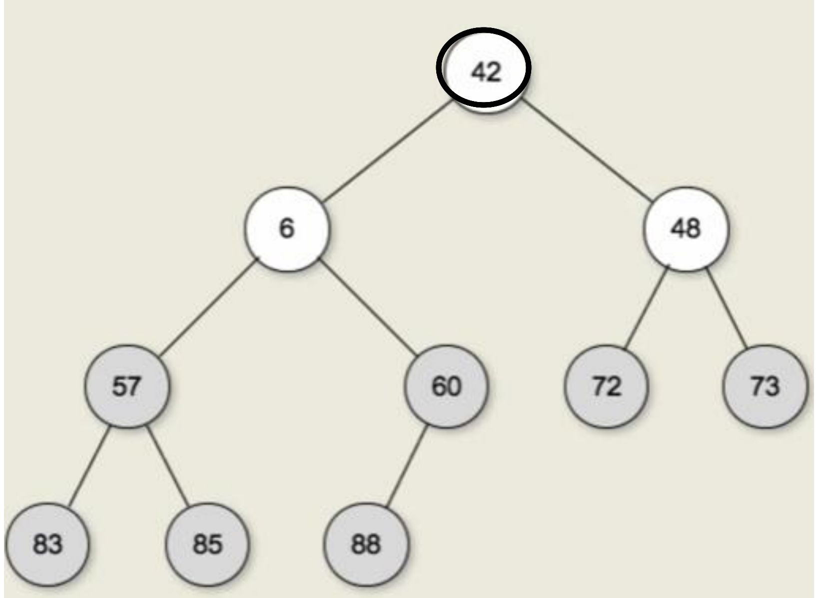


Heap sort

6	48	42	57	60	72	73	83	85	88

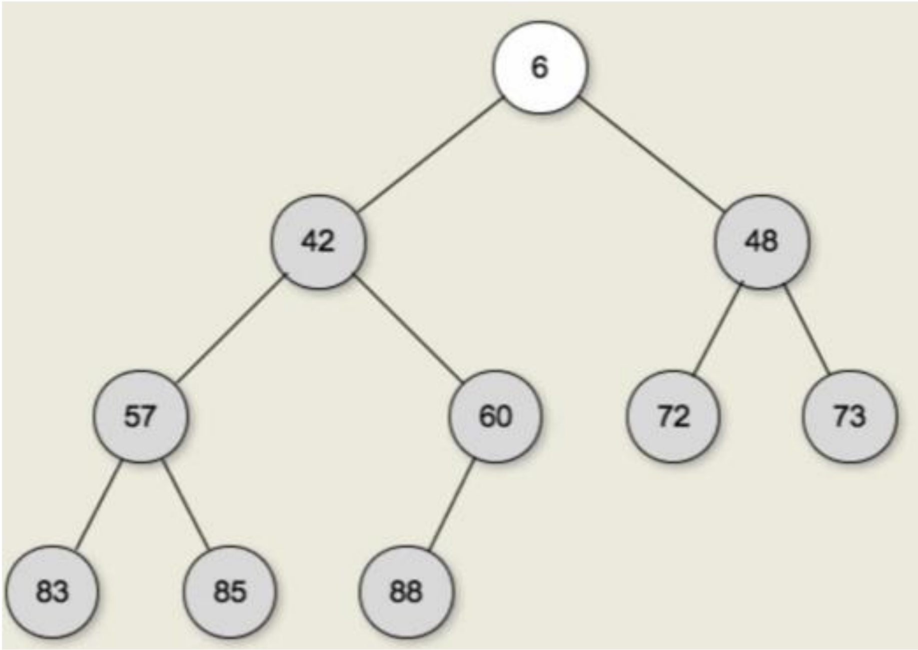
6	42	48	57	60	72	73	83	85	88





Heap sort

6	42	48	57	60	72	73	83	85	88



Heap sort

```
procedure Heapsort (var A: Vetor; var n: Indice);
var Esq, Dir: Indice;
    x        : Item;
{-- Entra aqui o procedimento Refaz --}
{-- Entra aqui o procedimento Constroi --}
begin
  Constroi(A, n);  { constroi o heap }
  Esq := 1;  Dir := n;
  while Dir > 1 do  { ordena o vetor }
    begin
    x := A[1];  A[1] := A[Dir];  A[Dir] := x;
    Dir := Dir - 1;
    Refaz (Esq, Dir, A);
    end;
end;
```

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

O(n)

```
procedure Refaz (Esq, Dir: Indice; var A: Vetor);
label 999;
var i: Indice;
    j: integer;
    x: Item;
begin
  i := Esq; j := 2 * i;
  x := A[i];
  while j <= Dir do
    begin
    if j < Dir
    then if A[j].Chave < A[j + 1].Chave then j := j+1;
    if x.Chave >= A[j].Chave then goto 999;
    A[i] := A[j];
    i := j; j := 2 * i;
    end;
  999: A[i] := x;
end;
```

O(log n)

Heap sort

- **Vantagens**:
  - 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.
  - 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.

Exercícios

1. Fazer um resumo sobre os métodos de ordenação quadráticos e n log n.

2. Implementação:
   i. Gerar vetores aleatórios de tamanho 10, 100, 1.000, 10.000, 100.000, 500.000
   ii. Ordenar os vetores com os 6 métodos vistos até o momento
   iii. Armazenar o tempo de execução que cada método levou para ordenar
   iv. Após ordenar os vetores, chamar os métodos para ordenar novamente e armazenar o tempo
   v. Ordenar os vetores em ordem descendente e armazenar o tempo
   vi. Plotar três gráficos com os respectivos tempos

3. Qual seria o algoritmo de ordenação mais eficiente para cada caso?
   i) uma lista ordenada em ordem ascendente;
   ii) uma lista ordenada em ordem descendente;
   iii) uma lista com os valores desordenados.

Tentar não significa conseguir, mas quem conseguiu tentou.

Aristóteles