DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Algoritmos Eficientes de Ordenação

Sedgewick: Capítulo 6

IAED

Algoritmos Eficientes de Ordenação

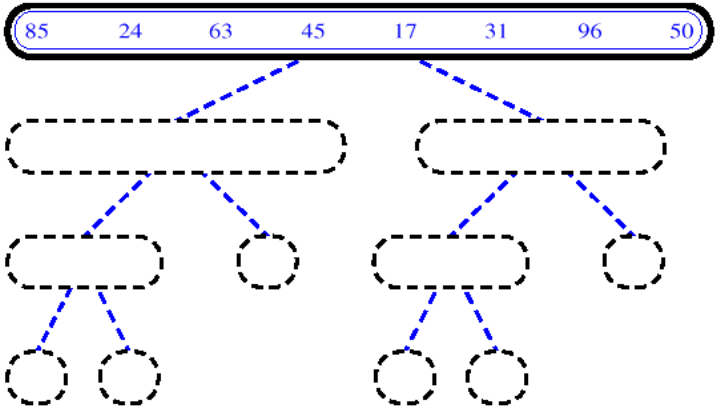
- Quick Sort
- Merge Sort
- Heap Sort

Utilizar informação das chaves:

- Counting Sort
- Radix Sort

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Quick Sort



IAED

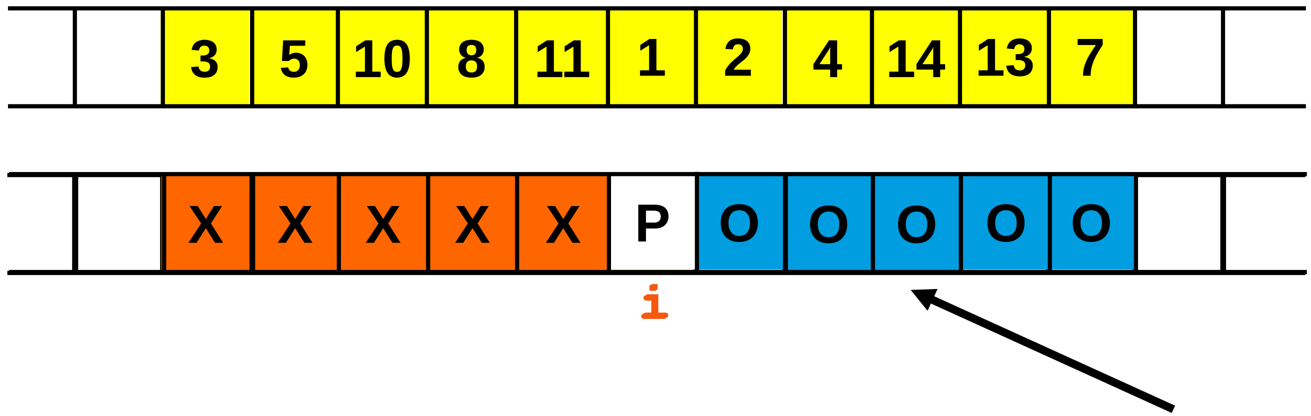
Quick Sort - Introdução

- Inventado nos anos 60 por A. R. Hoare
- Vantagens
  - popular devido à facilidade de implementação e eficiência
  - $O(N \lg N)$, em **média**, para ordenar N objectos
  - ciclo interno muito simples e conciso
- Inconvenientes
  - não é estável; $O(N^2)$ no **pior caso**!
  - frágil: qualquer pequeno erro de concretização pode não ser detectado mas levar a ineficiência
- Biblioteca C fornece uma concretização: `qsort()`

Quick Sort - Introdução

- Aplica método ***dividir para conquistar*** para ordenar *("divide and conquer")*
- Ideia chave: efectuar partição dos dados e ordenar as várias partes independentemente (de forma recursiva)
  - particionar os dados: menores para um lado, maiores para outro
  - usar recursão e aplicar algoritmo a cada uma das partes
  - processo de partição é crítico para evitar partições degeneradas

Quick Sort - ideia

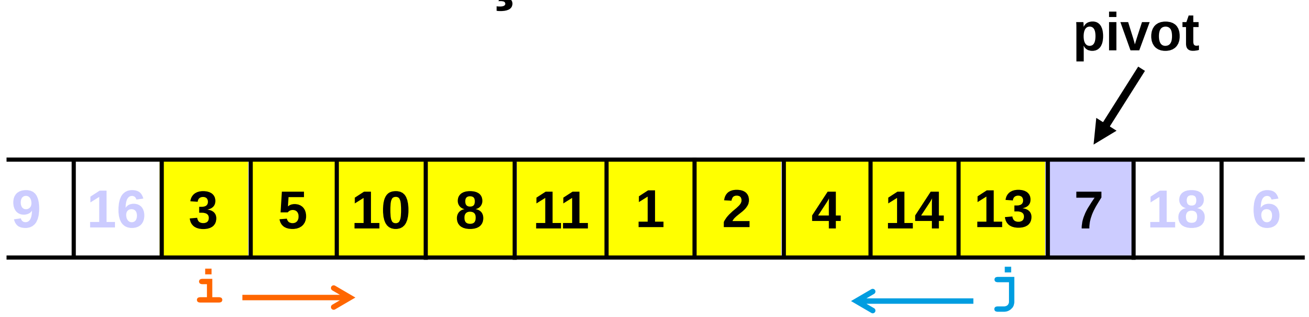


Todos os X’s < P e
Todos os O’s > P
P fica na posição final

```
void quicksort(Item a[], int left, int
    right) {
   int i;
   if (right <= left) return;
   i = partition(a, left, right);
   quicksort(a, left, i-1);
   quicksort(a, i+1, right);
}
```

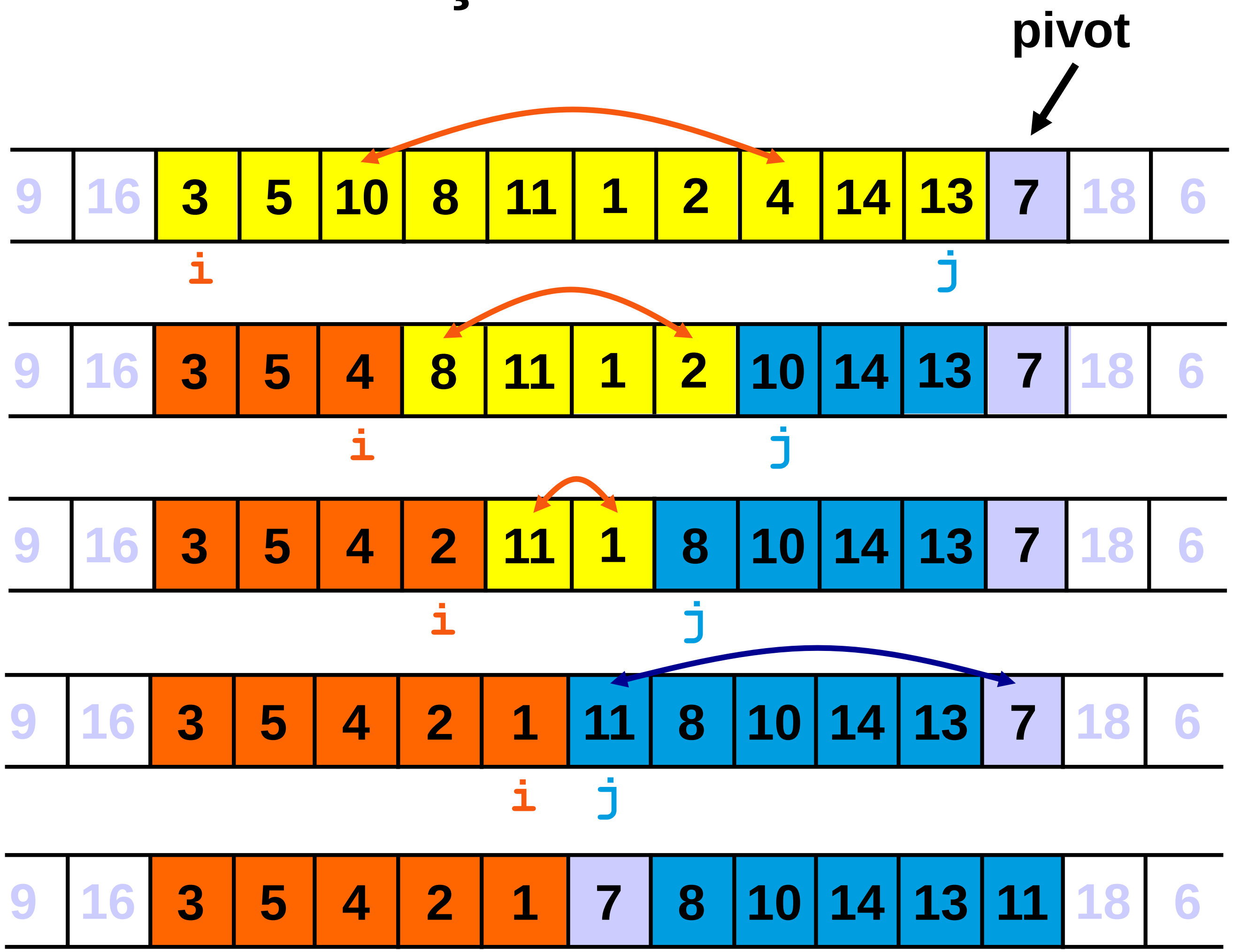
- Volto a aplicar o mesmo algoritmo aos XX e aos OO’s
- Quando não pudermos dividir mais, paramos

Quick Sort – Partição - ideia



- 2 iteradores: o `i` (esquerda) e o `j` (direita, mas antes do pivot)
- Avanço com `i` (para a direita) até encontrar um elemento **maior que o pivot**, ou seja, que deveria estar no lado direito do vector.
- Avanço com `j` (para a esquerda) até encontrar um elemento **menor que o pivot**, ou seja, que deveria estar no lado esquerdo do vector.
- Troco `a[i]` com `a[j]`
- Repito estes passos até que `i` cruze com `j`.
- Nessa altura, troco o pivot com o `a[i]`, colocando-o na divisão entre o **“lado” esquerdo** e o **lado “direito”** do vector.

Quick Sort - Partição



Quick Sort - Partição

- Recebe vector **`a`** e os limites da parte a ordenar, **`l`** e **`r`**
- Rearranja os elementos do vector de forma a que as **quatro condições** seguintes sejam válidas
  - o elemento **`a[i]`**, para algum **`i`**, fica na sua posição final (i.e., o pivot)
  - nenhum dos elementos de **`a[l]`** a **`a[i – 1]`** é maior do que **`a[i]`**
  - nenhum dos elementos de **`a[i + 1]`** a **`a[r]`** é menor do que **`a[i]`**
  - processo coloca pelo menos um elemento na sua posição final

- Após partição, a tabela fica sub-dividida em duas partes que podem ser ordenadas independentemente aplicando o mesmo processo

Quick Sort - Partição

`v = a[r]` *é o pivot!!*

```
int partition(Item a[], int left, int right) {
  int i = left-1;
  int j = right;
  Item v = a[right];
  while (i < j) {
    while (less(a[++i], v));
    while (less(v, a[--j]))
      if (j == left)
        break;
    if (i < j)
      exch(a[i], a[j]);
  }
  exch(a[i], a[right]);
  return i;
}
```

Enquanto o iterador da esquerda `i` for menor que o iterador da direita `j`....

Enquanto `a[i]<v` avança com o `i` para a direita.

Enquanto `v<a[j]` avança com o `j` para a esquerda.

Protege do caso em que o elemento onde se faz a partição está na 1ª posição

Faz a troca...

Coloca o pivot na posição `i`, de forma a que pelo menos este elemento fique na pos final

Retorna o ponto onde partiu o vector, para que esta informação seja usada na função `quicksort`

Quick Sort - Recursão

- Ordenação realizada através de partição e aplicação recursiva do algoritmo aos dois subconjuntos de dados daí resultantes

```
void quicksort(Item a[], int left, int right)
{
  int i;

  if (right <= left)
    return;

  i = partition(a, left, right);
  quicksort(a, left, i-1);
  quicksort(a, i+1, right);
}
```

Quick Sort - Recursão

9	16	3	5	4	2	1	7	8	10	14	13	11	18	6
9	16	1	5	4	2	3	7	8	10	14	13	11	18	6
9	16	1	2	3	5	4	7	8	10	14	13	11	18	6
9	16	1	2	3	4	5	7	8	10	14	13	11	18	6
9	16	1	2	3	4	5	7	8	10	11	13	14	18	6
9	16	1	2	3	4	5	7	8	10	11	13	14	18	6
9	16	1	2	3	4	5	7	8	10	11	13	14	18	6
9	16	1	2	3	4	5	7	8	10	11	13	14	18	6

Quick Sort | *função partição* | Exercício

Considere a função int partition (Item a[], int left, int right) usada no quicksort.

```
int partition(Item a[], int left,
    int right)
{
  int i = left-1
  int j = right;
  Item v = a[right];
  while (i < j) {
    while (less(a[++i], v));
    while (less(v, a[--j]))
      if (j == l)
        break;
    if (i < j)
      exch(a[i], a[j]);
  }
  exch(a[i], a[right]);
  return i;
}
```

Suponha que a função `partition` é invocada com os seguintes argumentos:

`a = { 10, 2, -2, 13, 14, -1, 7, 5, 0, 6 },`
`l = 0, r = 9`

Indique qual o conteúdo do vector a após a execução da função `partition`.

IAED TÉCNICO LISBOA

Quick Sort | *função partição* | Exercício

- Pivot = `v = a[right] = 6`

```
    10, 2, -2, 13, 14, -1, 7, 5, 0, 6
i                                   j
```

- Troco o 10 com o 0

```
     0, 2, -2, 13, 14, -1, 7, 5, 10, 6
     i                           j
```

- Troco o 13 com o 5

```
    0, 2, -2, 5, 14, -1, 7, 13, 10, 6
              i              j
```

- Troco o 14 com o -1

Quick Sort | *função partição* | Exercício

`0, 2, -2, 5, -1, 14, 7, 13, 10, 6`

`i`

`j`

- Como `i>=j`, nada acontece e saímos do ciclo `while`. Agora trocamos o elemento na posição `i` (o 14) com o pivot (**`exch(a[i],a[right])`**), ficando

`0, 2, -2, 5, -1, 6, 7, 13, 10, 14`

`i`

- Retorno o `i` (neste caso o índice `l+5`)

Quick Sort - Complexidade

- Pior caso: pivot é sempre o maior/menor elemento
  - Cerca de $N^2/2$ comparações
  - Partições degeneram e a função chama-se a si própria N vezes;
  - O número de comparações é

$$N+(N-1)+(N-2)+\ldots+2+1=\frac{(N+1)N}{2}$$

  - Na nossa implementação se o vector estiver ordenado

- Não apenas o tempo necessário para a execução do algoritmo cresce quadraticamente como o espaço necessário para o processo recursivo é de cerca de N o que é inaceitável para vectores grandes. É possível modificar para obter espaço *O(lg N)*

Quick Sort - Complexidade

- Melhor caso: quando cada partição divide o vector de entrada em duas metades iguais
  - Número de comparações usadas por **`quicksort`** satisfaz a recursão de dividir para conquistar: $C_N = 2C_{N/2} + N$
  - Solução: $\boldsymbol{C_N \approx N \lg N}$

- Propriedade: QuickSort efectua cerca de $\boldsymbol{C_N \approx 2N \lg N}$ comparações **em média**

Quick Sort - Melhorias

- Algoritmo pode ainda ser melhorado com alterações triviais
- Ordenação de sub-vectores de pequenas dimensões pode ser efectuada de forma mais eficiente
  – Natureza recursiva de QuickSort garante que uma fracção grande dos sub-vectores terão tamanho pequeno
- Como escolher correctamente o elemento de partição?
  – Aleatoriamente
  – Média de vários elementos
- Como melhorar o desempenho se os dados tiverem um grande número de chaves repetidas?

Quick Sort - Melhorias

- QuickSort é garantido instanciar-se a si próprio múltiplas vezes para vectores pequenos!
- Conveniente utilizar o melhor método possível nesta situação: insertion sort

```
void quicksort(Item a[], int l, int r)
{
  int i;

  if (right <= left)
    return;

  i = partition(a, left, right);
  quicksort(a, left, i-1);
  quicksort(a, i+1, right);
}
```

```
if(right-left <= M) {
  insertion(a, left, right)
  return;
}
```

III.c) Considere a função `int partition (Item a[], int l, int r)` usada no algoritmo `quicksort`.

```
int partition(Item a[], int l, int r) {
   int i = l-1, j = r;
   Item v = a[r];
   while (i < j) {
      while (less(a[++i], v));
      while (less(v, a[--j]))
         if (j == l)
            break;
      if (i < j)
         exch(a[i], a[j]);
   }
   exch(a[i], a[r]);
   return i;
}
```

Exercício

Esta função recebe o vector `a` e as posições `l` e `r` que definem, respectivamente, os índices limite esquerdo e direito do vector a considerar na função. Suponha que a função `partition` é invocada com os seguintes argumentos:

```
a = {8, 1, -2, 10, 12, -7, 7, 11, 0, 5}, l = 0, r = 9
```

Indique qual o conteúdo do vector `a` após a execução da função `partition` e indique o valor de `i` retornado na ultima linha da função.

`8, 1, -2, 10, 12, -7, 7, 11, 0, 5`

(o *pivot* é o 5!)

o *iterador* `i` começa por parar no 8 e o `j` no 0 : troco o 8 com o 0

`0, 1, -2, 10, 12, -7, 7, 11, 8, 5`

o *iterador* `i` pára no 10 e o `j` no -7: troco o 10 com o -7

`0, 1, -2, -7, 12, 10, 7, 11, 8, 5`

o *iterador* `i` pára no 12 e o `j` no -7: como o `i>j` já não troco

```
            j   i
0, 1, -2, -7, 12, 10, 7, 11, 8, 5
```

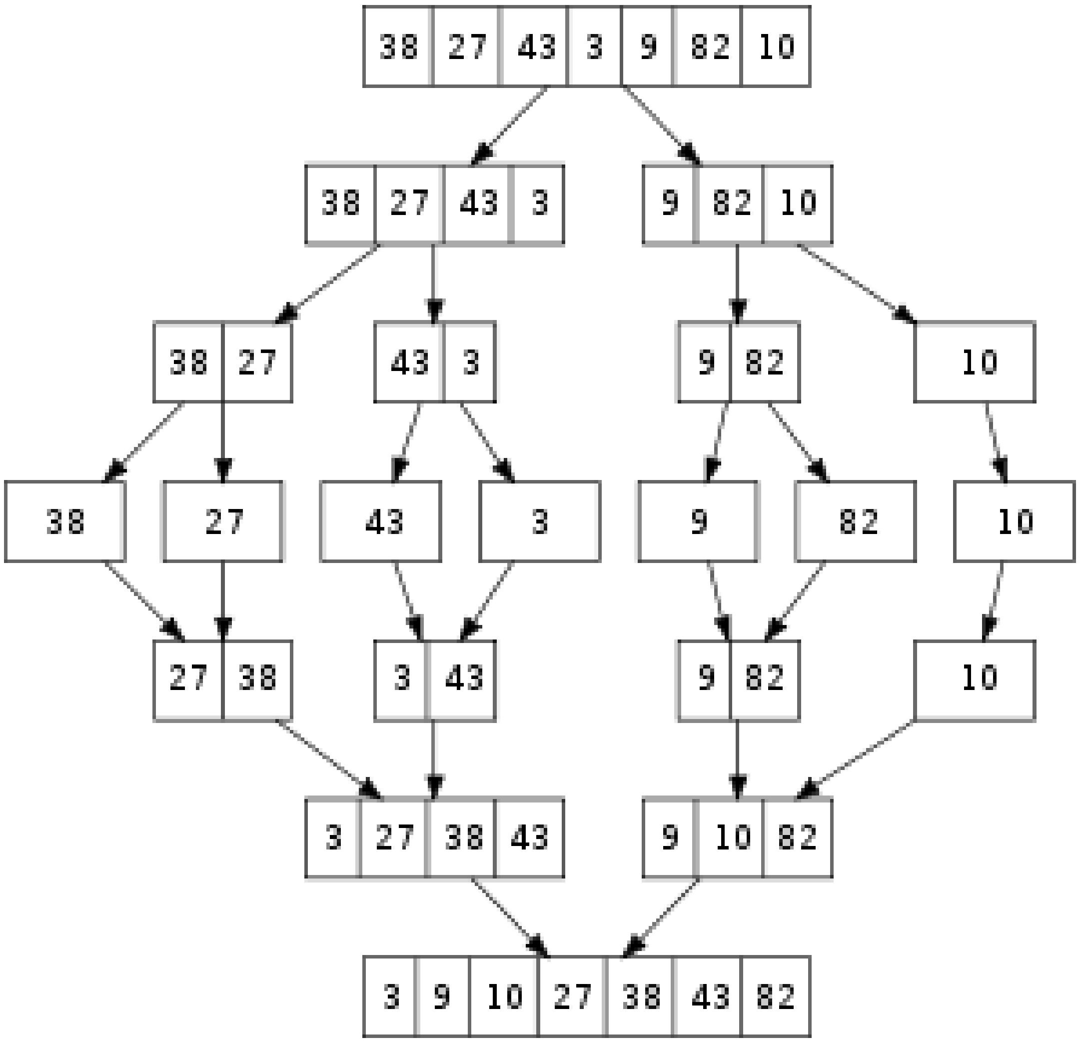
Resta-me portanto trocar o `a[i]` com o *pivot*, ficando:

resultado : `0, 1, -2, -7, 5, 10, 7, 11, 8, 12`

...e a função partição deverá retornar o inteiro $\boxed{4}$

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Merge Sort

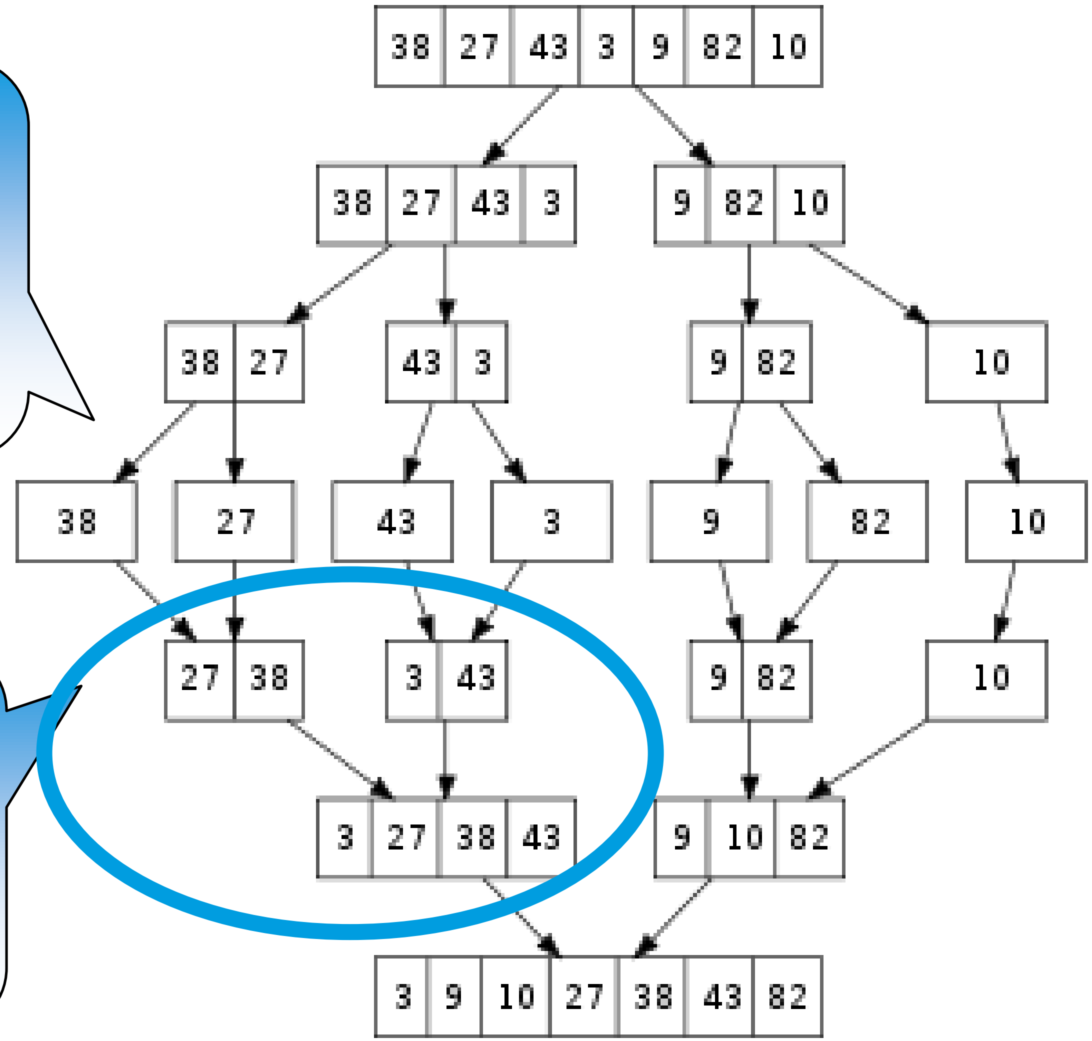


IAED

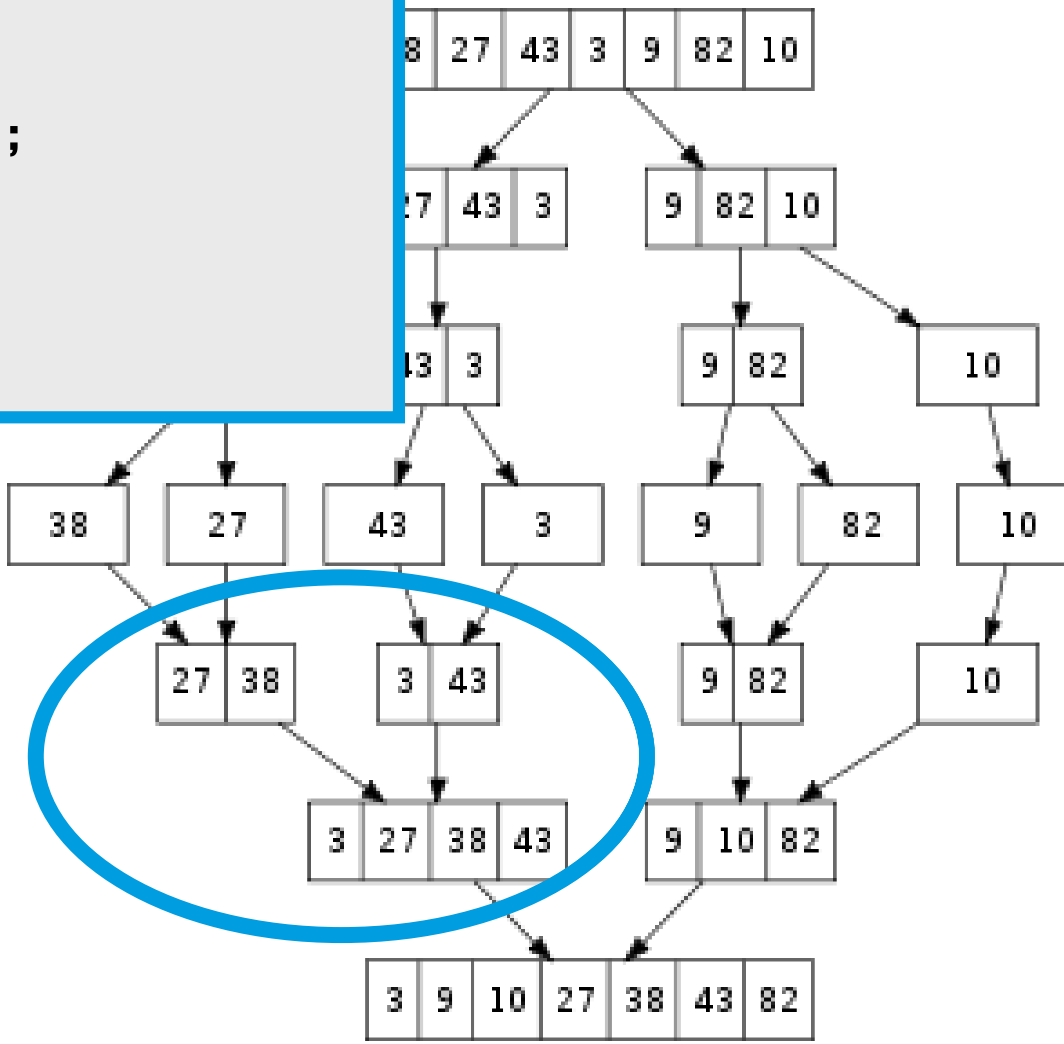
Merge Sort - ideia

Partir sucessivamente ao meio o vector de elementos a ordenar, até obtermos vectores com apenas um elemento

Aplicar sucessivamente o procedimento de Merge, para gerar um vector ordenado a partir de dois vectores ordenados

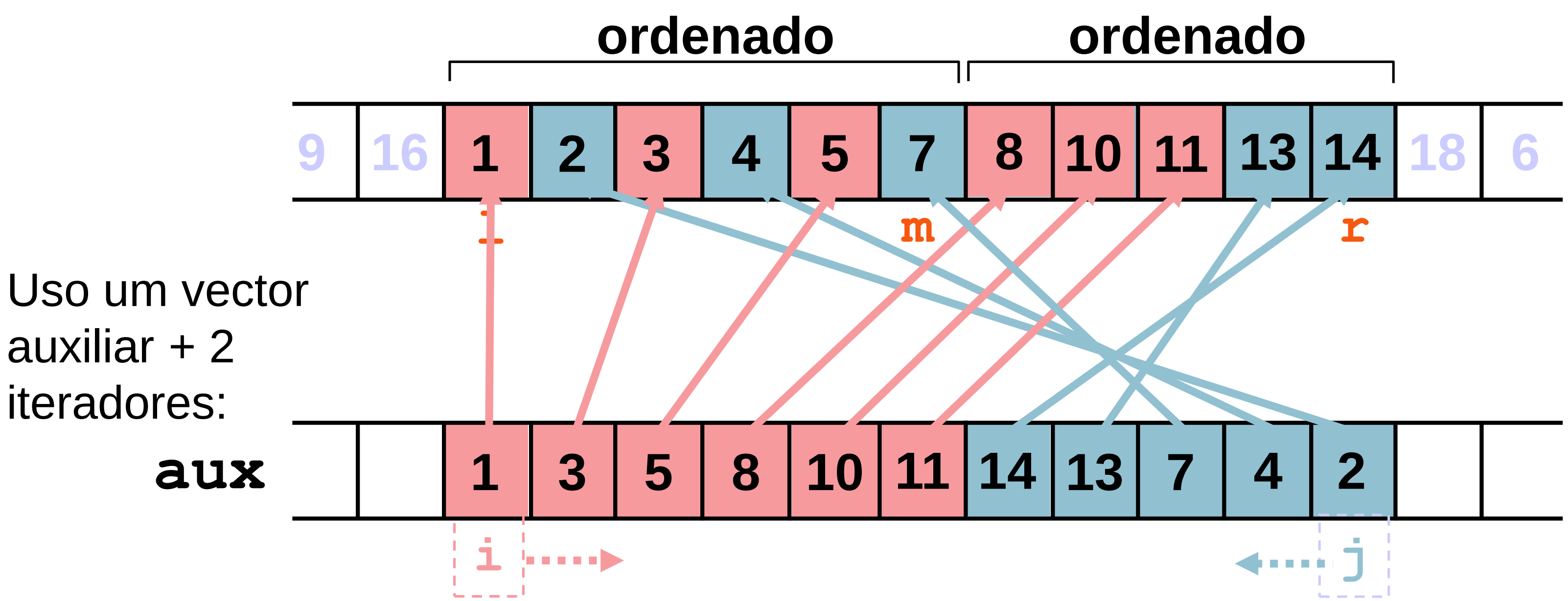


```
void mergesort(Item a[], int left, int
   right)
{
  int m = (right+left)/2;
  if (right <= left) return;
  mergesort(a, left, m);
  mergesort(a, m+1, right);
  merge(a, left, m, right);
}
```



Merge Sort - Merge

- Gerar um vector ordenado a partir de dois vectores ordenados é simples!



Merge Sort - Merge

Devolve um vector ordenado, em `a[left..right]`, dados dois vectores ordenados em `a[left..m]` e `a[m+1..right]`

```
Item aux[maxN];

void merge(Item a[], int left, int m, int right)
{
  int i, j, k;
  for (i = m+1; i > left; i--)
    aux[i-1] = a[i-1];
  for (j = m; j < right; j++)
    aux[right+m-j] = a[j+1];
  for (k = left; k <= right; k++)
    if (less(aux[j], aux[i]) || i == m+1)
      a[k] = aux[j--];
    else
      a[k] = aux[i++];
}
```

Merge Sort - Merge

```
Item aux[maxN];

void merge(Item a[], int left, int m, int right)
{
  int i, j, k;
  for (i = m+1; i > left; i--)
    aux[i-1] = a[i-1];
  for (j = m; j < right; j++)
    aux[right+m-j] = a[j+1];
  for (k = left; k <= right; k++)
    if (less(aux[j], aux[i]) || i == m+1)
      a[k] = aux[j--];
    else
      a[k] = aux[i++];
}
```

Constroi o vector auxiliar

Merge Sort - Merge

A função Merge quebra a estabilidade do algoritmo?

```
Item aux[maxN];

void merge(Item a[], int left, int m, int right)
{
  int i, j, k;
  for (i = m+1; i > left; i--)
    aux[i-1] = a[i-1];
  for (j = m; j < right; j++)
    aux[right+m-j] = a[j+1];
  for (k = left; k <= right; k++)
    if (less(aux[j], aux[i]) || i =
      a[k] = aux[j--];
    else
      a[k] = aux[i++];
}
```

Vai escolhendo os elementos das pontas de forma a ordenar o vector a [ ]

Merge Sort *(top-down* version)

```
void mergesort(Item a[], int left, int right) {
  int m = (right+left)/2;

  if (right <= left)
    return;

  mergesort(a, left, m);
  mergesort(a, m+1, right);
  merge(a, left, m, right);
}
```

Tal como o quicksort, temos uma função recursiva.

Merge Sort

```
mergesort(a,0,10)
    mergesort(a,0,5)
    ...
    mergesort(a,6,10)
    ...
```

```
void mergesort(Item a[], int
   left, int right)
{
  int m = (right+left)/2;

  if (right <= left)
    return;

  mergesort(a, left, m);
  mergesort(a, m+1, right);
  merge(a, left, m, right);
}
```

Merge Sort

```
mergesort(a,0,10)
    mergesort(a,0,5)
        mergesort(a,0,2)
              ...
        mergesort(a,3,5)
              ...
    mergesort(a,6,10)
        mergesort(a,6,8)
              ...
        mergesort(a,9,10)
              ...
```

```
void mergesort(Item a[], int
   left, int right)
{
  int m = (right+left)/2;

  if (right <= left)
    return;

  mergesort(a, left, m);
  mergesort(a, m+1, right);
  merge(a, left, m, right);
}
```

Merge Sort

```
mergesort(a,0,10)
      mergesort(a,0,5)
            mergesort(a,0,2)
                  mergesort(a,0,1)

    mergesort(a,0,0)

    mergesort(a,1,1)
                  mergesort(a,2,2)
            mergesort(a,3,5)
                  mergesort(a,3,4)

    mergesort(a,3,3)

    mergesort(a,4,4)
                  mergesort(a,5,5)
      mergesort(a,6,10)
            mergesort(a,6,8)
                  mergesort(a,6,7)

    mergesort(a,6,6)

    mergesort(a,7,7)
                  mergesort(a,8,8)
            mergesort(a,9,10)
                  mergesort(a,9,9)
```

```
void mergesort(Item a[], int
   left, int right)
{
  int m = (right+left)/2;

  if (right <= left)
    return;

  mergesort(a, left, m);
  mergesort(a, m+1, right);
  merge(a, left, m, right);
}
```

Merge Sort

	0	1	2	3	4	5	6	7	8	9	10
	3	**5**	**10**	**8**	**11**	**1**	**2**	**4**	**14**	**13**	**7**
`mergesort(a,0,1)`	**3**	**5**	10	8	11	1	2	4	14	13	7
`mergesort(a,0,2)`	**3**	**5**	**10**	8	11	1	2	4	14	13	7
`mergesort(a,3,4)`	3	5	10	**8**	**11**	1	2	4	14	13	7
`mergesort(a,3,5)`	3	5	10	**1**	**8**	**11**	2	4	14	13	7
`mergesort(a,0,5)`	**1**	**3**	**5**	**8**	**10**	**11**	2	4	14	13	7
`mergesort(a,6,7)`	1	3	5	8	10	11	**2**	**4**	14	13	7
`mergesort(a,6,8)`	1	3	5	8	10	11	**2**	**4**	**14**	13	7
`mergesort(a,9,10)`	1	3	5	8	10	11	2	4	14	**7**	**13**
`mergesort(a,6,10)`	1	3	5	8	10	11	**2**	**4**	**7**	**13**	**14**
`mergesort(a,0,10)`	**1**	**2**	**3**	**4**	**5**	**7**	**8**	**10**	**11**	**13**	**14**

Merge Sort - Complexidade

- Tempo de execução:

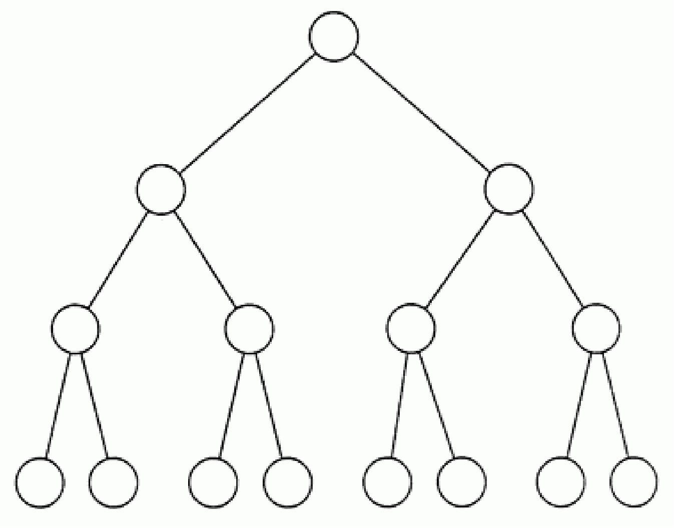
$$T_N = T_{\lfloor N/2 \rfloor} + T_{\lceil N/2 \rceil} + O(N) = O(N \lg N)$$

- Fácil de verificar quando N é potência de 2, e no caso geral recorrendo a indução

- Complexidade do pior caso é $O(N \lg N)$

DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Heap Sort

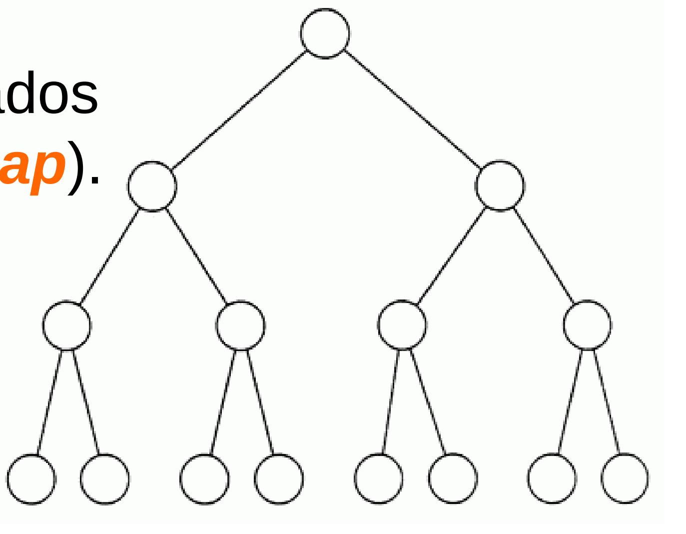


IAED

HeapSort (ideia base)

- Podemos pensar no ***heapsort*** como um selection sort mais eficiente.

- o ***heapsort*** mantém os dados organizados numa estrutura de dados (chamada ***heap***).



- A raiz dessa estrutura, contém sempre o maior elemento.

- Os elementos podem ser sucessivamente removidos da raiz da ***heap***, na ordem desejada, tendo o cuidado de manter as propriedades dessa estrutura

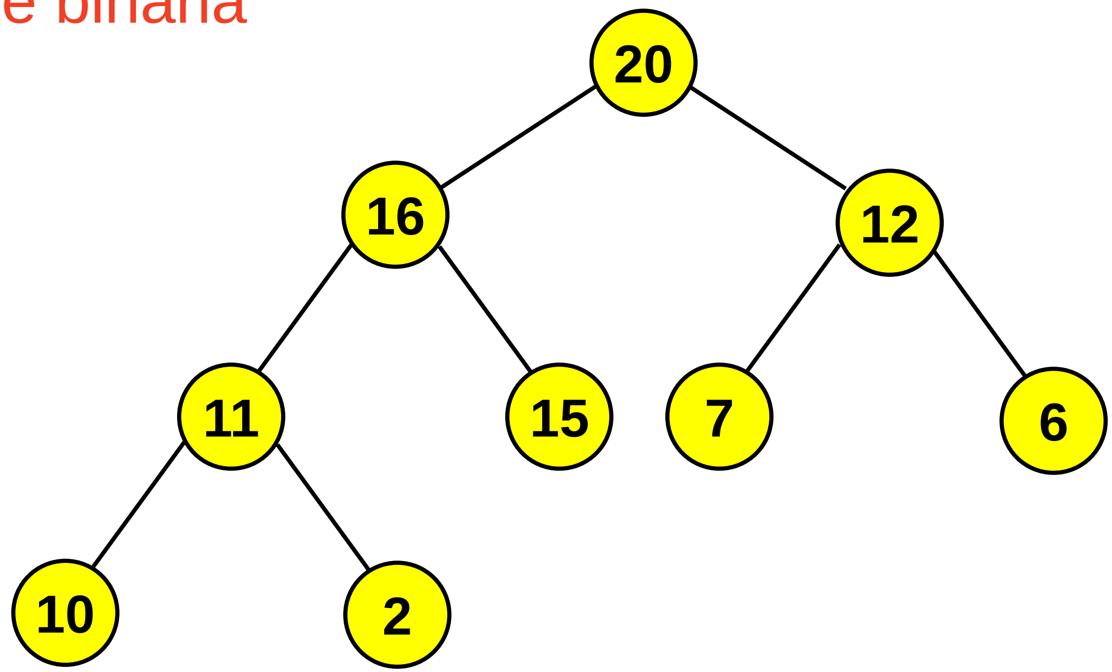
Das árvores aos amontoados (*heaps*)

- Um vector de elementos que pode ser visto como uma árvore binária

20	**16**	**12**	**11**	**15**	**7**	**6**	**10**	**2**

Das árvores aos amontoados (*heaps*)

- Um vector de elementos pode ser visto como uma árvore binária

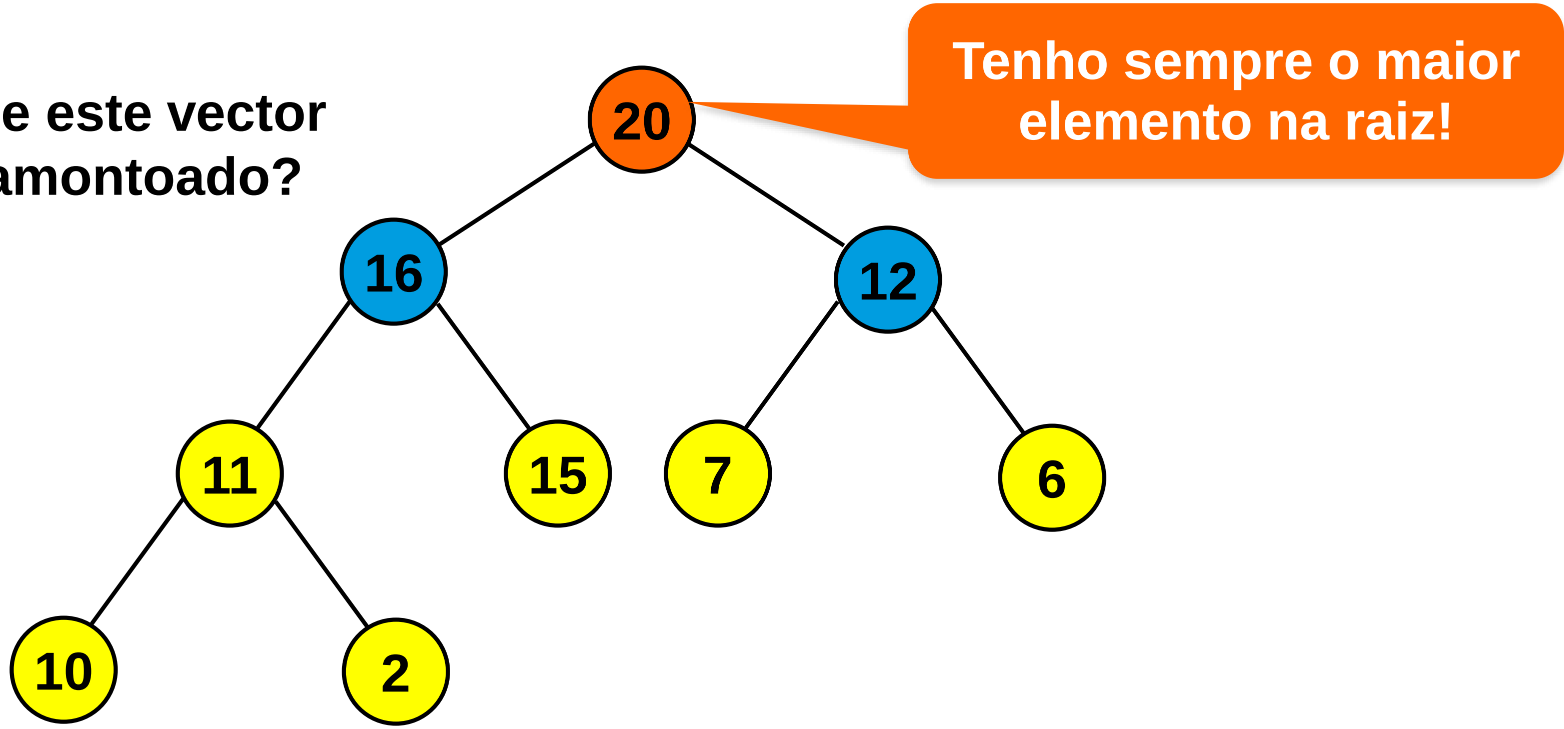


20	16	12	11	15	7	6	10	2

Amontoado (heap): definição

1. Nenhum nó tem uma chave superior à raiz
2. Heap-condition: ***a chave de cada nó é sempre maior ou igual que as chaves de ambos os filhos***

Será que este vector é um amontoado?



Amontoado (heap): definição | Exercício

Quais dos seguintes vectores corresponde a um amontoado (heap)?

- a. <50, 25, 30, 27, 24, 21, 28>
- b. <50, 30, 25, 27, 24, 28, 21>
- c. <60, 50, 9, 40, 41, 10, 8>
- d. <40, 15, 18, 13, 11, 14, 16>
- e. <60, 30, 80, 10, 35, 70, 40>

1. **Nenhum nó tem uma chave superior à raiz**
2. **Heap-condition:** ***a chave de cada nó é sempre maior ou igual que as chaves de ambos os filhos***

Amontoado (heap): definição | Exercício

Quais dos seguintes vectores corresponde a um amontoado (heap)?

- a. <50, 25, 30, 27, 24, 21, 28>
- b. <50, 30, 25, 27, 24, 28, 21>
- c. <60, 50, 9, 40, 41, 10, 8>
- d. <40, 15, 18, 13, 11, 14, 16>
- e. <60, 30, 80, 10, 35, 70, 40>

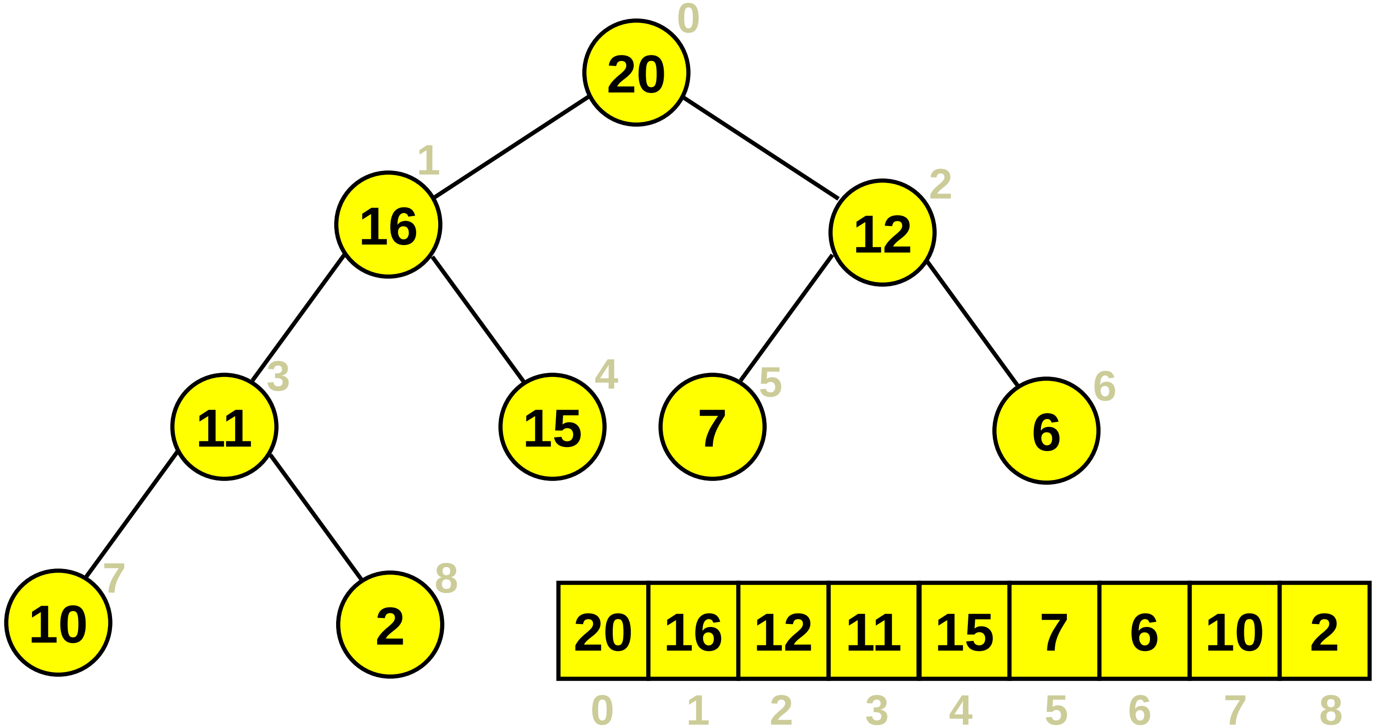
1. **Nenhum nó tem uma chave superior à raiz**
2. **Heap-condition: *a chave de cada nó é sempre maior ou igual que as chaves de ambos os filhos***

Amontoado: Índices

Filho esquerdo

Filho direito

- Pai do nó i é o nó $\lfloor (i+1)/2 \rfloor - 1$
- Filhos do nó i são os nós $2i + 1$ e $2(i + 1)$



Amontoado: Índices

```
int parent(int k) {
  return ((k+1)/2)-1;
}

int left(int k) {
  return 2*k+1;
}

int right(int k) {
  return 2*(k+1);
}
```

Operações em Amontoados: `fixDown`

Fixdown (i):
•chamada quando se pretende diminuir a chave de um nó
•confirma se ambos os filhos são menores do que “ele”;
•se não for o caso, troca de posição com o filho maior e volta a chamar-se para a posição onde o filho maior estava.

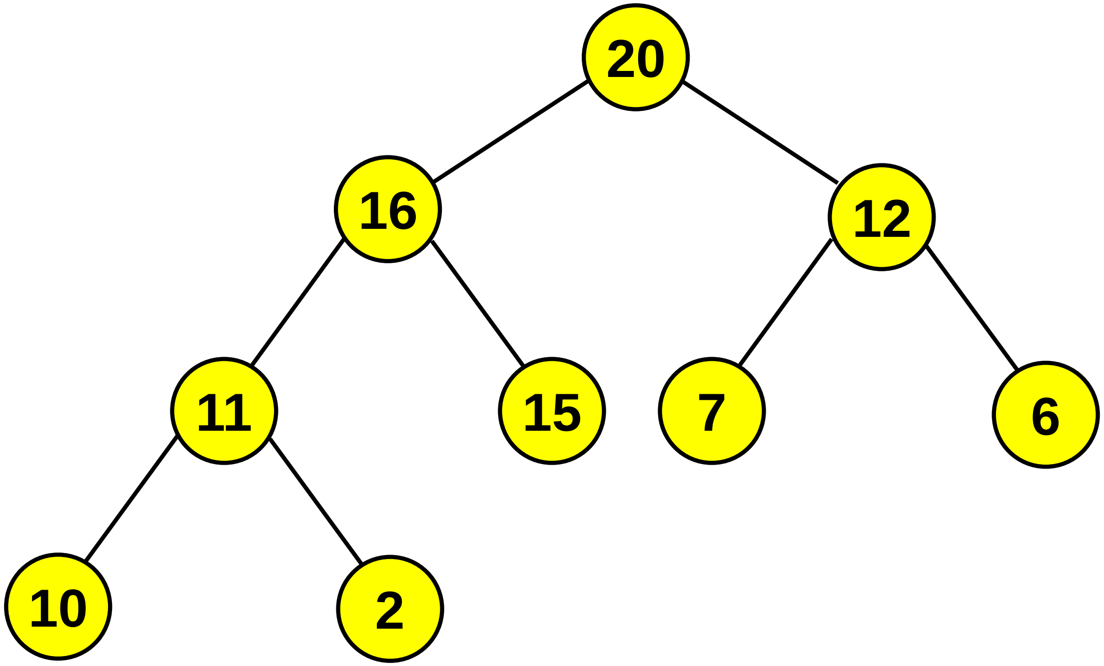
Ideia do código em Linguagem Coloquial:

```
Fixdown (indice i)
      verifica se i tem um filho maior que ele
      SE i tem um filho maior:
            troca i com o filho_maior
            Fixdown (pos_do_filho_maior)
```

Função recursiva

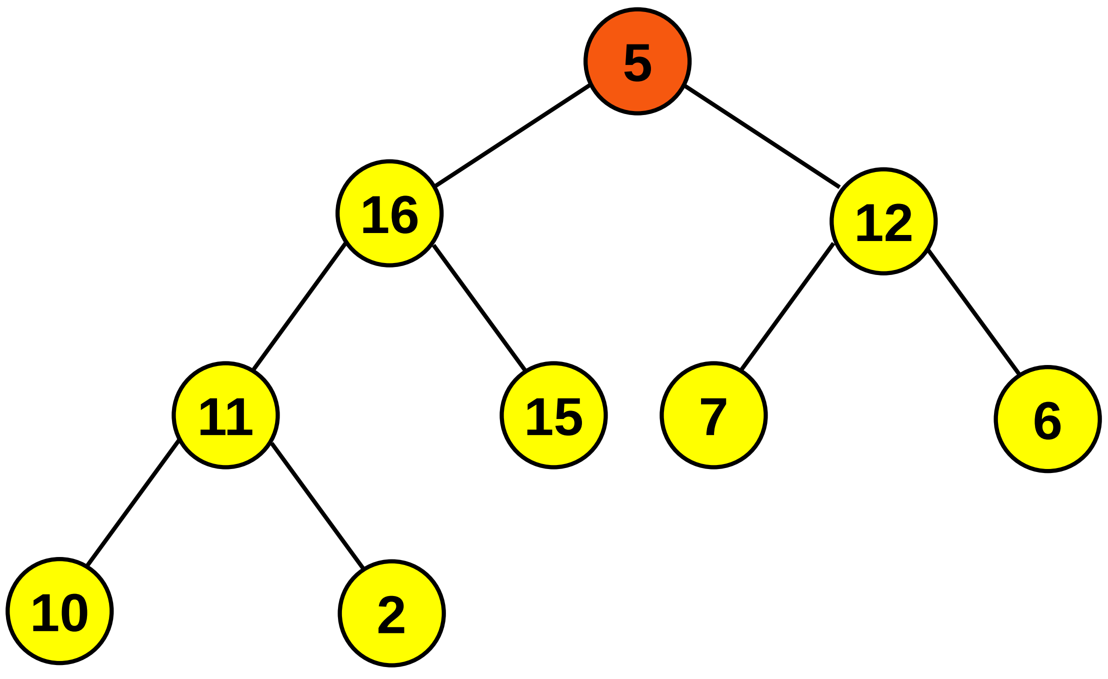
Operações em Amontoados: `fixDown`

- Se tivermos um amontoado...



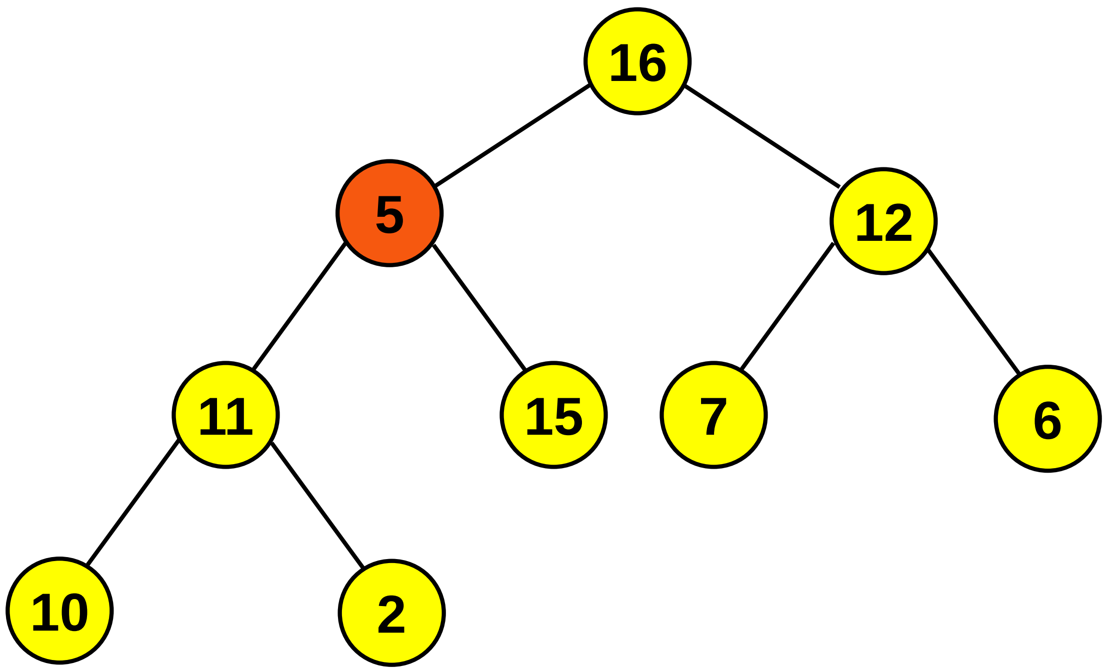
Operações em Amontoados: `fixDown`

- Se tivermos um amontoado... e alterarmos a raiz.
- Podemos voltar a arrumar o vector num “amontoado”, aplicando o fixDown na raiz



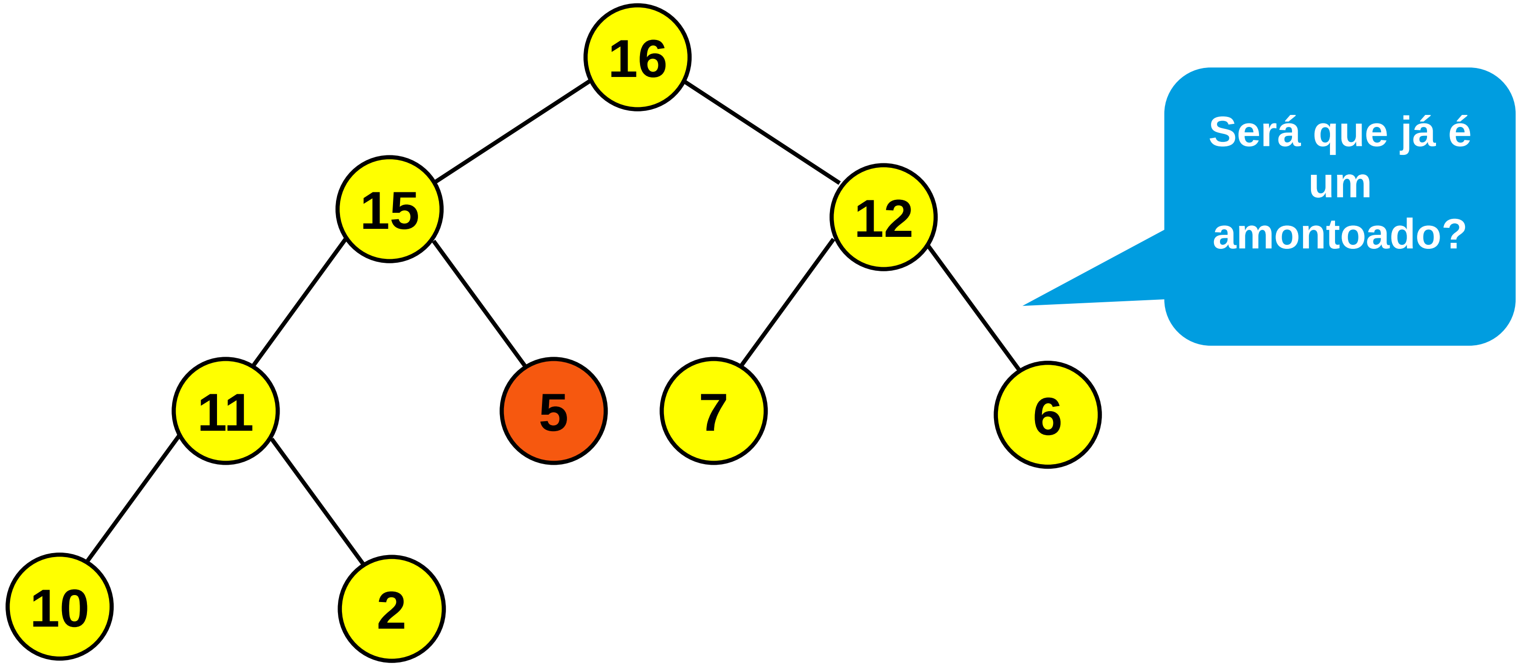
Operações em Amontoados: `fixDown`

- Se tivermos um amontoado... e alterarmos a raiz.
- Podemos voltar a arrumar o vector num "amontoado", aplicando o fixDown na raiz



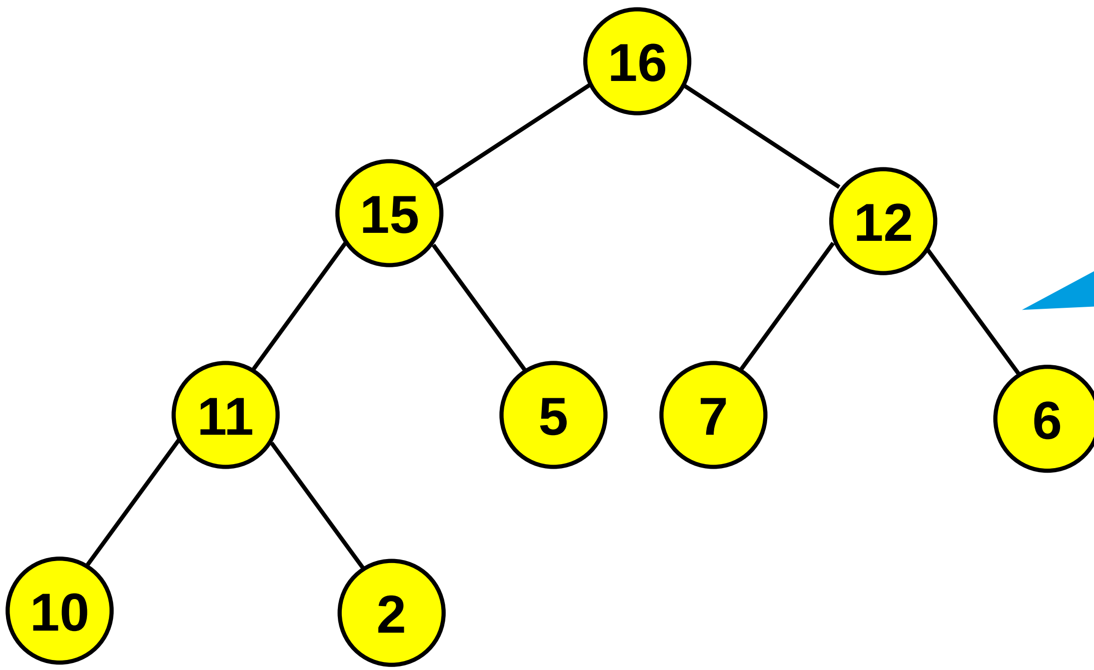
Operações em Amontoados: `fixDown`

- Se tivermos um amontoado... e alterarmos a raiz.
- Podemos voltar a arrumar o vector num “amontoado”, aplicando o fixDown na raiz



Operações em Amontoados: `fixDown`

- Se tivermos um amontoado... e alterarmos a raiz.
- Podemos voltar a arrumar o vector num “amontoado”, aplicando o fixDown na raiz



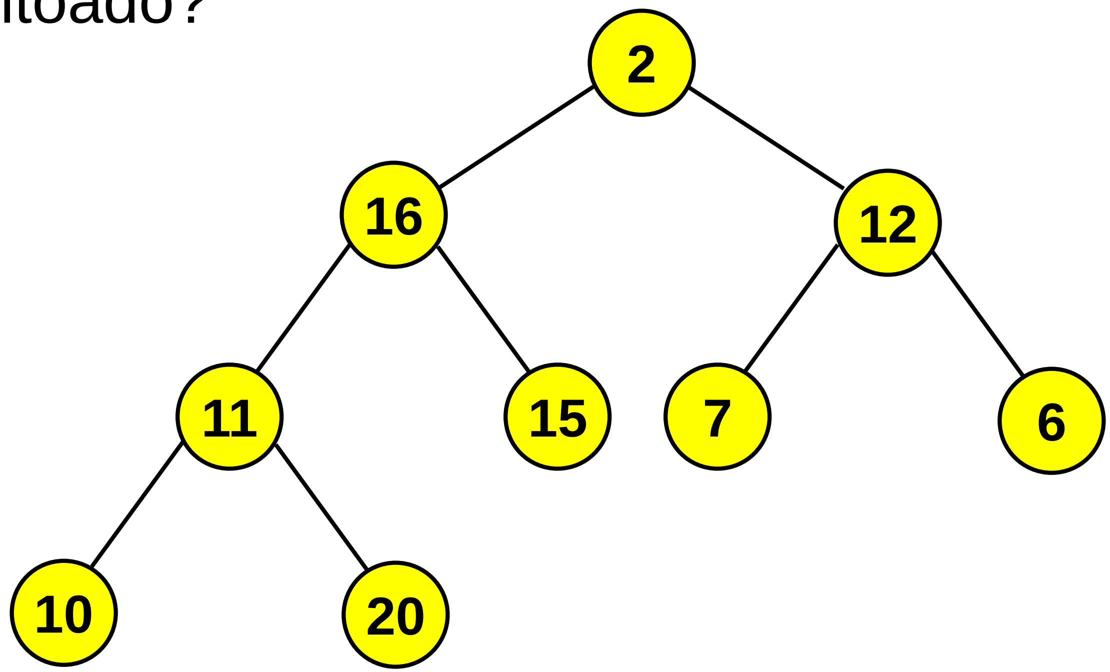
Será que já é um amontoado? SIM!!!!

Operações em Amontoados: `fixDown` (em C)

```
void fixDown(Item a[], int l, int r, int k)
{
    int ileft, iright, largest=k;

    ileft=l+left(k-l);
    iright=l+right(k-l);

    if (ileft<=r && less(a[largest],a[ileft]))
        largest=ileft;
    if (iright<=r && less(a[largest],a[iright]))
        largest=iright;

    if (largest!=k) {
        exch(a[k],a[largest]);
        fixDown(a, l, r, largest);
    }
}
```

Operações em Amontoados: `buildheap`

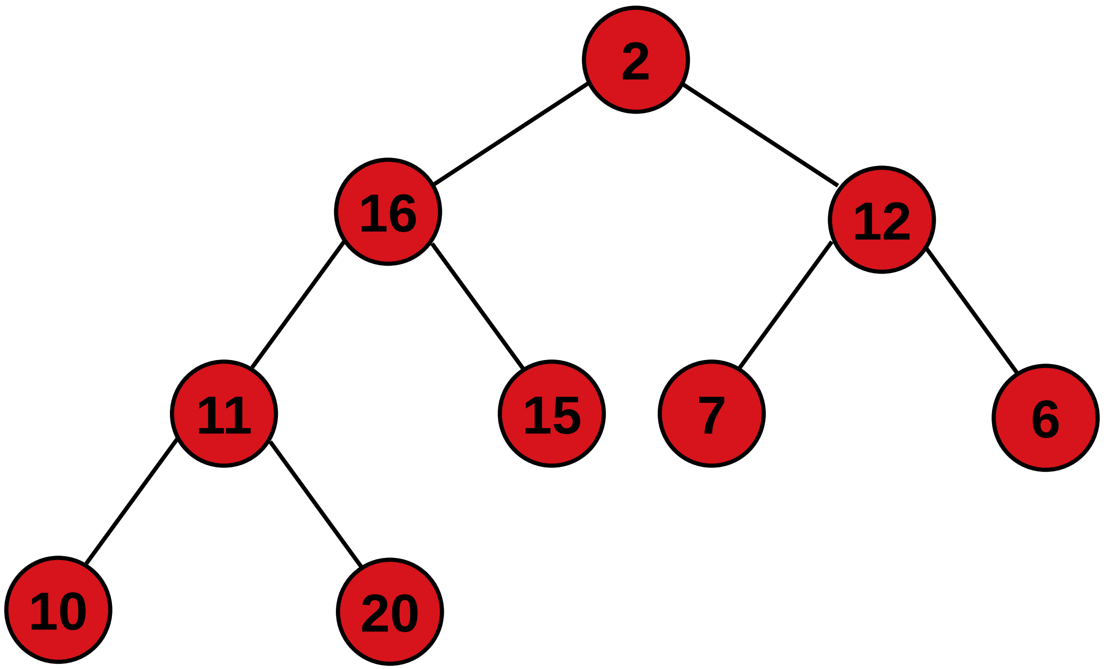
- Mas como transformar um vector arbitrário num amontoado?



2	16	12	11	15	7	6	10	20

Operações em Amontoados: `buildheap`

- Começo por transformar o vector numa árvore.

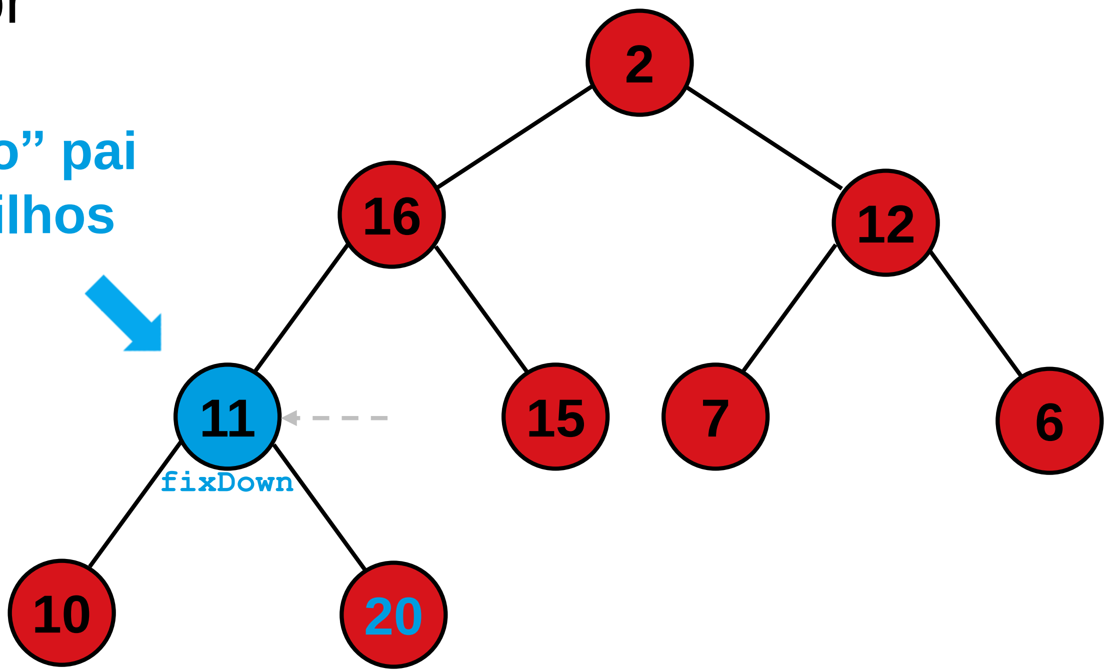


2	16	12	11	15	7	6	10	20

Operações em Amontoados: `buildheap`

- Chamo `fixDown` do último até ao primeiro elemento do vector

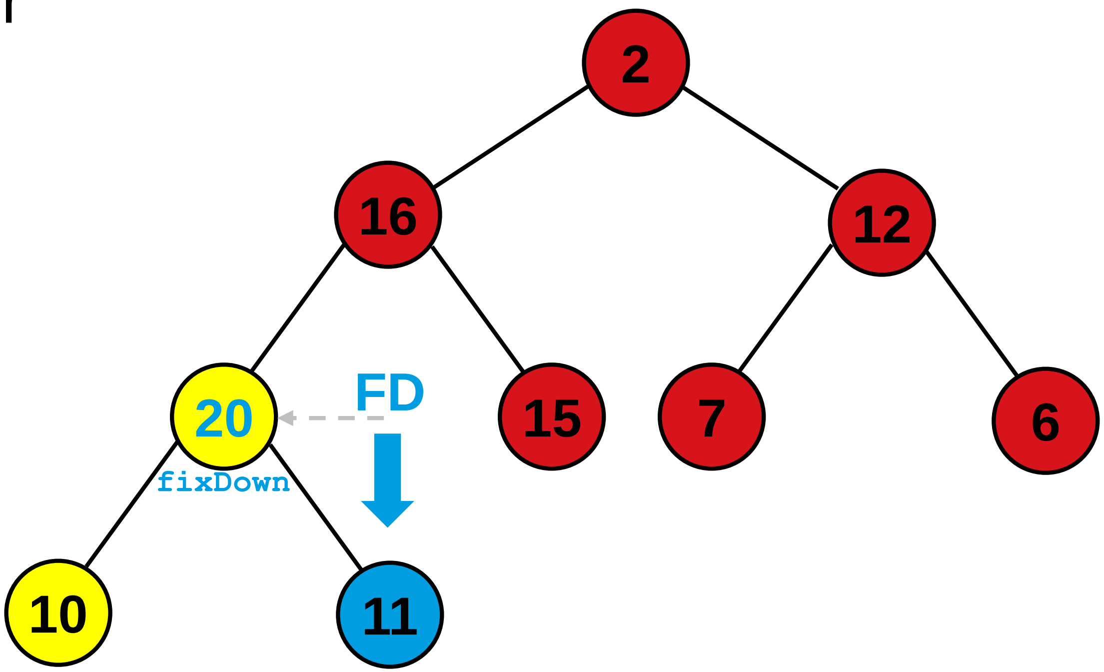
“Último” pai com filhos



2	16	12	11	15	7	6	10	20

Operações em Amontoados: `buildheap`

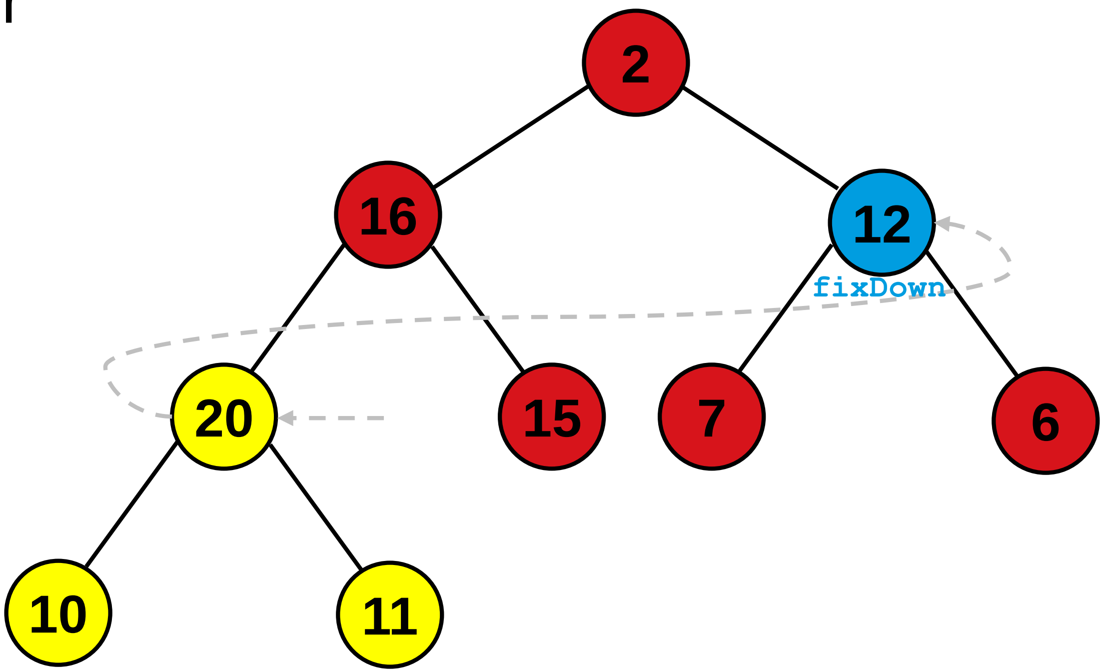
- Chamo `fixDown` do último até ao primeiro elemento do vector



2	16	12	20	15	7	6	10	11

Operações em Amontoados: `buildheap`

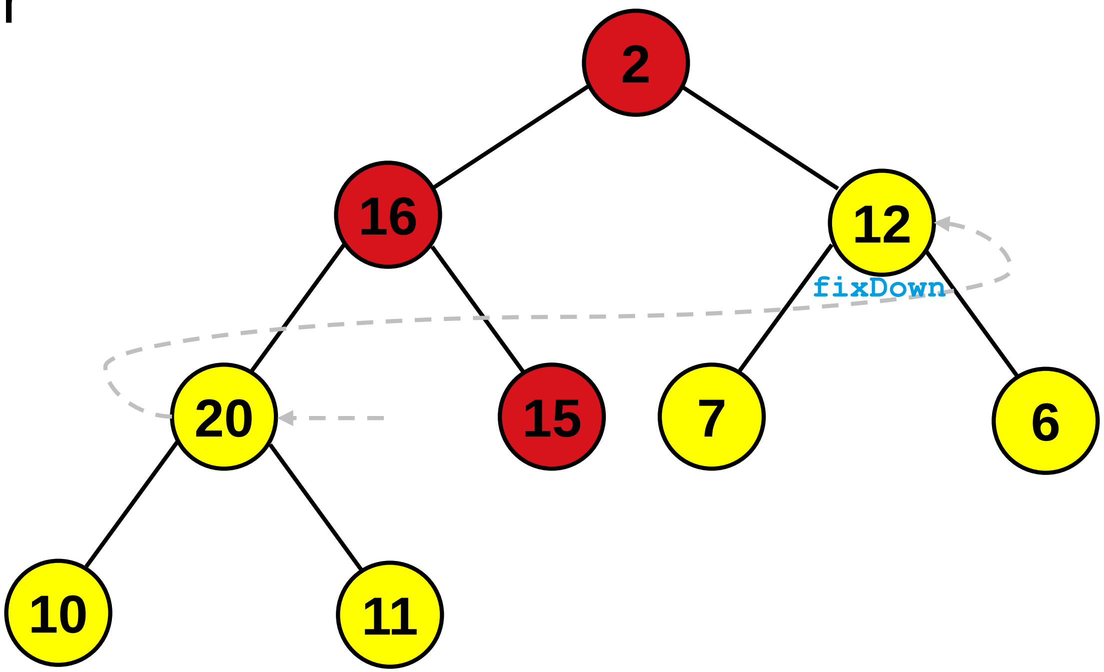
- Chamo `fixDown` do último até ao primeiro elemento do vector



2	16	12	20	15	7	6	10	11

Operações em Amontoados: `buildheap`

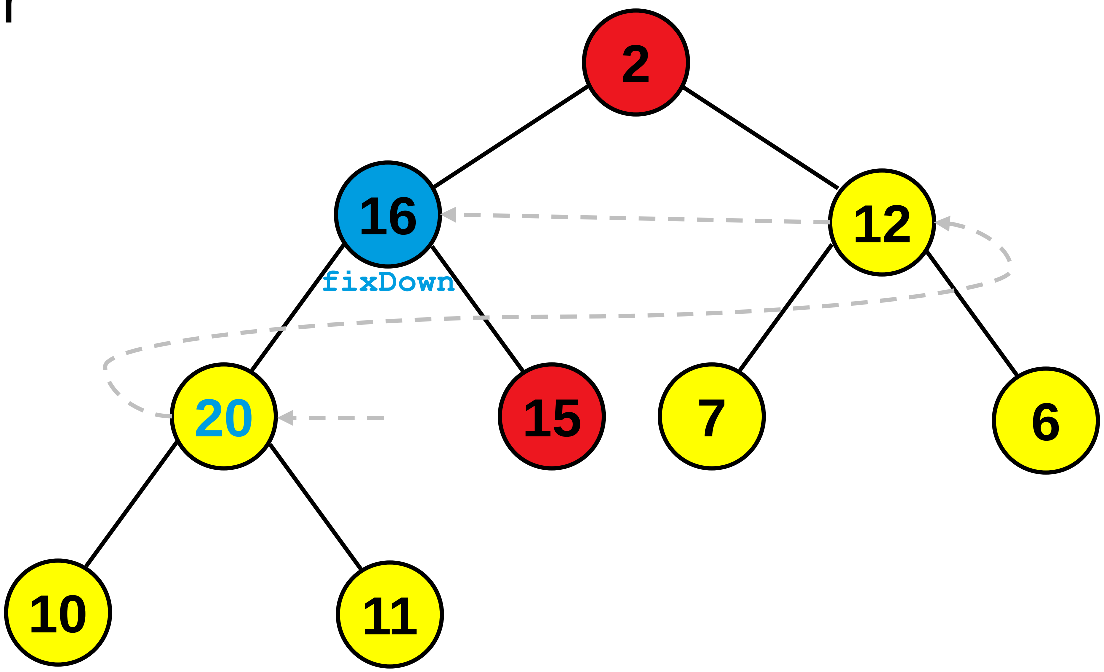
- Chamo `fixDown` do último até ao primeiro elemento do vector



2	16	12	20	15	7	6	10	11

Operações em Amontoados: `buildheap`

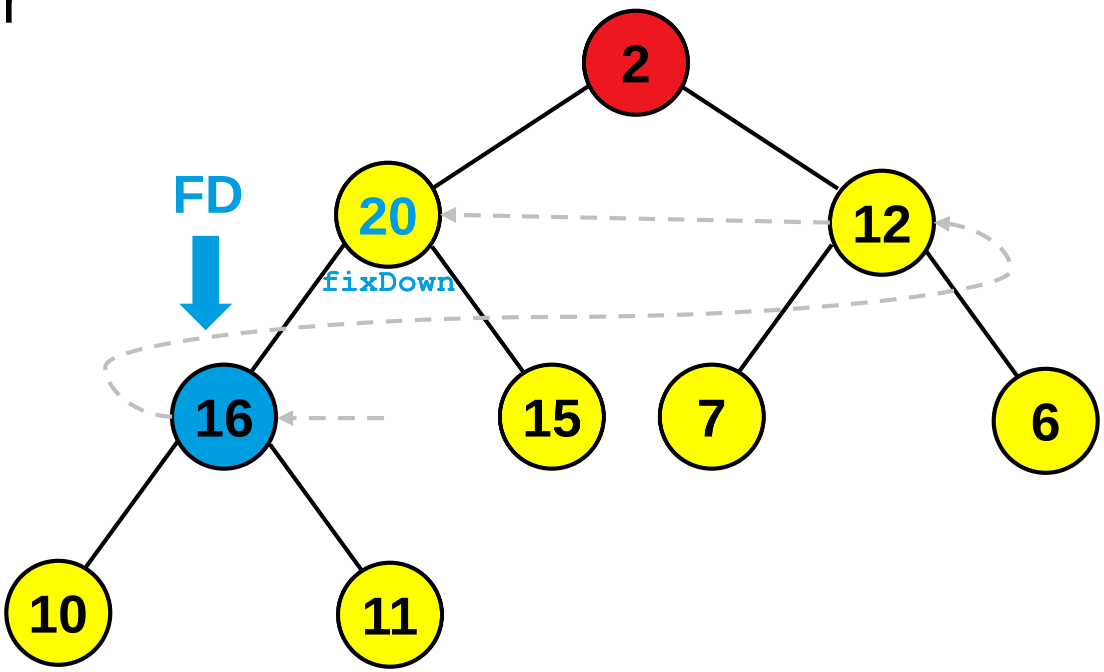
- Chamo `fixDown` do último até ao primeiro elemento do vector



2	16	12	20	15	7	6	10	11

Operações em Amontoados: `buildheap`

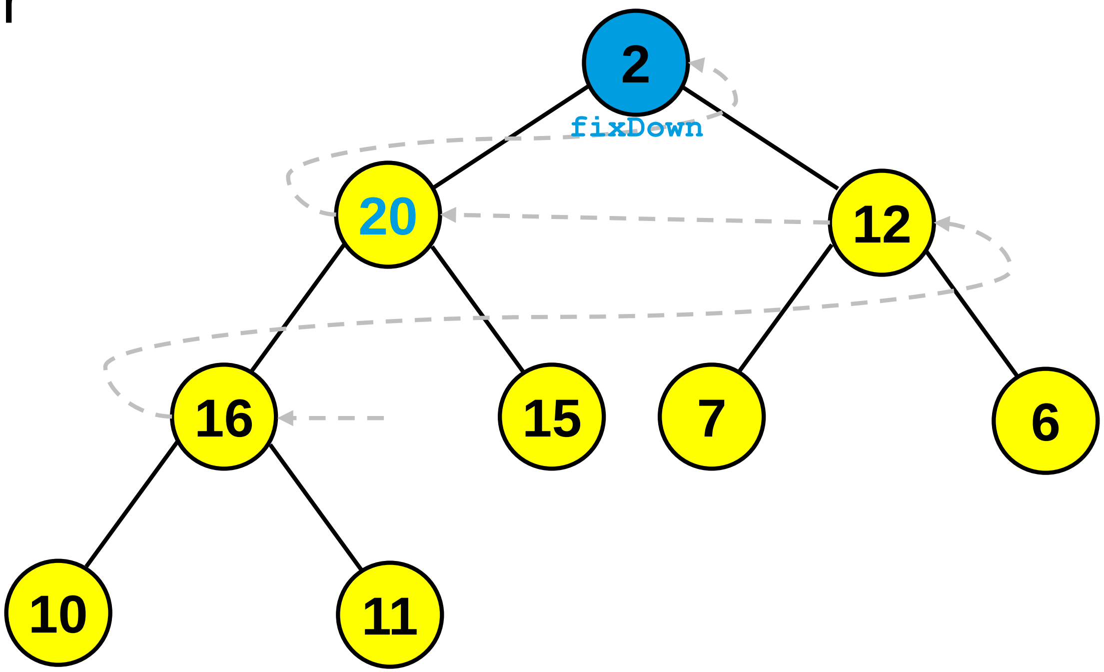
- Chamo `fixDown` do último até ao primeiro elemento do vector



2	20	12	16	15	7	6	10	11

Operações em Amontoados: `buildheap`

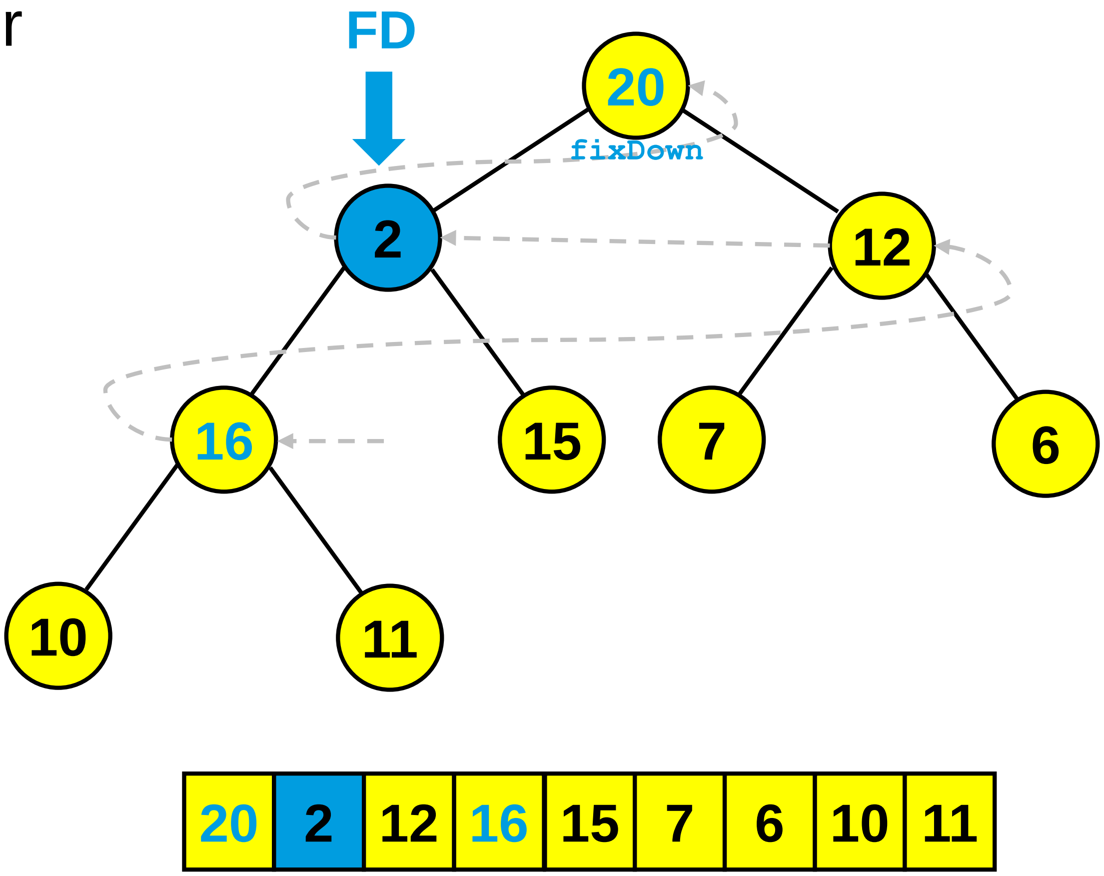
- Chamo `fixDown` do último até ao primeiro elemento do vector



2	20	12	16	15	7	6	10	11

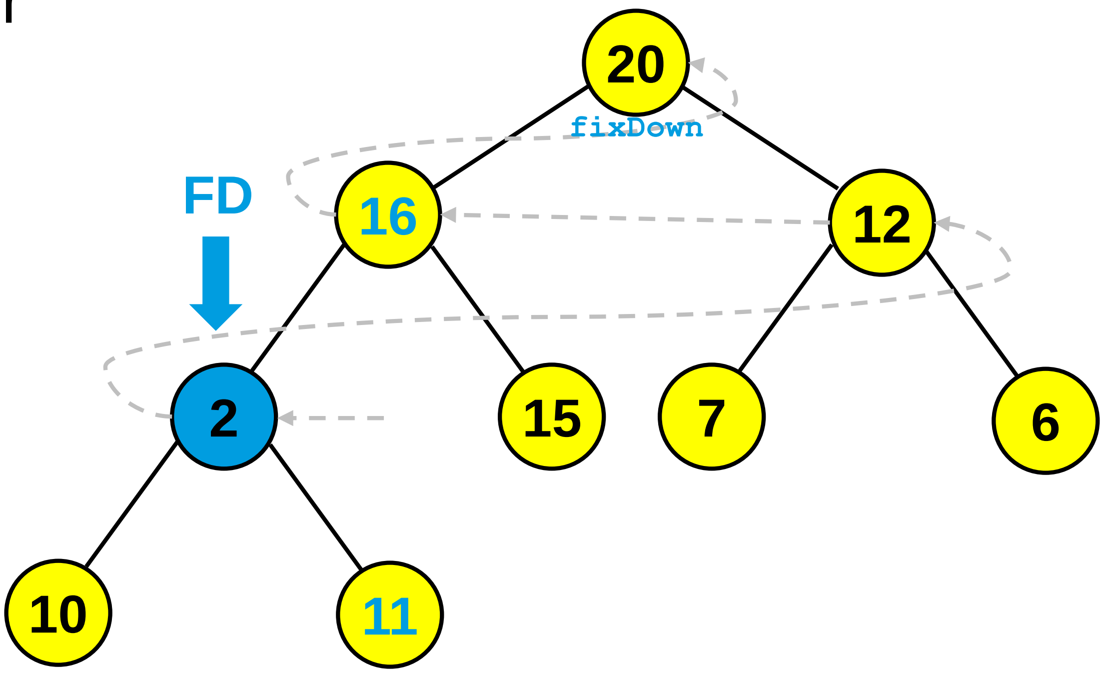
Operações em Amontoados: `buildheap`

- Chamo **`fixDown`** do último até ao primeiro elemento do vector



Operações em Amontoados: `buildheap`

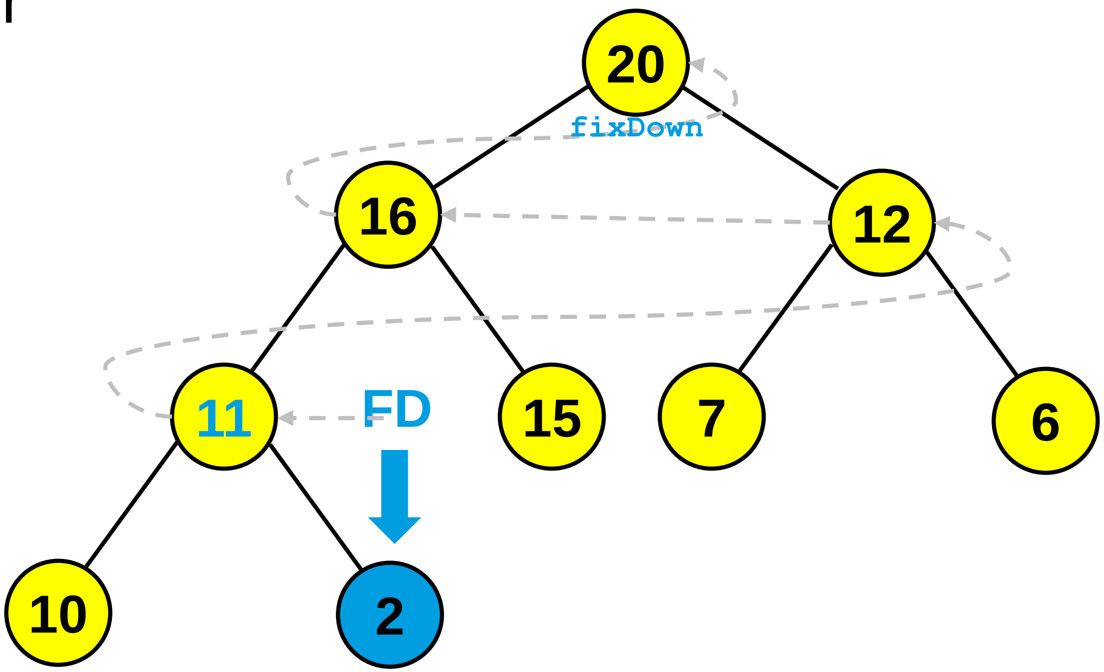
- Chamo `fixDown` do último até ao primeiro elemento do vector



20	16	12	2	15	7	6	10	11

Operações em Amontoados: `buildheap`

- Chamo `fixDown` do último até ao primeiro elemento do vector

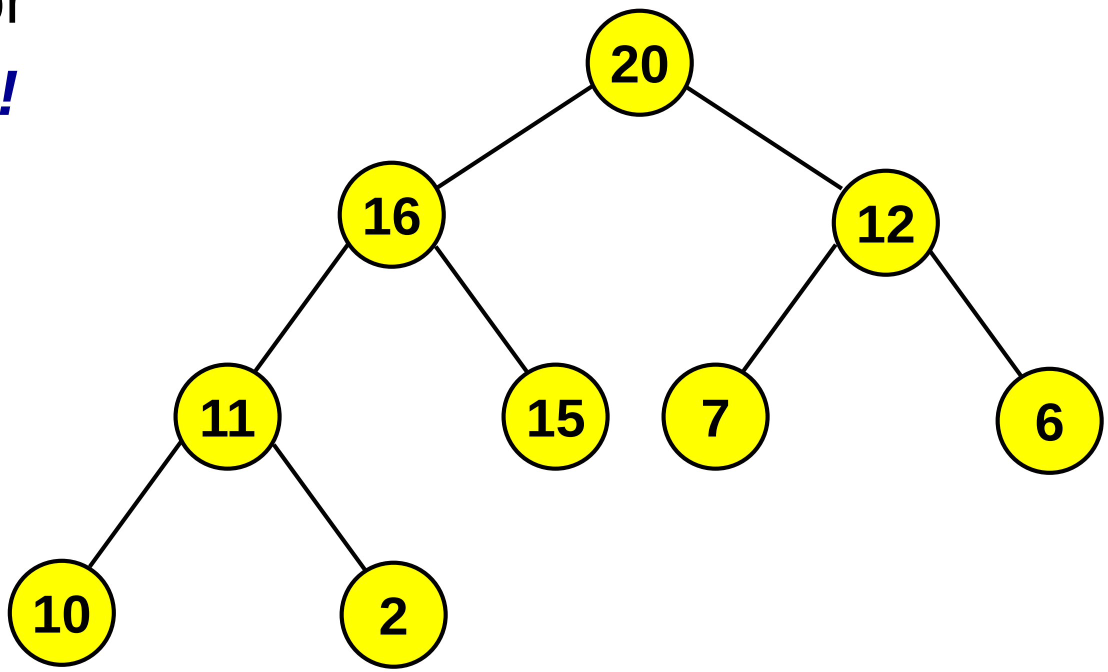


20	16	12	11	15	7	6	10	2

Operações em Amontoados: `buildheap`

- Chamo `fixDown` do último até ao primeiro elemento do vector

Já está!



20	16	12	11	15	7	6	10	2

Operações em Amontoados: `buildheap`

`buildheap`:

•Chama **`fixDown`** do último até ao primeiro elemento do vector até todos os elementos cumprirem a heap condition.

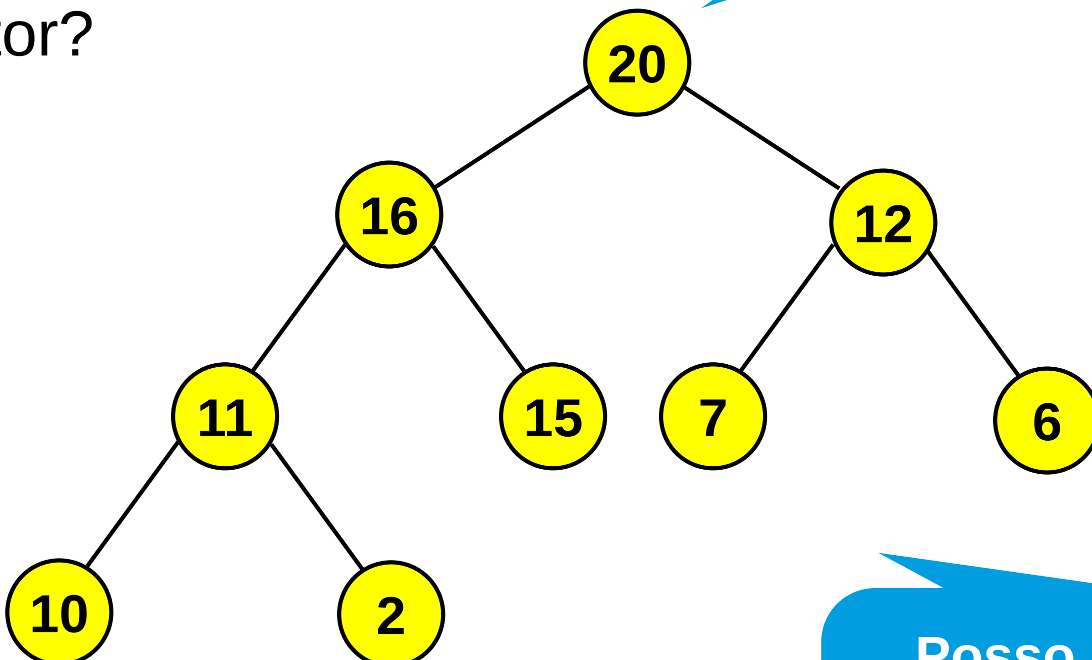
•Uma forma mais rápida de o fazer é começar por chamar o **`fixDown`** ao pai com o índice mais elevado (K=heapsize/2-1), e a todos os índices<K

```
void buildheap(Item a[], int l, int r){
    int k, heapsize = r-l+1;
    for (k = heapsize/2-1; k >= l; k--)
        fixDown(a, l, r, l+k);
}
```

HeapSort

Se tiver um amontoado, sei sempre quem é o maior.

Como posso partir destas estruturas para ordenar um vector?



... Posso ir trocando com o ultimo, reconstruindo o amontoado em cada passo.

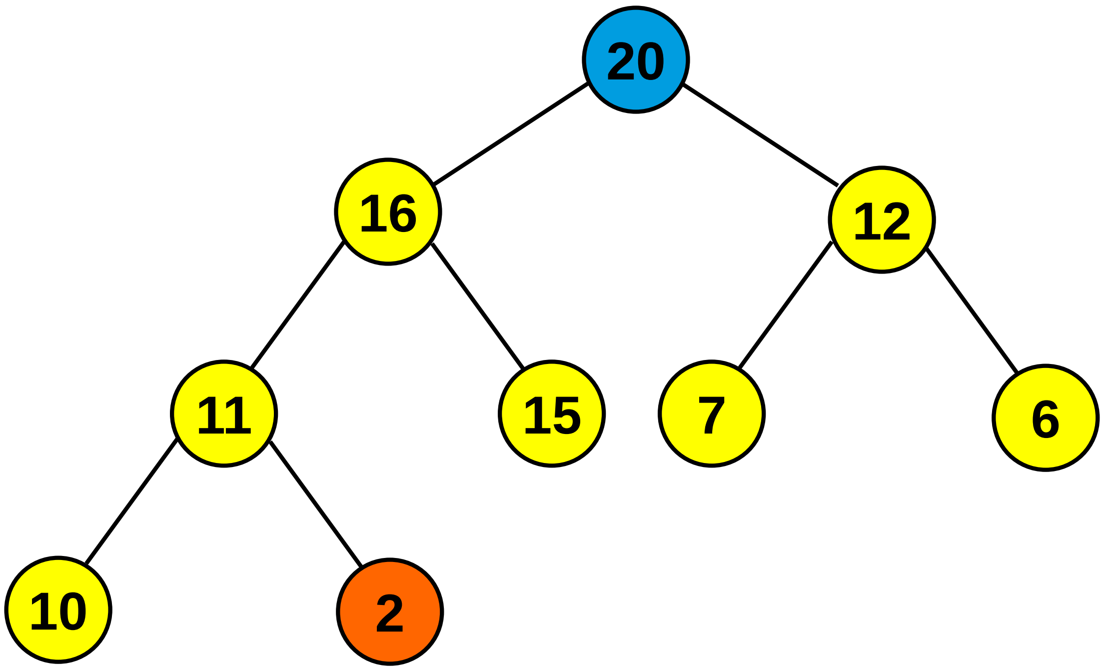
20	16	12	11	15	7	6	10	2

HeapSort - Algoritmo

1. transforma o vector num amontoado;
2. troca o primeiro elemento (raiz) com o último;
3. reduz a dimensão do amontoado em uma unidade;
4. aplica o **`fixDown`** sobre a nova raiz para reparar o amontoado;

Heapsort

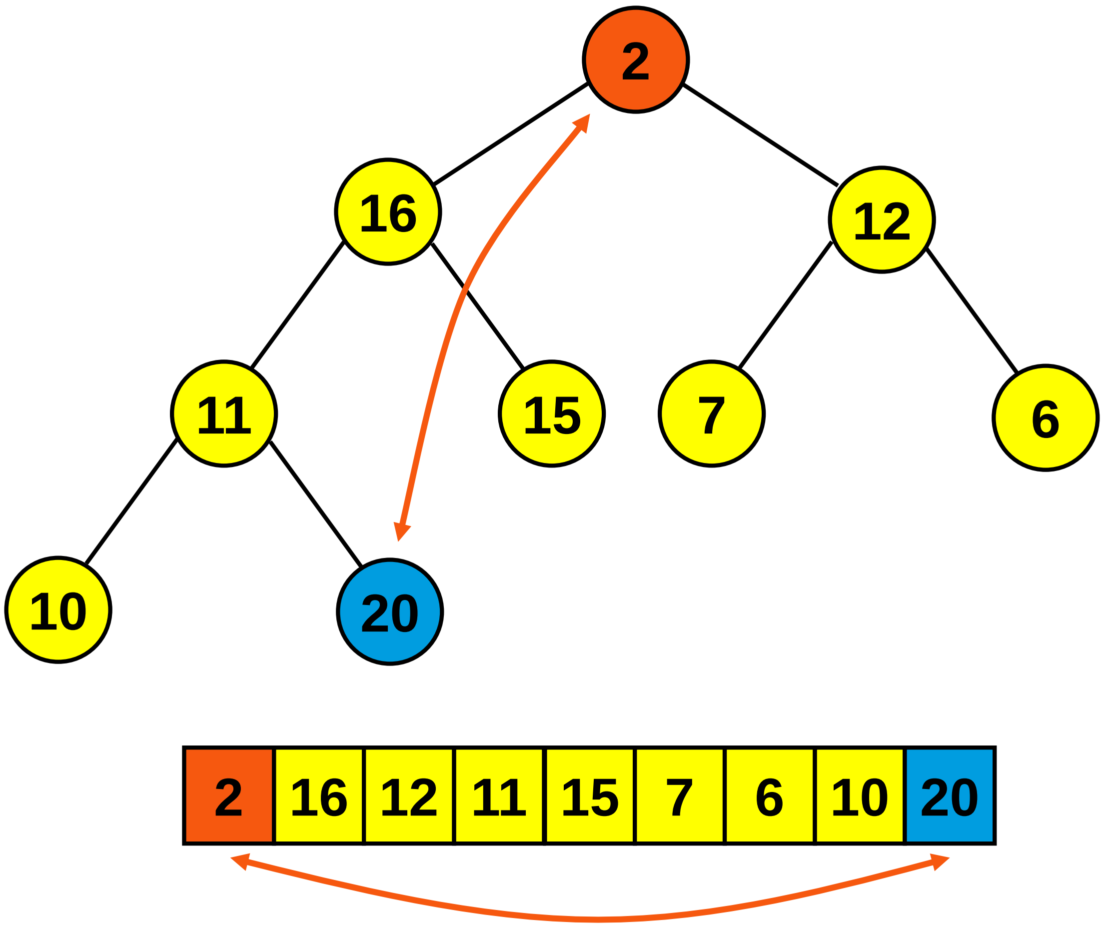
- Trocar o primeiro elemento (raiz) com o último



20	**16**	**12**	**11**	**15**	**7**	**6**	**10**	**2**

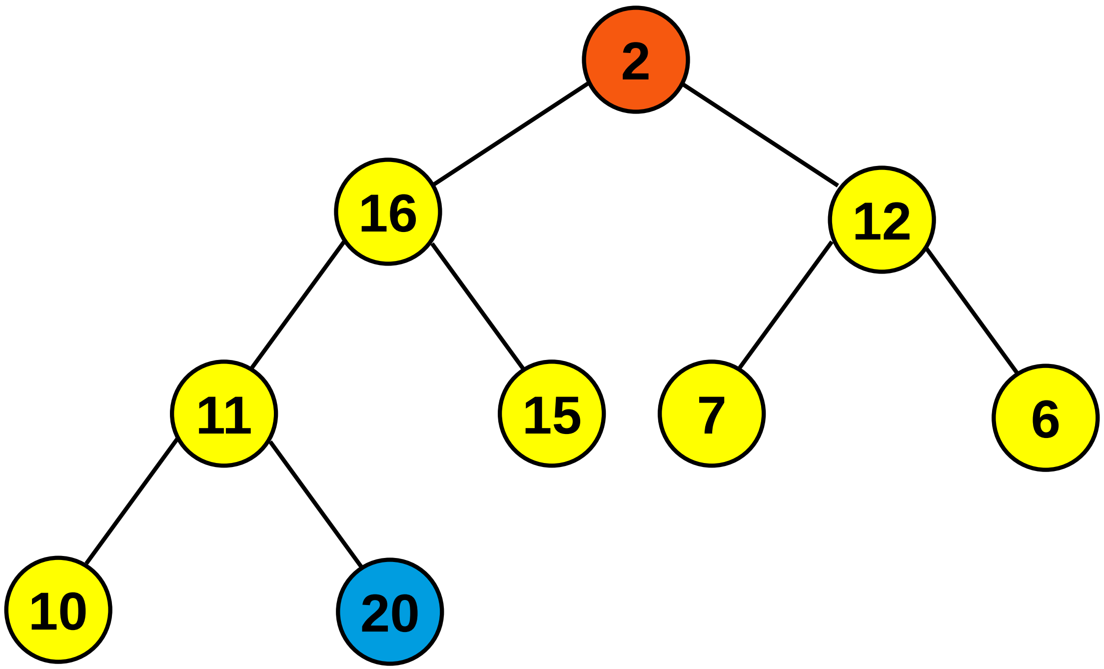
Heapsort

- Trocar o primeiro elemento (raiz) com o último



Heapsort

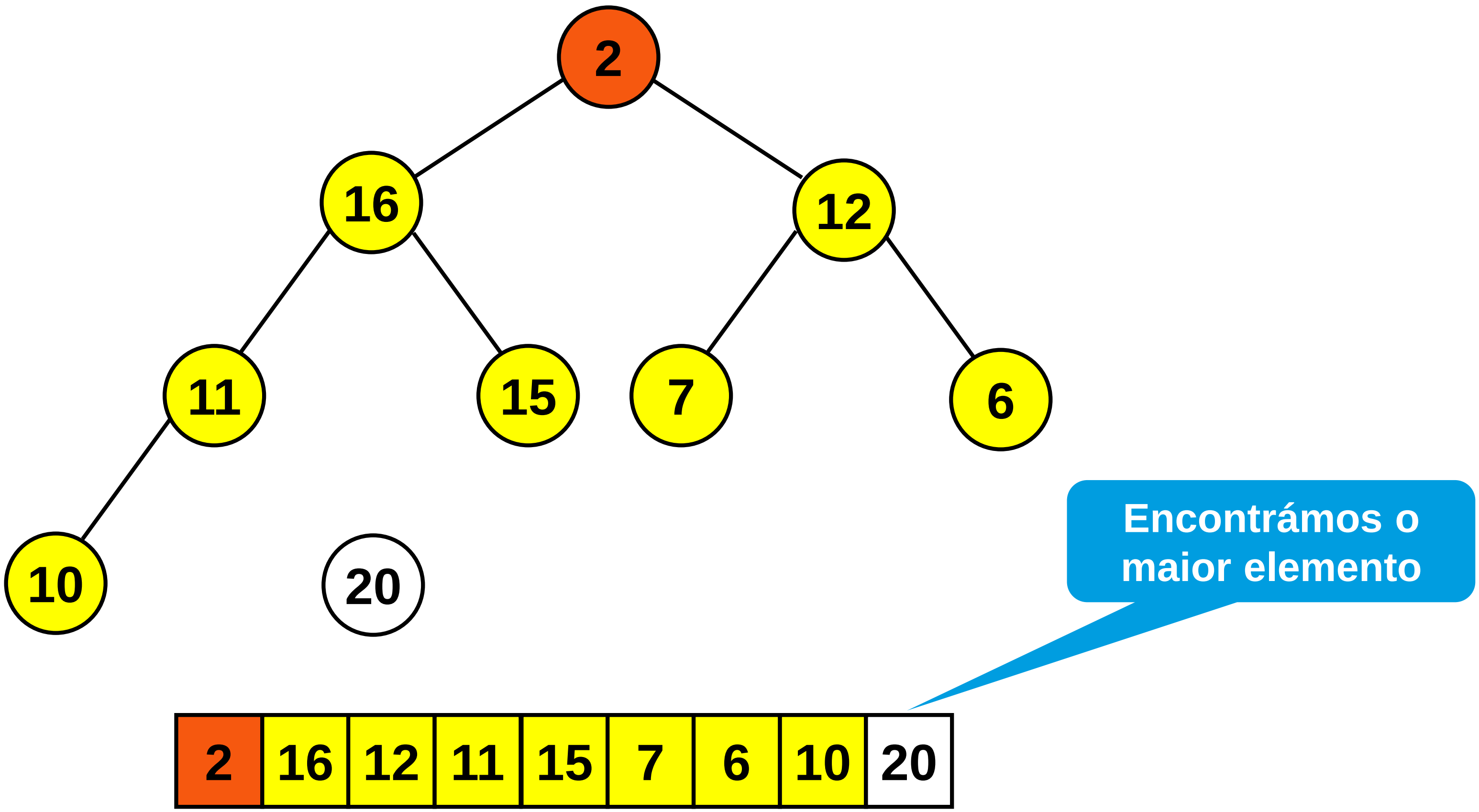
- Reduzir a dimensão do amontoado em uma unidade



2	16	12	11	15	7	6	10	20

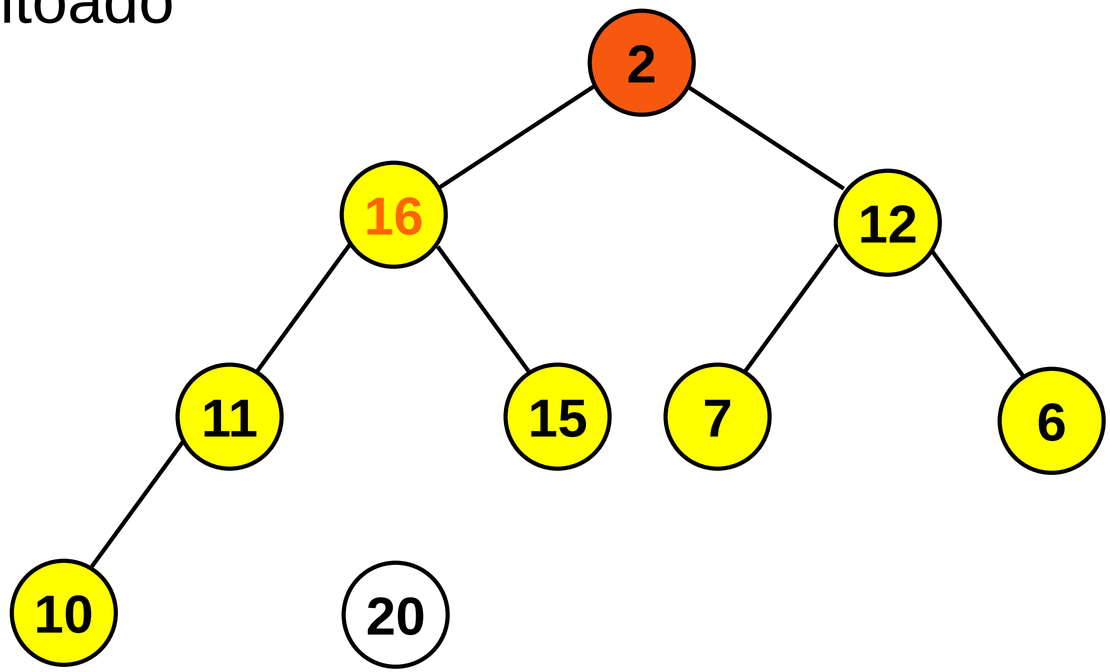
Heapsort

- Reduzir a dimensão do amontoado em uma unidade



Heapsort

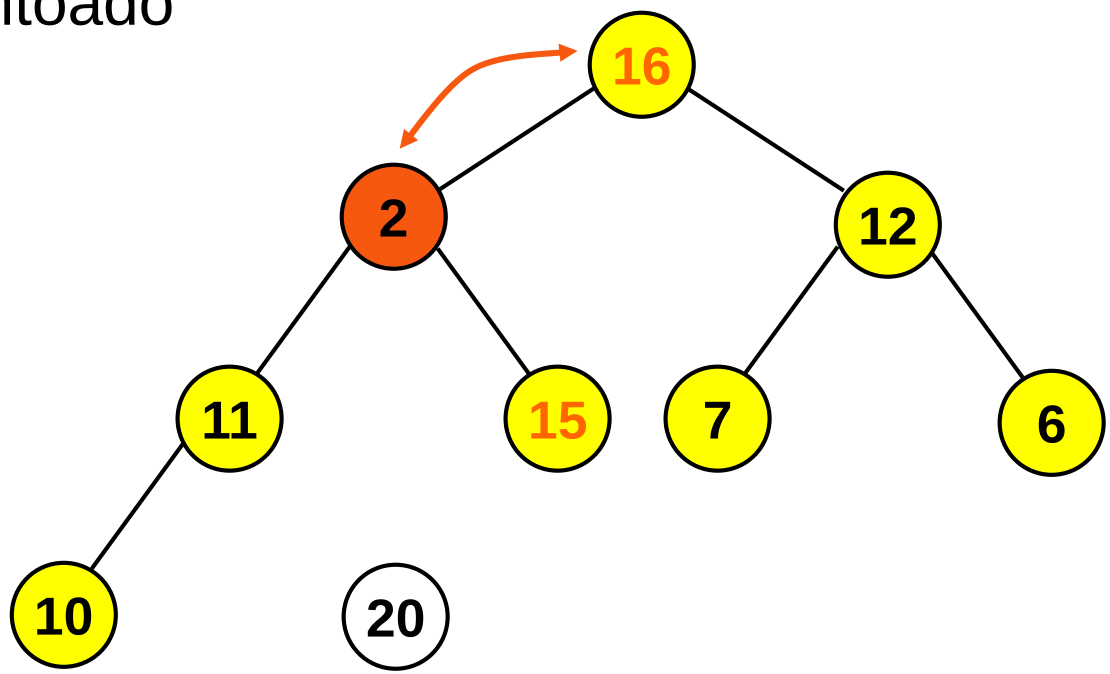
- Executar **`fixDown`** sobre a nova raiz para reparar o amontoado



2	16	12	11	15	7	6	10	20

Heapsort

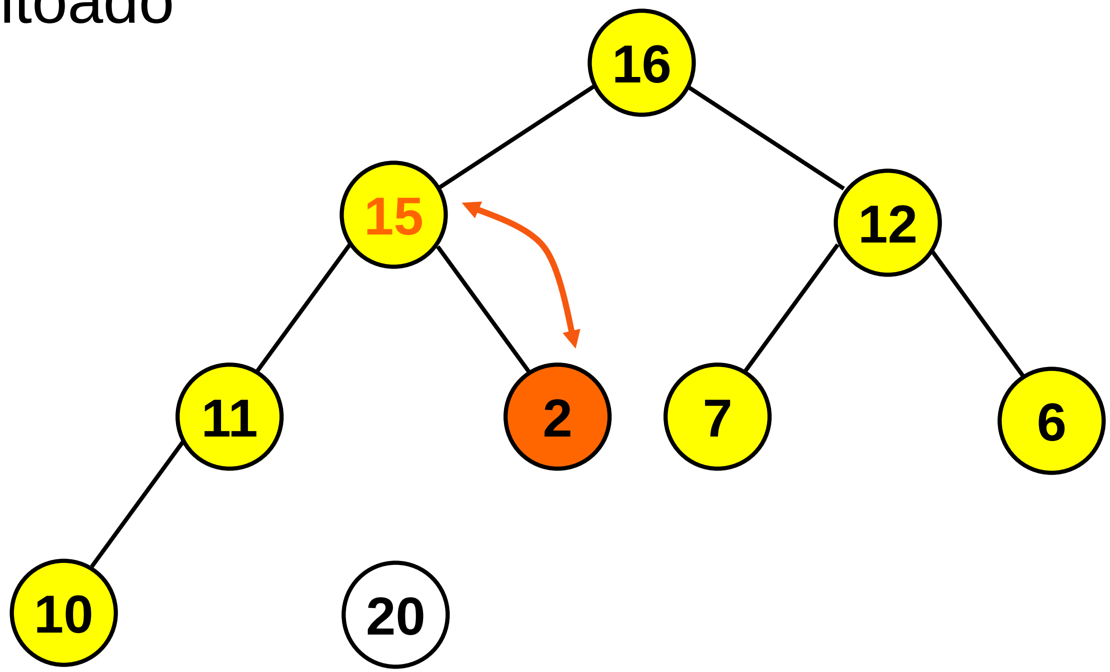
- Executar **`fixDown`** sobre a nova raiz para reparar o amontoado



16	2	12	11	15	7	6	10	20

Heapsort

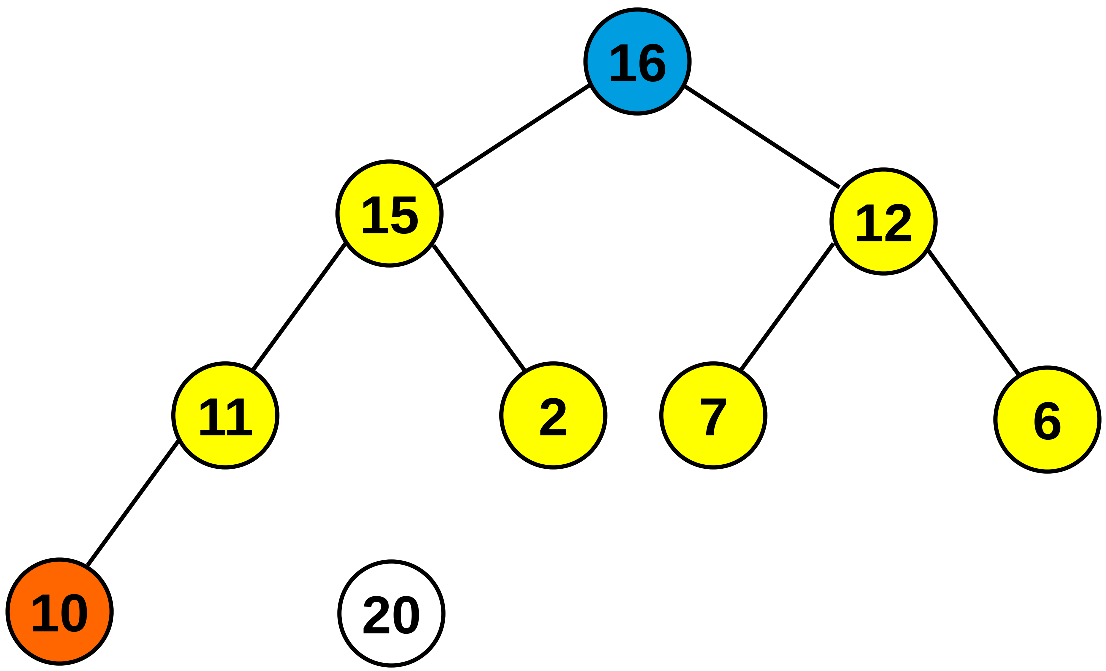
- Executar **`fixDown`** sobre a nova raiz para reparar o amontoado



16	15	12	11	2	7	6	10	20

Heapsort

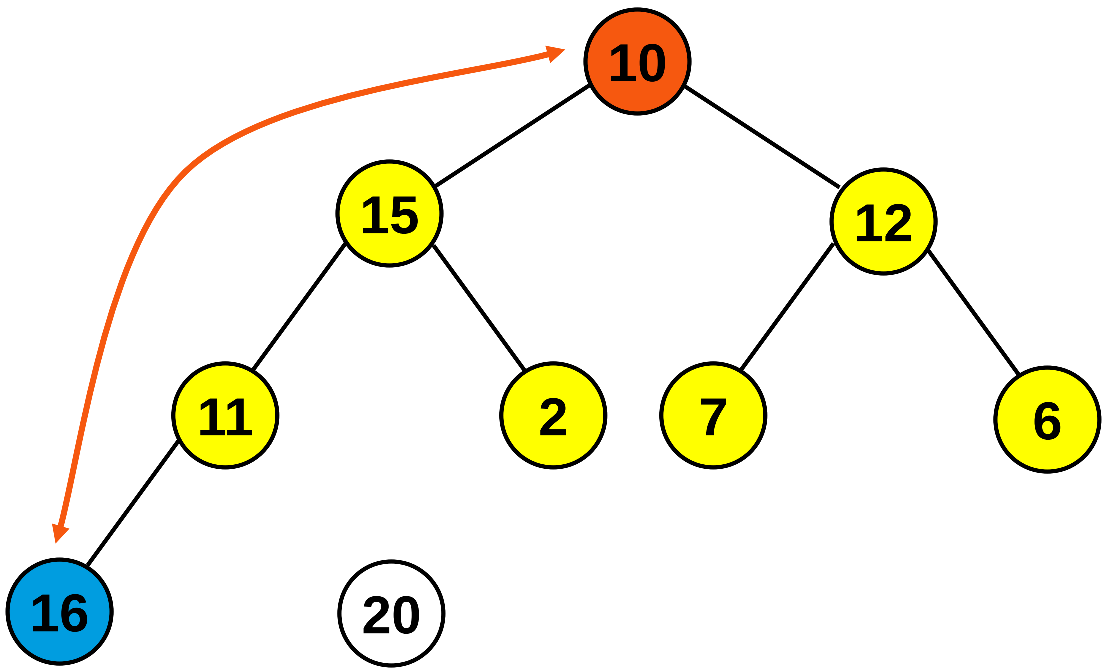
- Repetir



16	15	12	11	2	7	6	10	20

Heapsort

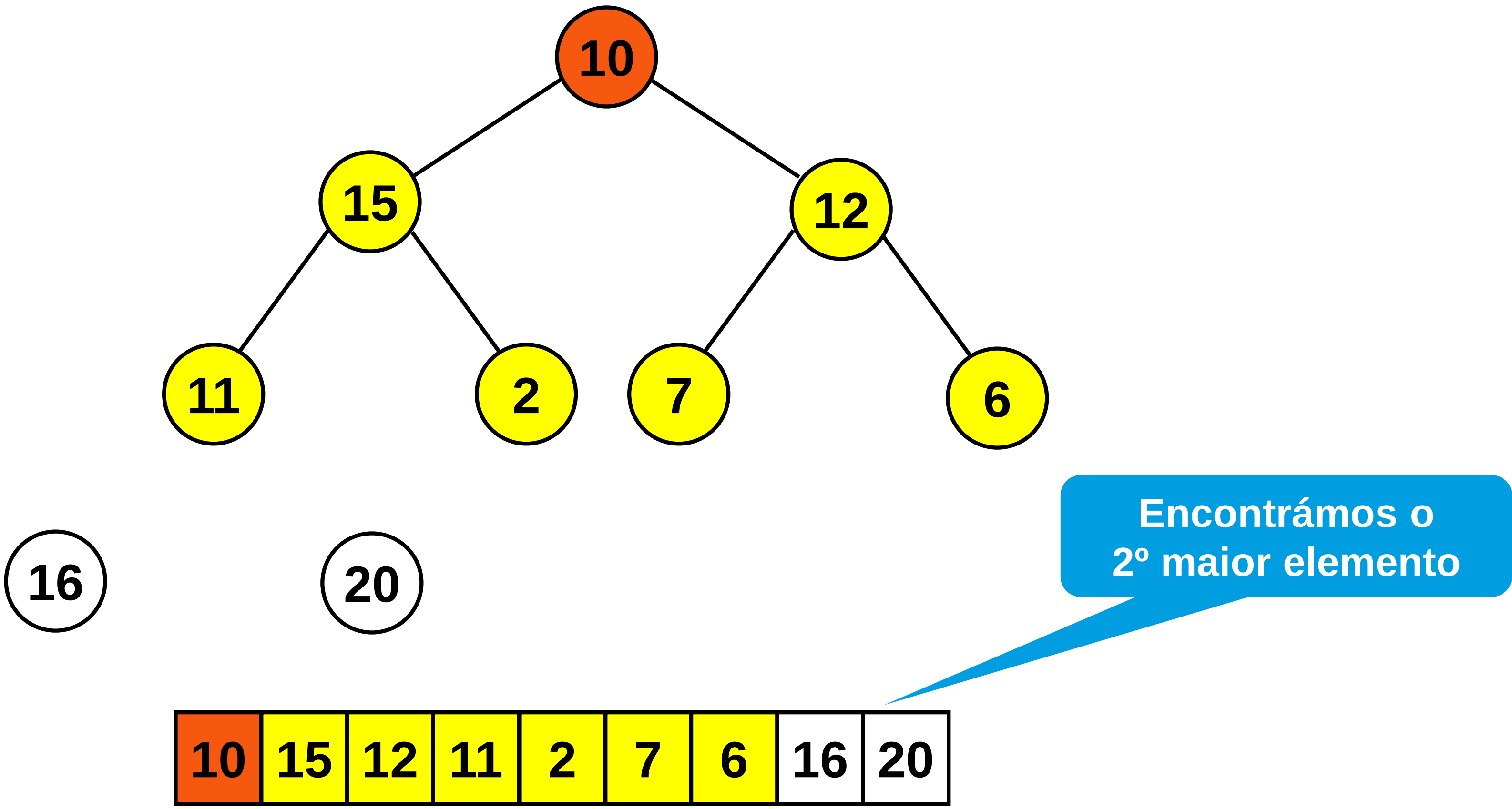
- Repetir



10	15	12	11	2	7	6	16	20

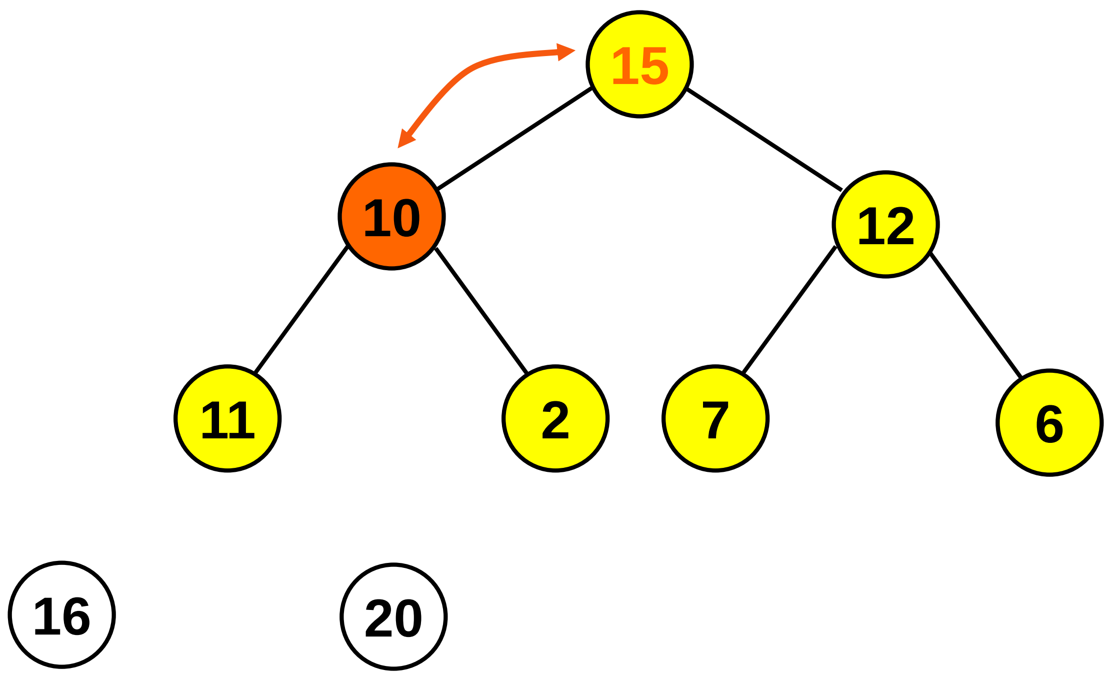
Heapsort

- Repetir



Heapsort

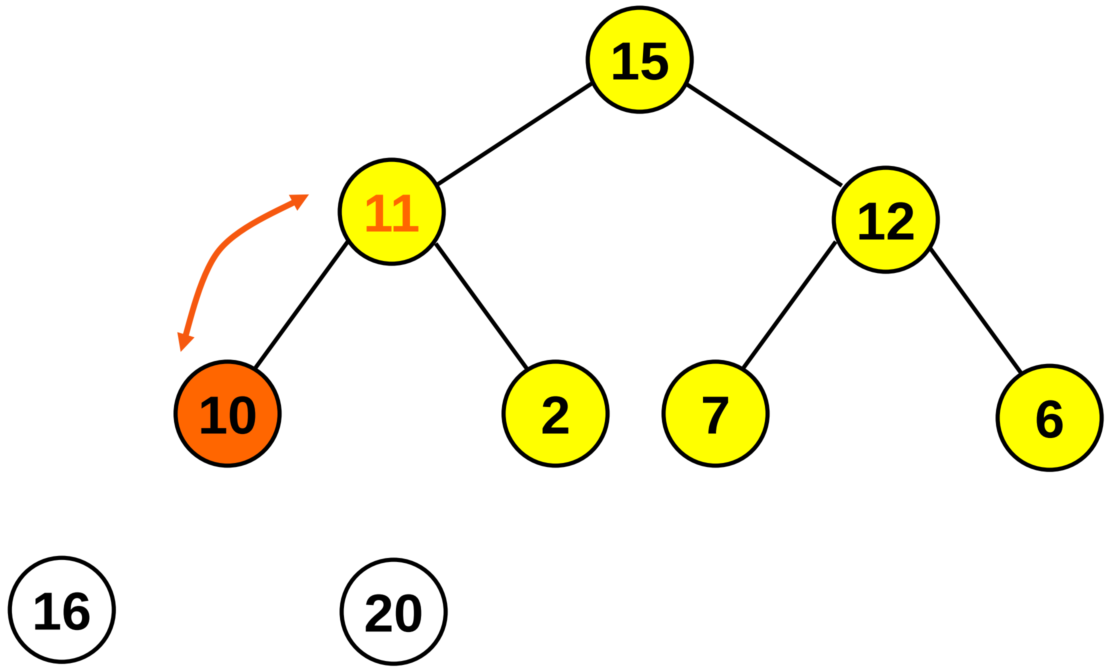
- Repetir



15	10	12	11	2	7	6	16	20

Heapsort

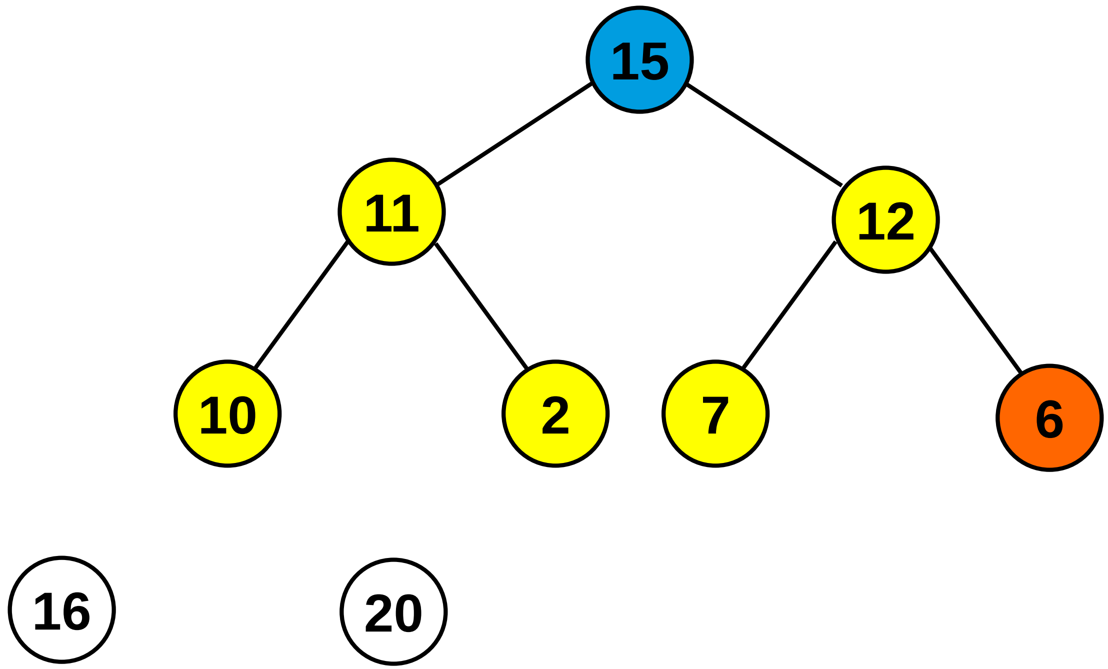
- Repetir



15	11	12	10	2	7	6	16	20

Heapsort

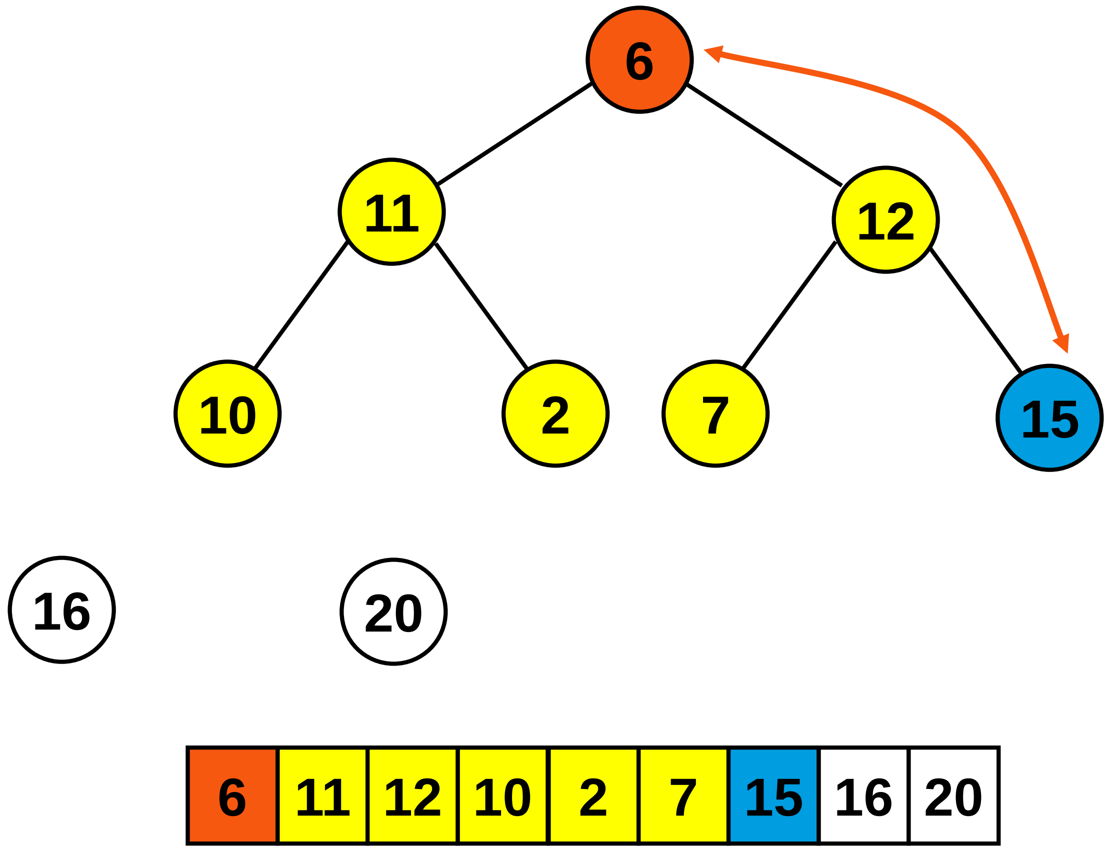
- Repetir



15	11	12	10	2	7	6	16	20

Heapsort

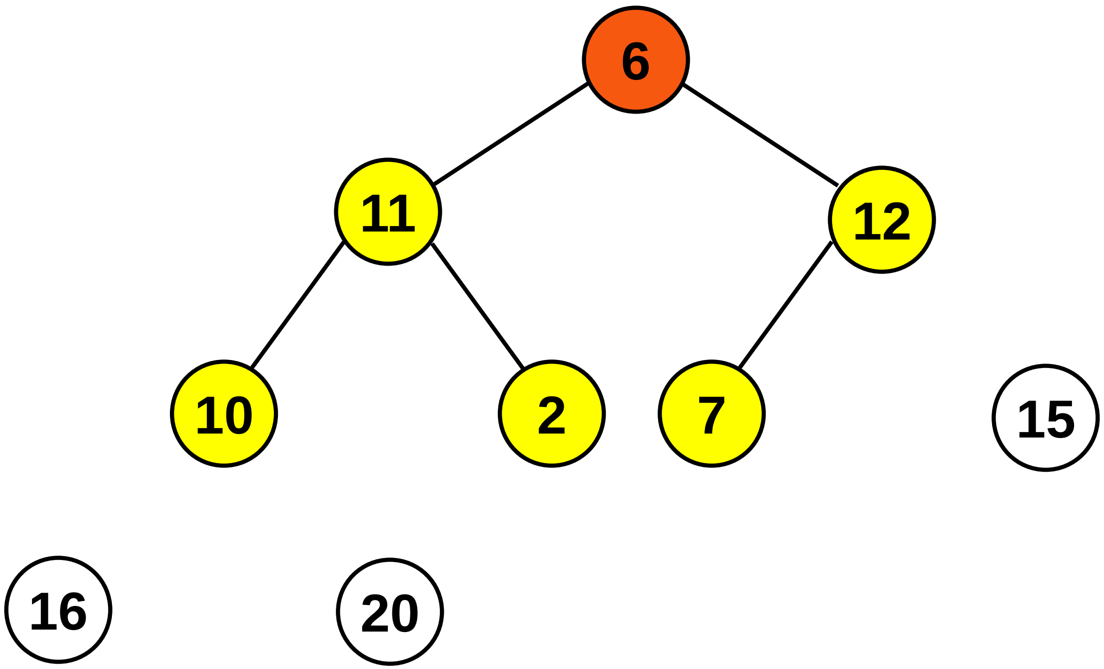
- Repetir



IAED TÉCNICO LISBOA

Heapsort

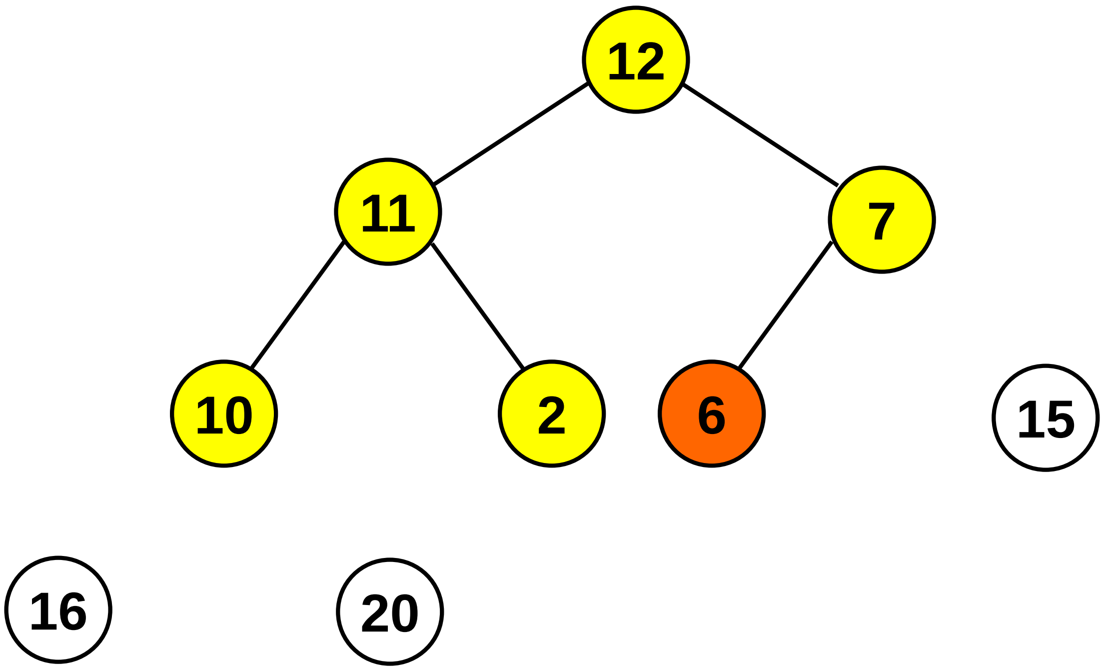
- Repetir



6	11	12	10	2	7	15	16	20

Heapsort

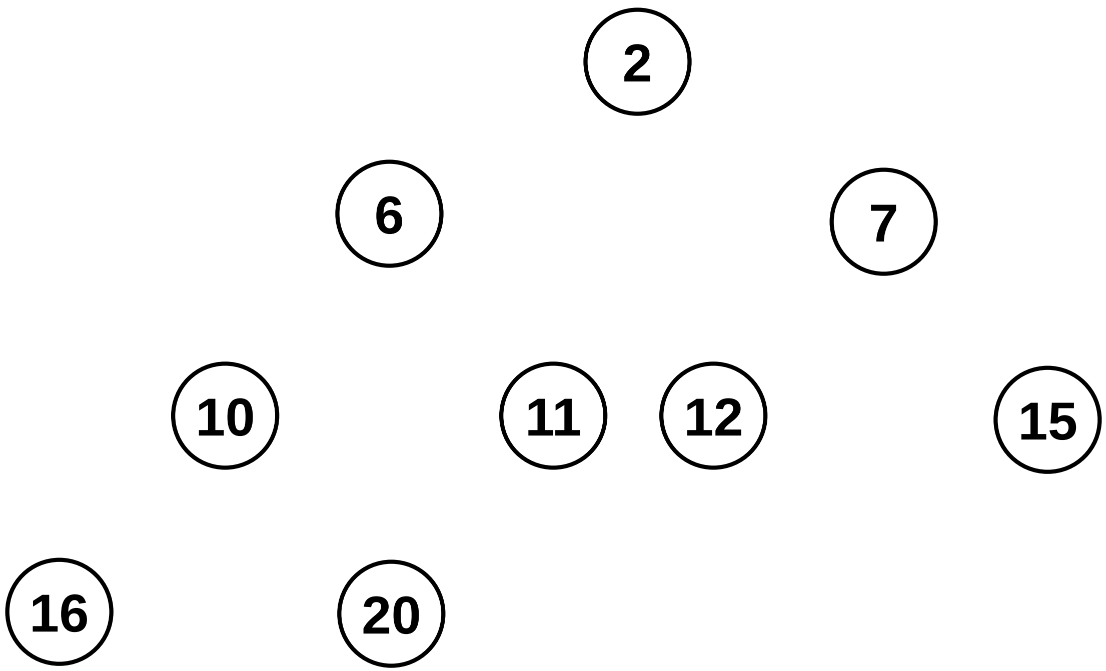
- Repetir ...



12	11	7	10	2	6	15	16	20

Heapsort

- No final



2	6	7	10	11	12	15	16	20

Heapsort

Começo por transformar o vector num amontoado

```
void heapsort(Item a[], int l, int r)
{
    buildheap(a,l,r);
    while (r-l > 0) {
        exch(a[l], a[r]);
        fixDown(a, l, --r, l);
    }
}
```

Troco o primeiro com o último elemento do vector (i.e., troco a raiz com o último elemento da árvore)

Reduzo a dimensão do meu amontoado

Executo estes 3 últimos passos até ordenar todos os elementos

...e chamo o `fixDown` à raiz, de forma a voltar a transformar o (sub-)vector num amontoado

Exercício

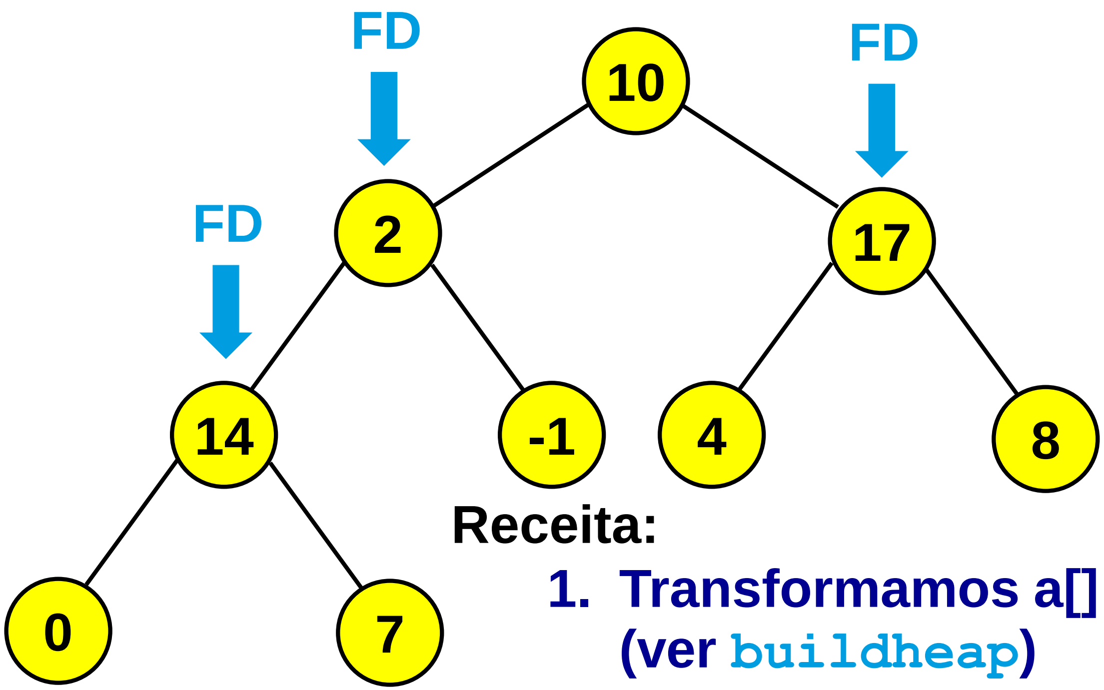
- Considere que se pretende ordenar o vector em baixo, utilizando o algoritmo de ordenação ***heapsort***. Qual será a configuração do vector após 2 iterações do heapsort?

```
a = { 10, 2, 17, 14, -1, 4, 8, 0, 7 }
```

- **Receita**:
  1. Transformamos a[] num amontoado (ver `buildheap`)
  2. Trocamos o ultimo elemento com a raiz, e extraímos esse elemento do amontoado.
  3. Aplicamos o `fixDown` à raiz
  4. Voltamos a 2.

Exercício

- Começo por transformar o vector num amontoado.

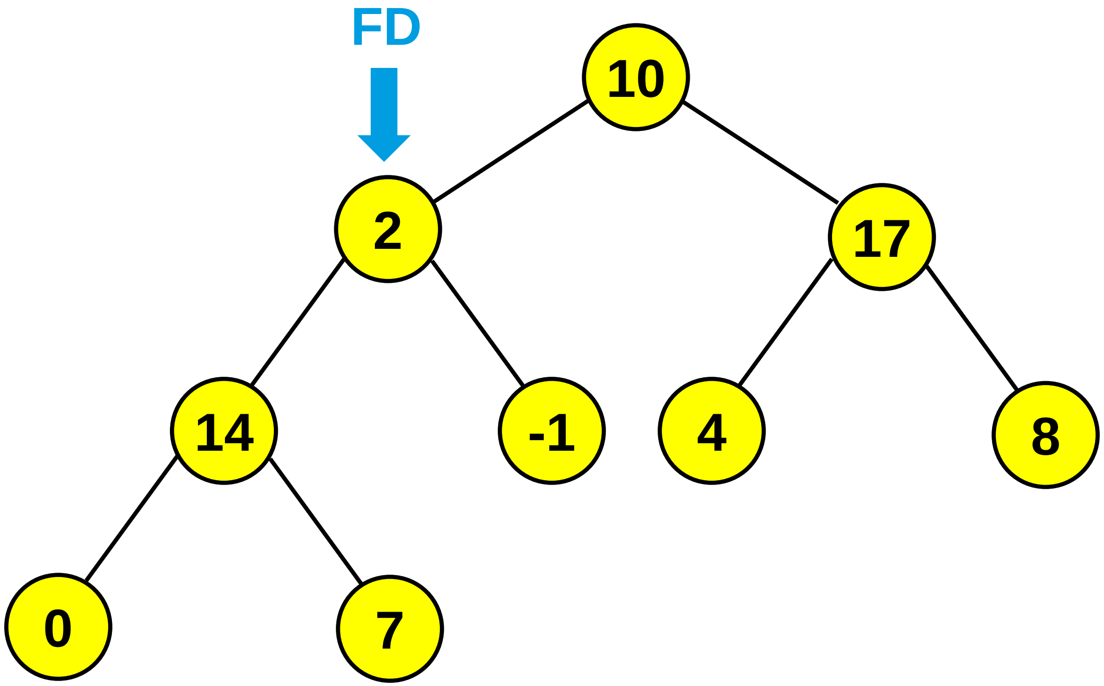


Receita:

1. **Transformamos a[] num amontoado (ver `buildheap`)**
2. **Trocamos o ultimo elemento com a raiz, e extraímos esse elemento do amontoado.**
3. **Aplicamos o `fixDown` à raiz**
4. **Voltamos a 2.**

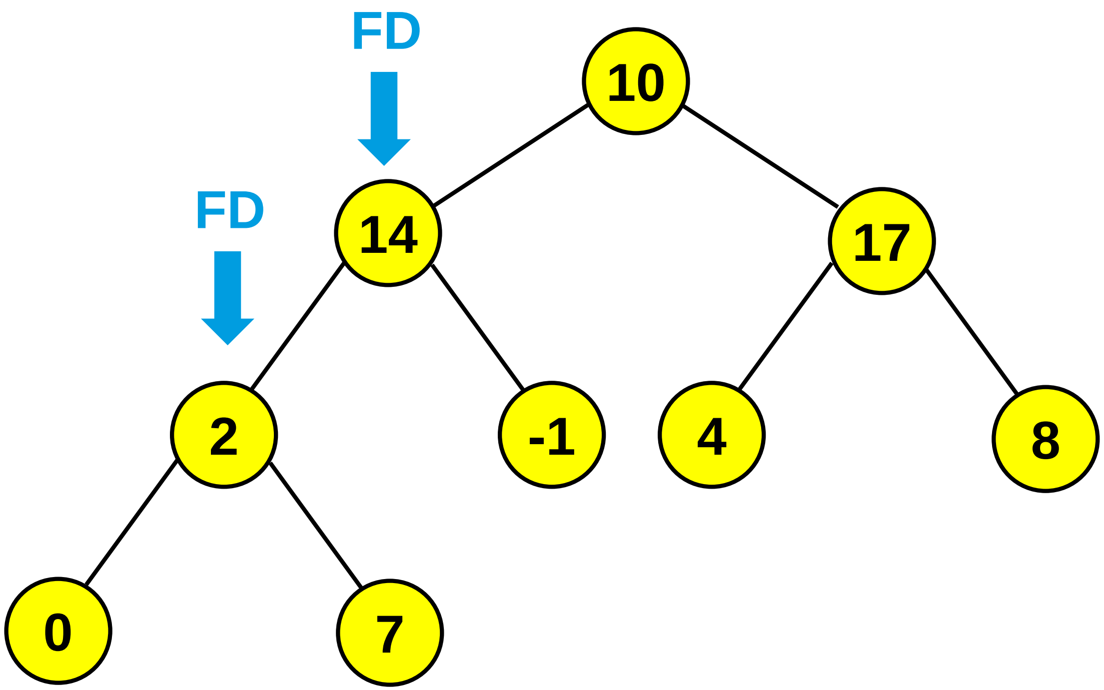
Exercício

- Começo por transformar o vector num amontoado.



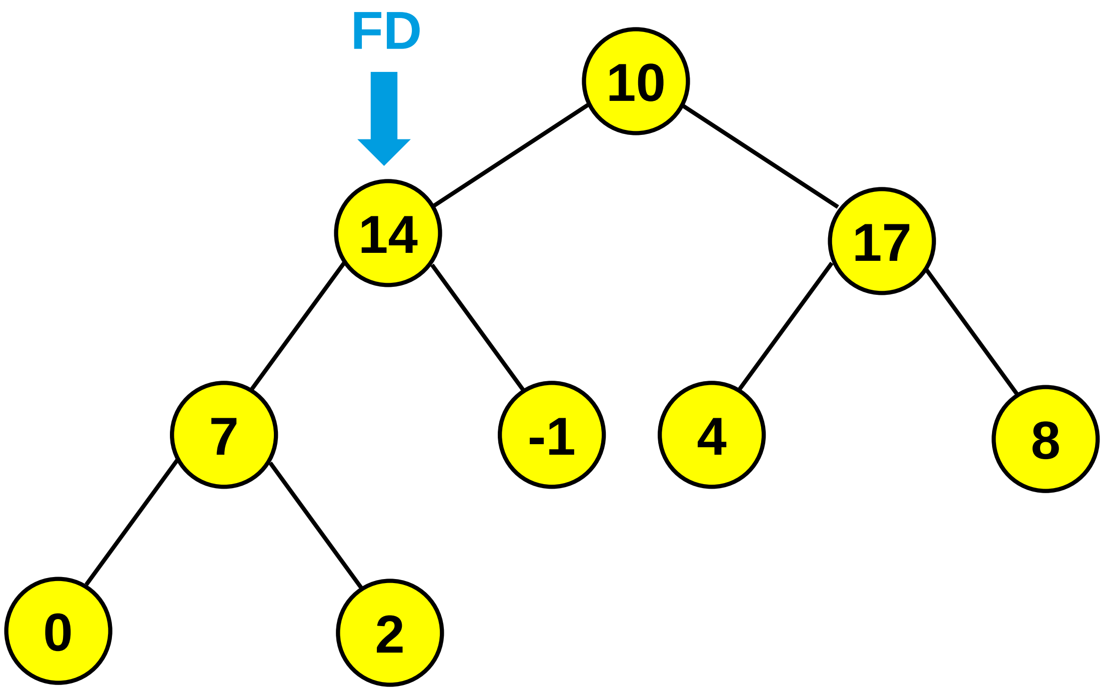
Exercício

- Começo por transformar o vector num amontoado.



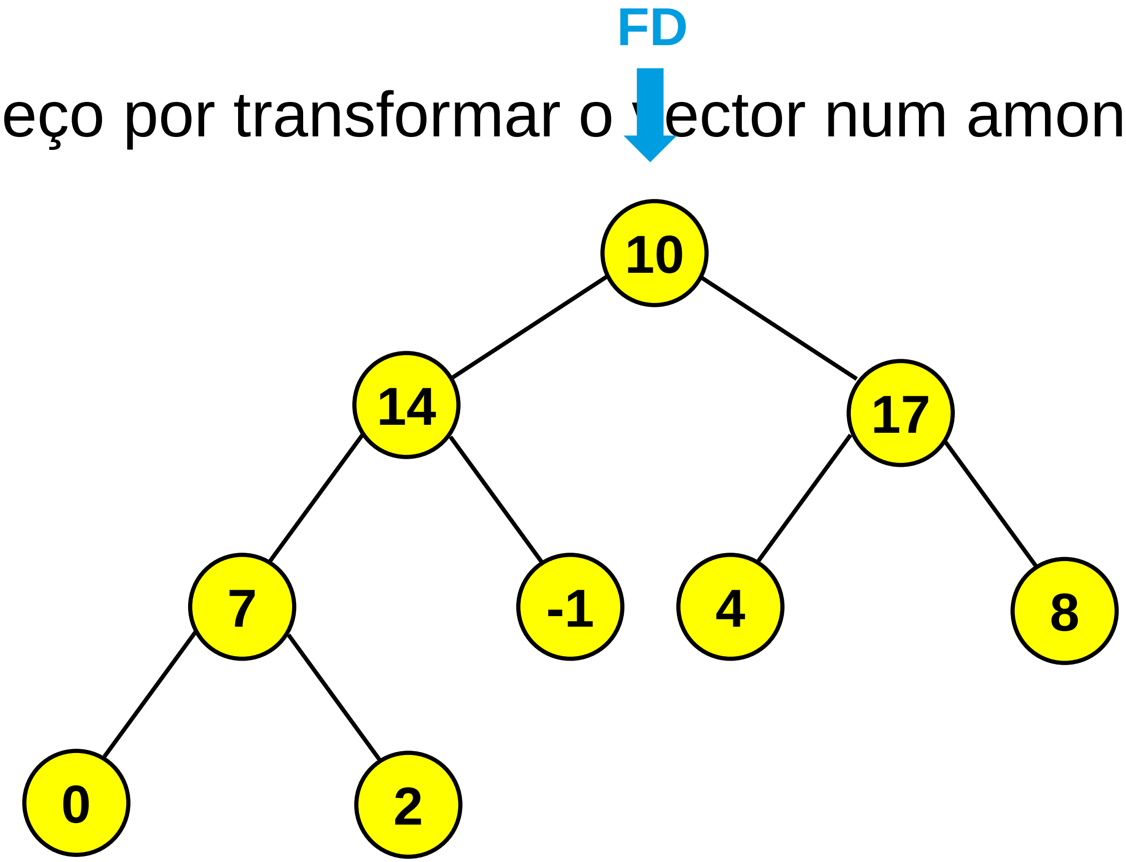
Exercício

- Começo por transformar o vector num amontoado.



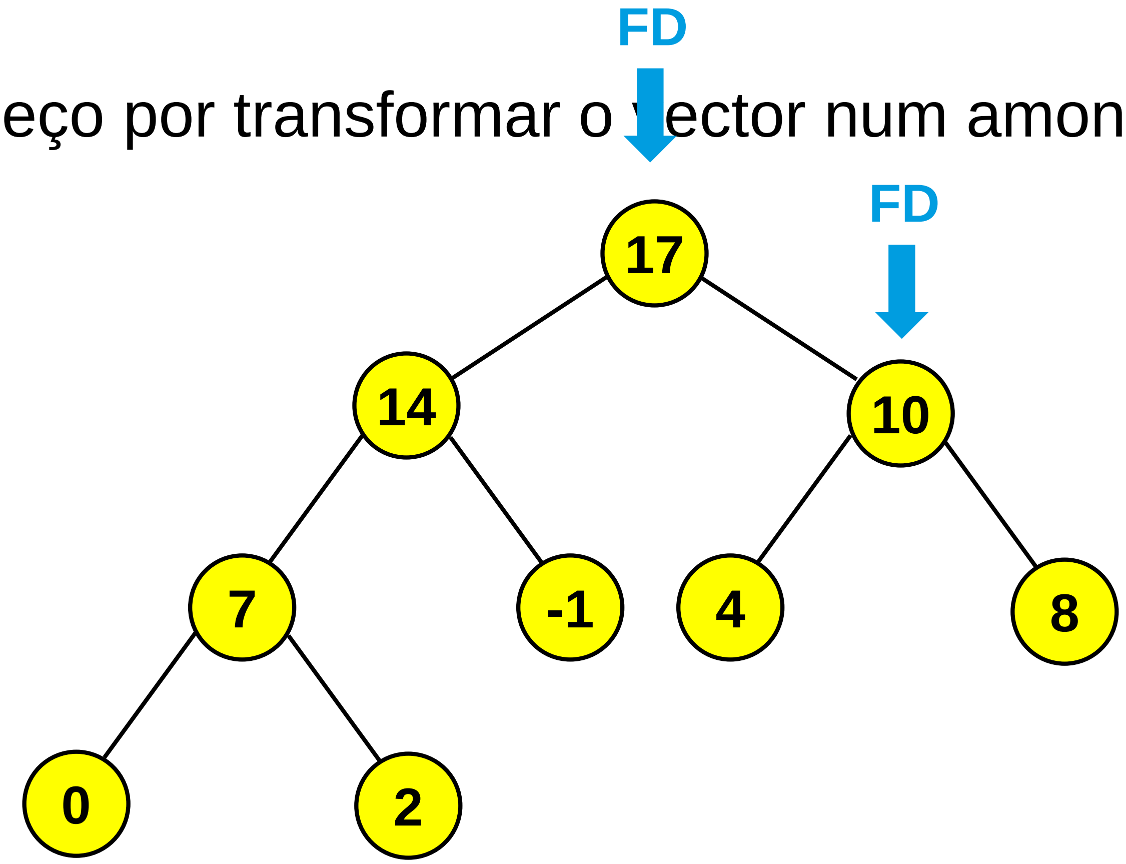
Exercício

- Começo por transformar o vector num amontoado.



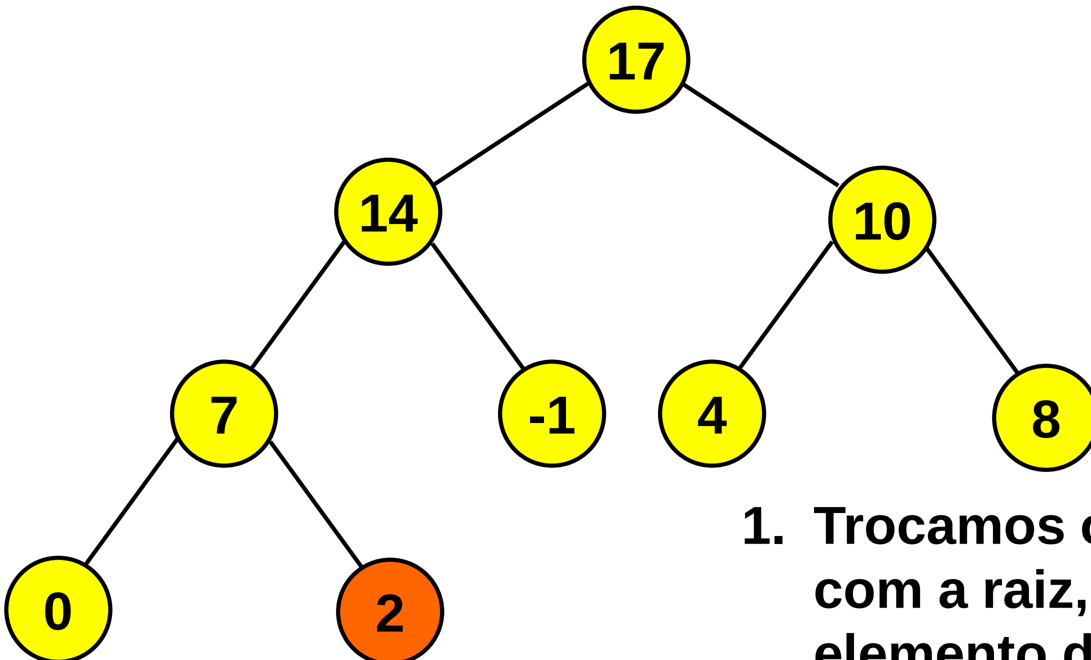
Exercício

- Começo por transformar o vector num amontoado.



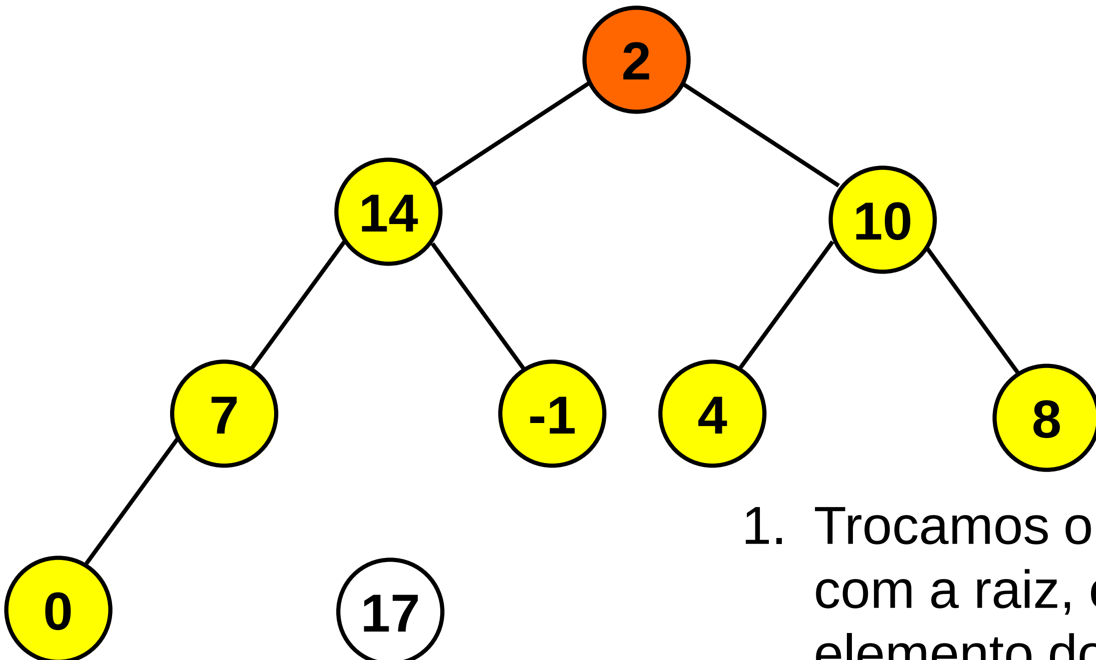
Exercício

- Agora já tenho um amontoado!!



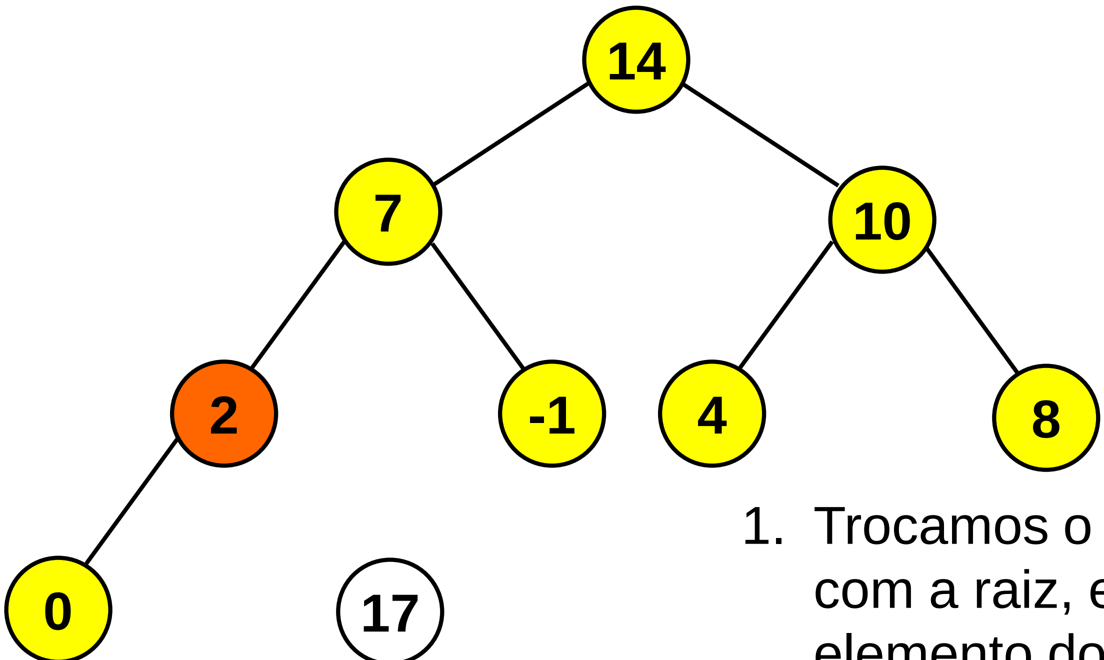
1. **Trocamos o ultimo elemento com a raiz, e extraímos esse elemento do amontoado.**
2. Aplicamos o `fixDown` à raiz
3. Voltamos a 1.

Exercício



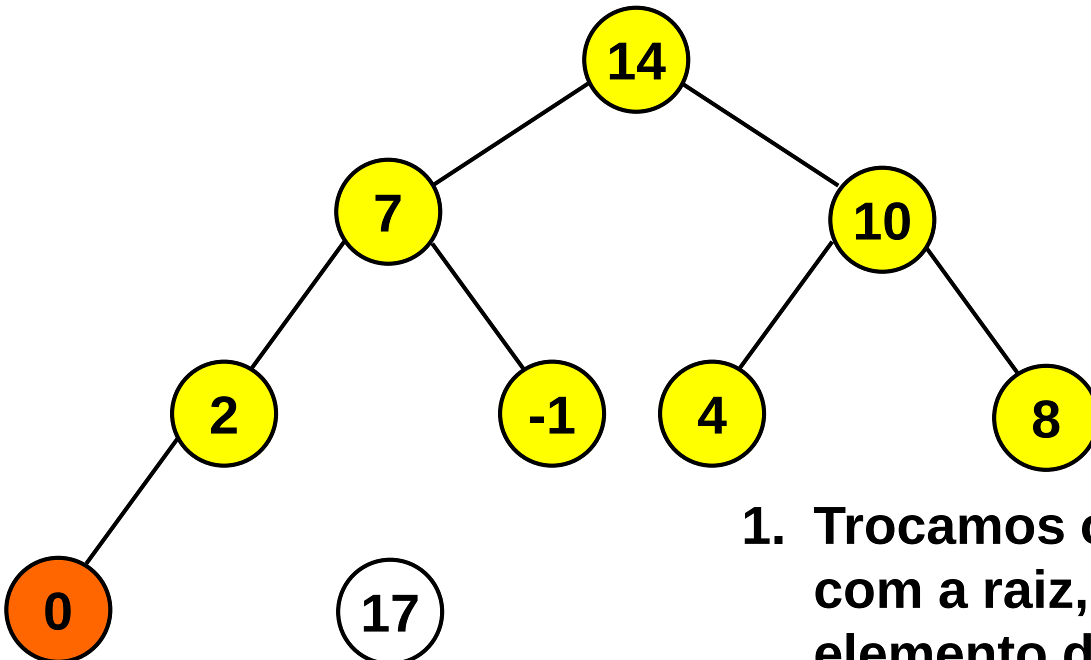
1. Trocamos o ultimo elemento com a raiz, e extraímos esse elemento do amontoado.
2. **Aplicamos o `fixDown` à raiz**
3. Voltamos a 1.

Exercício



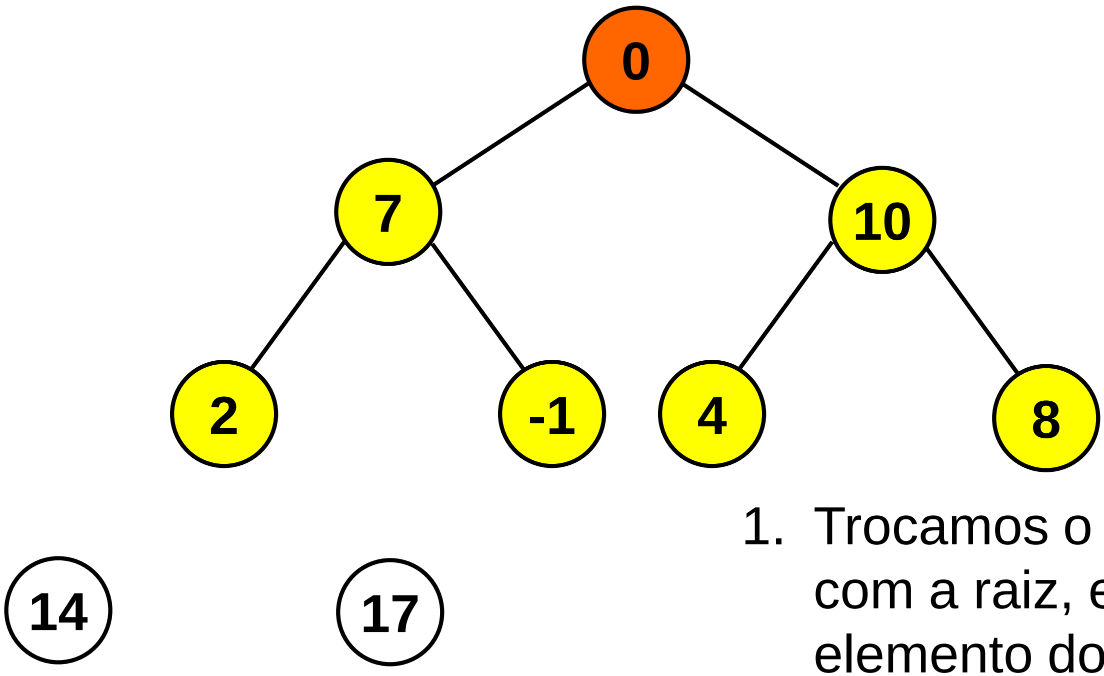
1. Trocamos o ultimo elemento com a raiz, e extraímos esse elemento do amontoado.
2. Aplicamos o `fixDown` à raiz
3. **Voltamos a 1.**

Exercício



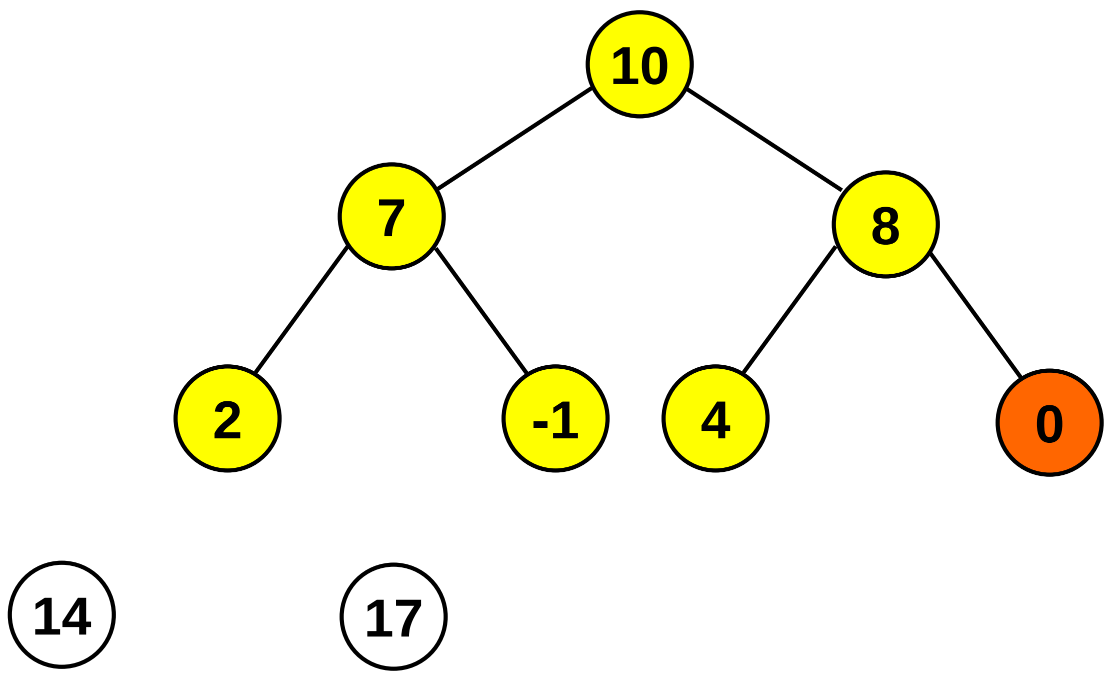
1. **Trocamos o ultimo elemento com a raiz, e extraímos esse elemento do amontoado.**
2. Aplicamos o `fixDown` à raiz
3. Voltamos a 1.

Exercício

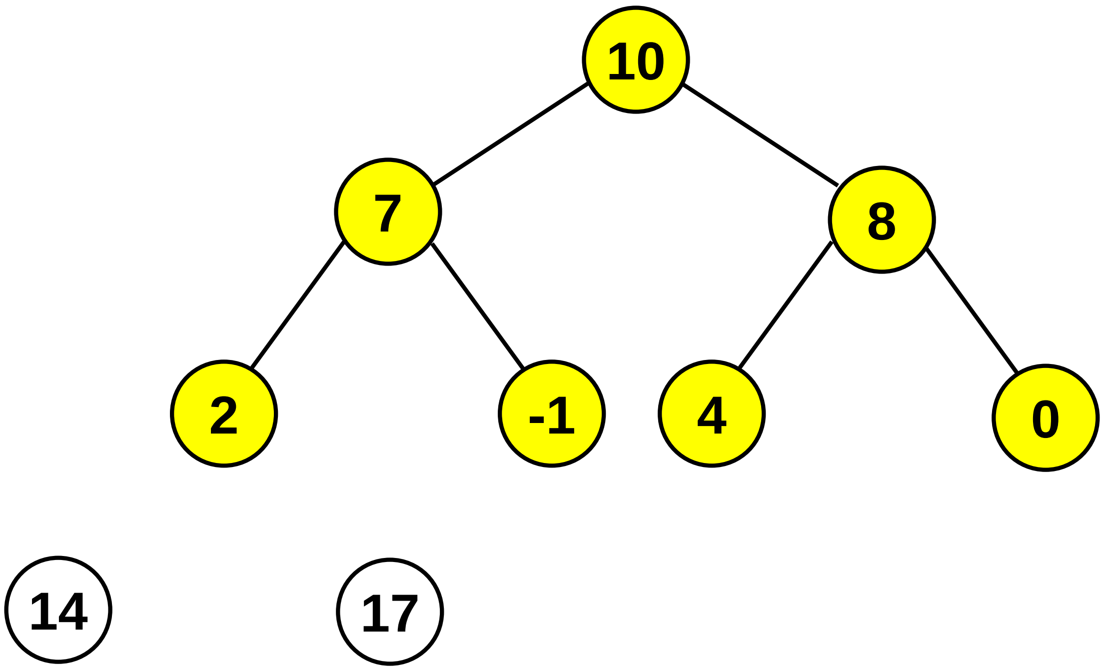


1. Trocamos o ultimo elemento com a raiz, e extraímos esse elemento do amontoado.
2. **Aplicamos o `fixDown` à raiz**
3. Voltamos a 1.

Exercício



Exercício




```
a = { 10, 7, 8, 2, -1, 4, 0, 14, 17 }
```

Heapsort - Complexidade

```
void buildheap(Item a[], int l, int r){
    int k, heapsize = r-l+1;
    for (k = heapsize/2-1; k >= l; k--)
        fixDown(a, l, r, l+k);
}
```

- Construção do amontoado (ciclo `for`):
  - $O(N \lg N)$ no pior caso
  - Pode ser provado $O(N)$

Porquê ?

- Colocação das chaves (ciclo `while`):
  - $O(N \lg N)$ no pior caso
  - Pode ser provado que para elementos distintos, o melhor caso também é $\Omega(N \lg N)$

Porquê ?

```
void heapsort(Item a[], int l, int
{
   buildheap(a,l,r);
   while (r-l > 0) {
      exch(a[l], a[r]);
      fixDown(a, l, --r, l);
   }
}
```

- Complexidade no pior caso é $O(N \lg N)$

- Não é estável

Ordenação por Comparação

- Algoritmos de ordenação baseados em comparações são pelo menos $\Omega(N \lg N)$
  - Para N chaves existem $N!$ ordenações possíveis das chaves
  - Algoritmo de ordenação por comparação utiliza comparações de pares de chaves para selecionar uma das $N!$ ordenações
    - Escolher uma folha em árvore com $N!$ folhas
    - Altura da árvore é não inferior $\lg(N!) \approx N \lg N$
- **É possível obter algoritmos mais eficientes *desde que não sejam apenas baseados em comparações***
- ***Alternativa***: Utilizar informação quanto às chaves utilizadas
  - Counting Sort
  - Radix Sort