Aula 18
Filas Prioritárias / Amontoados / Heapsort

Algoritmos e Estruturas de Dados

Filas Prioritárias

e Amontoados (Heaps)

Filas Prioritárias

- Estrutura de dados semelhante a uma fila
  - Inserimos elementos na fila
  - Quando removemos o próximo, removemos o mais prioritário

    O menor

    O maior

  - Abordagens simples não são eficientes

Array	Inserção	Remoção
array não ordenado	O(1)	O(n)
array ordenado	O(n)	O(1)

Filas Prioritárias

- Estrutura de dados semelhante a uma fila
  - Inserimos elementos na fila
  - Quando removemos o próximo, removemos o mais prioritário

    O menor

    O maior

- Muito usadas em Informática
  - E.g.

    Inteligência Artificial

    Procura A usa uma fila prioritária para organizar os nós abertos*

Filas Prioritárias

Implementadas através de Amontoados (*Heaps*)

Amontoado (Heap)

- *Binary Heap* (Amontoado binário)
  - Coleção informada otimizada para aceder de cada vez:

    Ao maior elemento (max-heap), ou

    Ao menor elemento (min-heap)

Definição:
Um amontoado (*heap*) é uma árvore binária com uma restrição de ordem adicional.

Para um max-heap:
Seja **n.v** o valor guardado em cada nó **n**, e **n'** qualquer nó descendente de **n** (ou seja pertecente à subárvore esquerda ou subárvore direita).

Então $\forall n \forall n' \; n.v \geq n'.v$

Traduzido por miúdos: O valor guardado num nó pai é sempre maior ou igual ao valor dos seus filhos.

Amontoado (*Heap*)

- *Binary Heap* (Amontoado binário)
  - Coleção informada otimizada para aceder de cada vez:

    Ao maior elemento (max-heap), ou

    Ao menor elemento (min-heap)

Definição:
Um amontoado (*heap*) é uma árvore binária com uma restrição de ordem adicional.

Para um min-heap:
Seja **n.v** o valor guardado em cada nó **n**, e **n'** qualquer nó descendente de **n** (ou seja pertecente à subárvore esquerda ou subárvore direita).

Então $\forall n \forall n' \; n.v \leq n'.v$

Traduzido por miúdos: O valor guardado num nó pai é sempre menor ou igual ao valor dos seus filhos.

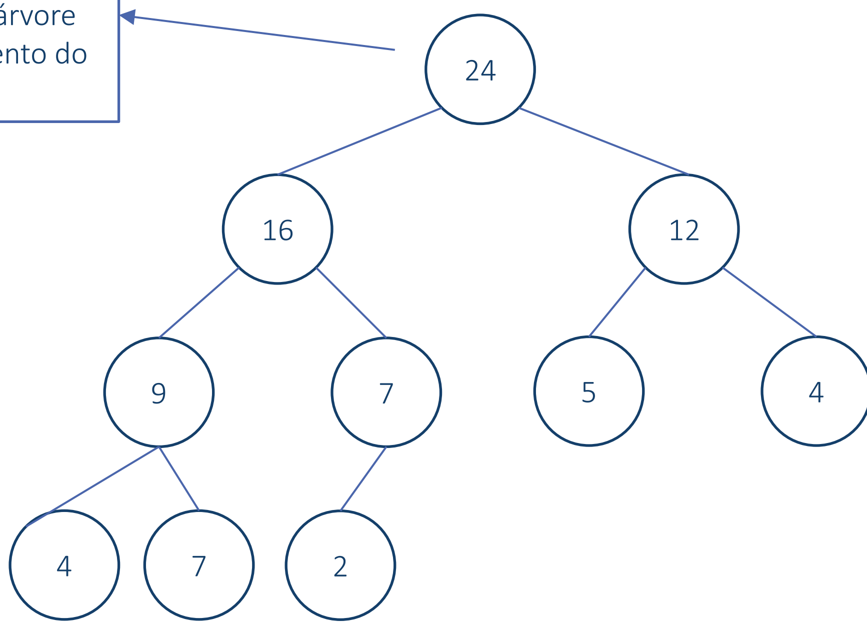
UAlg FCT
UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Amontoado (Heap)

- ***Def:*** *um amontoado é uma árvore binária onde cada nó é (maior ou igual)/(menor ou igual) que os seus dois filhos*

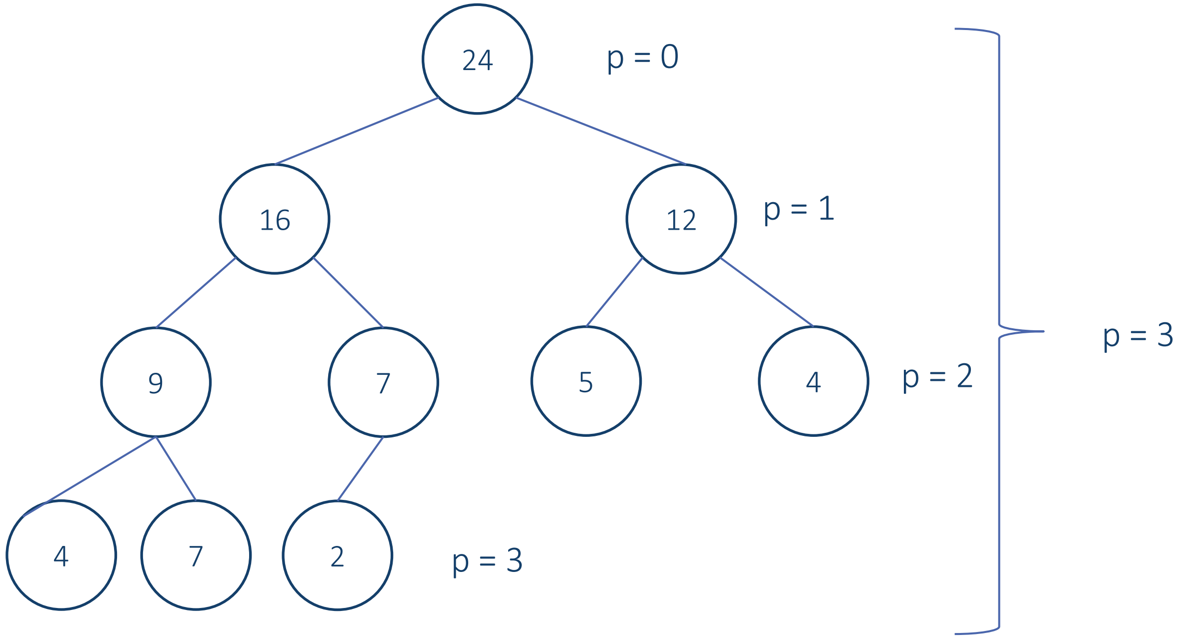
Observação: Este é um *max-heap*, pois cada nó é maior ou igual que os seus filhos. A raiz desta árvore será sempre o maior elemento do heap.

Um min-heap é usado quando queremos uma fila prioritária organizada do menor para o maior



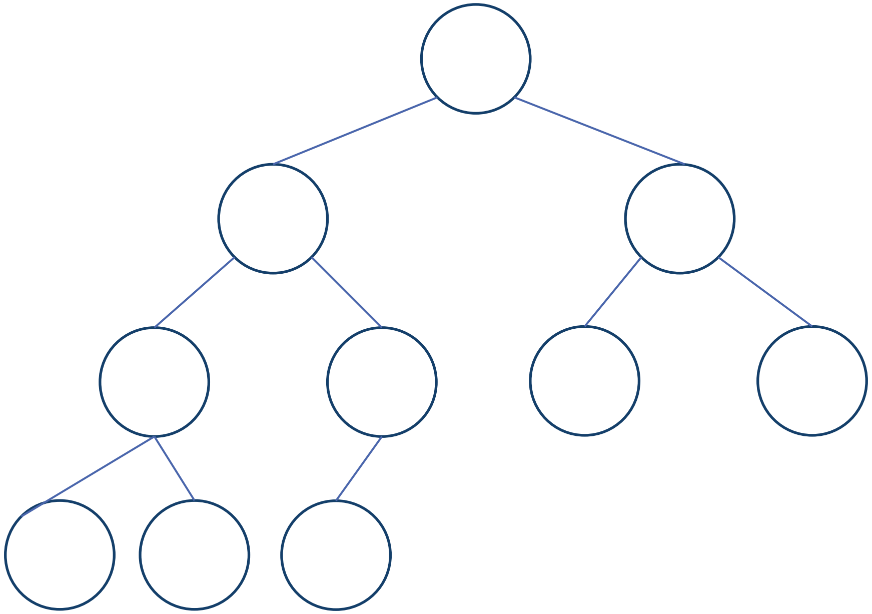
Propriedades Amontoado

- O maior elemento de um *max-heap* é a sua raiz
- A profundidade máxima de uma heap com n elementos é
  - $\lfloor \log_2 n \rfloor$



Representação de Amontoados

- Embora um amontoado possa ser representado através de nós ligados, existem 2 propriedades importantes:
  - Um amontoado é uma árvore binária
  - **Um amontoado é uma árvore balanceada**



Def: uma árvore é balanceada se para qualquer nó, a diferença entre a altura das subárvores ≤ 1

Representação de Amontoados

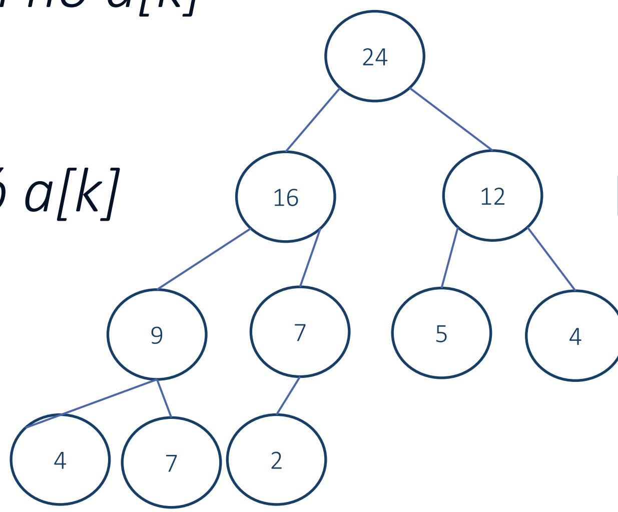
- Uma árvore binária balanceada pode ser representada usando um array

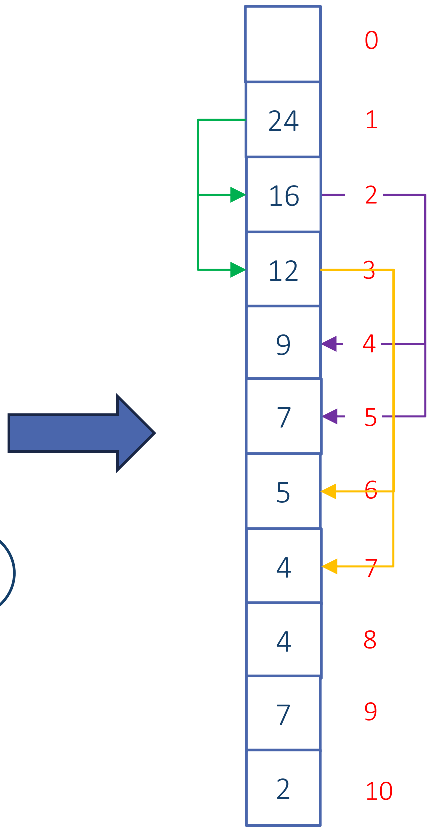
- É uma representação bastante eficiente

- Vamos usá-la para a implementação de um amontoado

Amontoado como *array*

- Para facilitar, vamos assumir que só usamos o array a partir do índice 1
- a[1] = raiz *heap*
- *Para obter os filhos de um nó a[k]*
  - *a[2k], a[2k+1]*
- *Para obter o pai de um nó a[k]*
  - *a[k/2]*





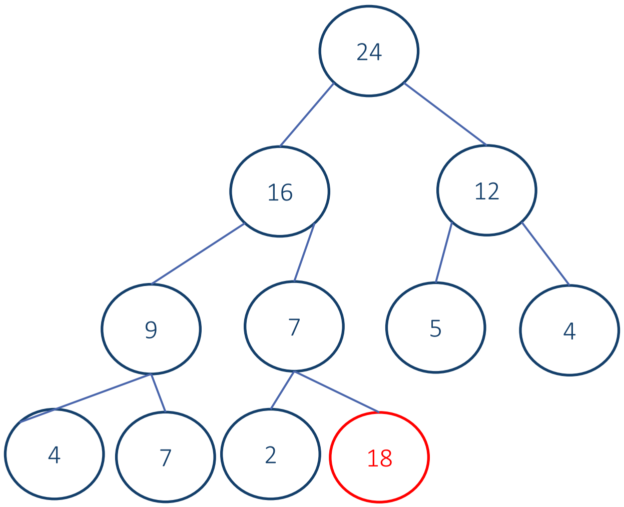
Heapify baixo para cima

- Quando um elemento é adicionado no fim da heap
- Temos que garantir as propriedades da heap, avançando de baixo para cima
- *Heapify* baixo para cima (ou *swim* up)
  - Se o filho for maior que o pai

    Trocar pai com filho

    Avançar para cima e repetir
  - Se o filho não for maior

    Podemos parar



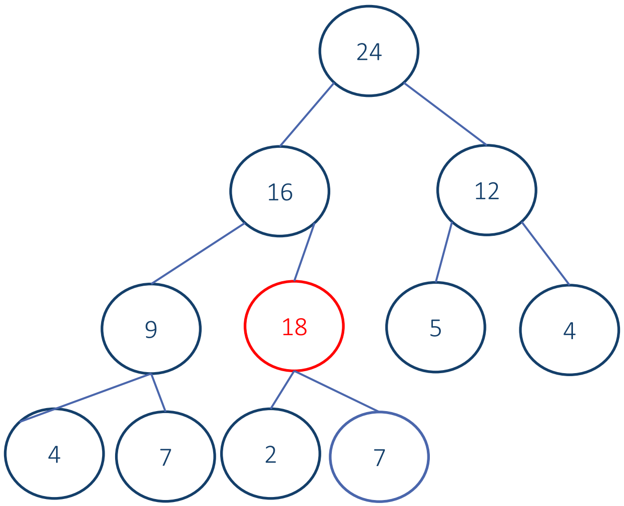
Heapify baixo para cima

- *Heapify* baixo para cima (ou *swim* up)
  - Se o filho for maior que o pai

    Trocar pai com filho

    Avançar para cima e repetir
  - Se o filho não for maior

    Podemos parar



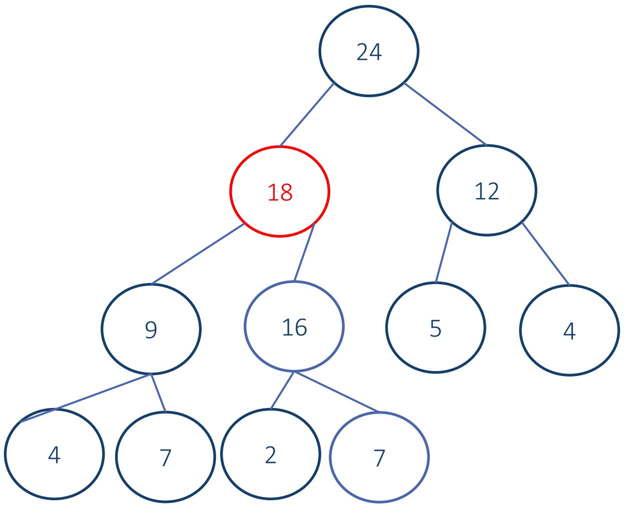
Heapify baixo para cima

- *Heapify* baixo para cima (ou *swim* up)
  - Se o filho for maior que o pai

    Trocar pai com filho

    Avançar para cima e repetir
  - Se o filho não for maior

    Podemos parar



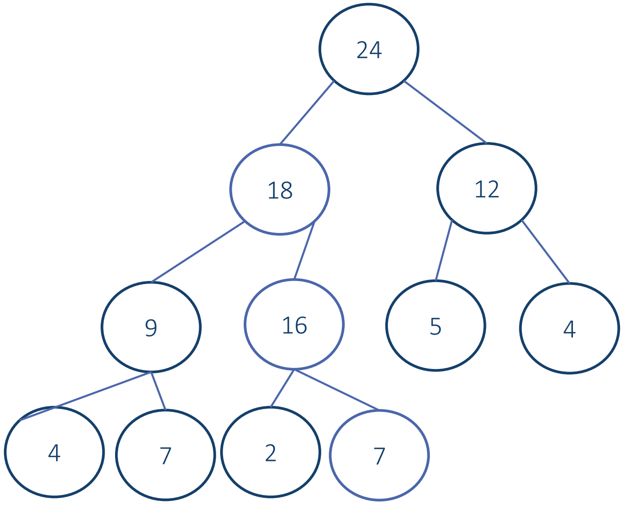
Heapify baixo para cima

- *Heapify* baixo para cima (ou *swim* up)
  - Se o filho for maior que o pai

    Trocar pai com filho

    Avançar para cima e repetir
  - Se o filho não for maior

    Podemos parar



Heapify baixo para cima

- *Heapify* baixo para cima (ou *swim* up)
  - Se o filho for maior que o pai

    Trocar pai com filho

    Avançar para cima e repetir
  - Se o filho não for maior

    Podemos parar

```
private void heapifyBottomUp(T[] a, int k)
{
    while(k > 1 && less(a[k/2],a[k]))
    {
        exchange(a, k/2, k);
        k = k/2;
    }
}
```

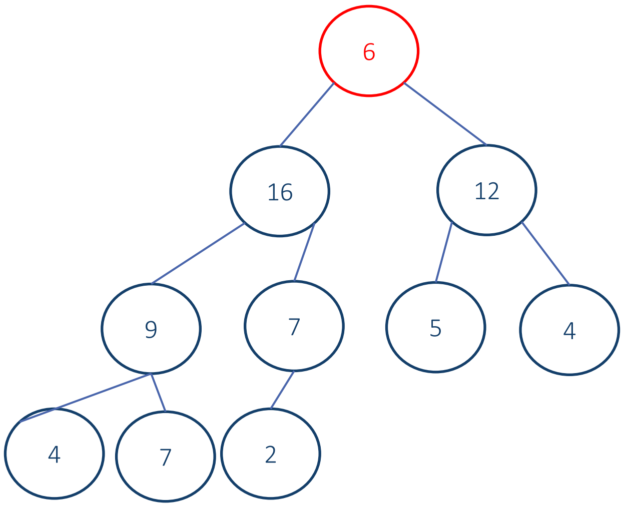
Heapify cima para baixo

- Quando um elemento é adicionado no início da heap
- Temos que garantir as propriedades da heap, avançando de cima para baixo
- *Heapify* cima para baixo (ou *sink*)
  - Se o pai for menor que o maior dos filhos

    Trocar pai com maior filho

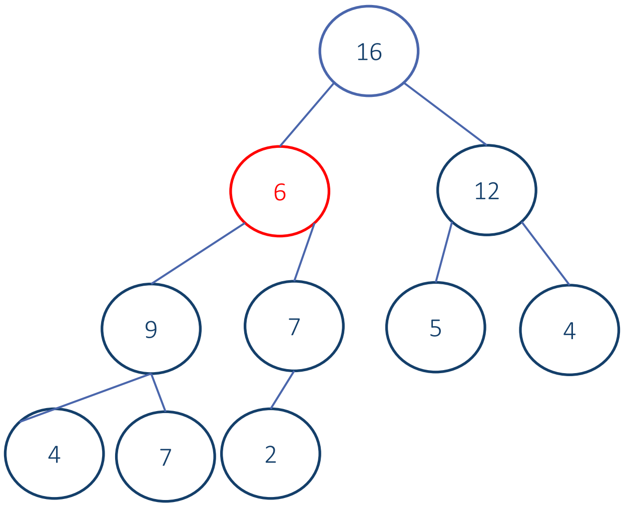
    Avançar para baixo e repetir
  - Caso contrário

    Podemos parar



Heapify cima para baixo

- *Heapify* cima para baixo (ou *sink*)
  - Se o pai for menor que o maior dos filhos
    - *Trocar pai com maior filho*
    - *Avançar para baixo e repetir*
  - Caso contrário
    - *Podemos parar*



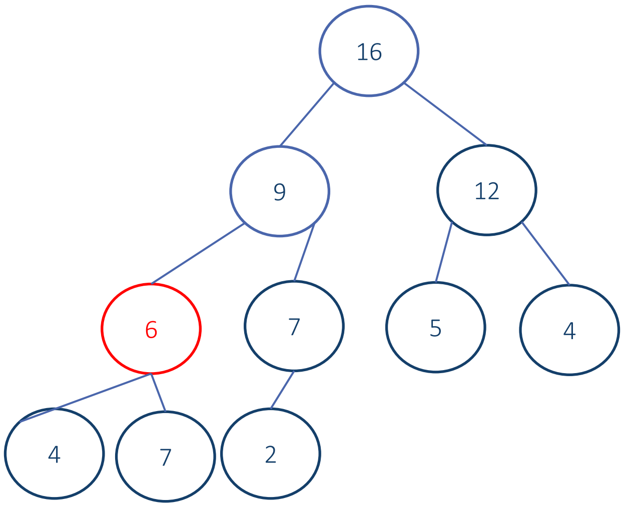
Heapify cima para baixo

- *Heapify* cima para baixo (ou *sink*)
  - Se o pai for menor que o maior dos filhos

    Trocar pai com maior filho

    Avançar para baixo e repetir
  - Caso contrário

    Podemos parar



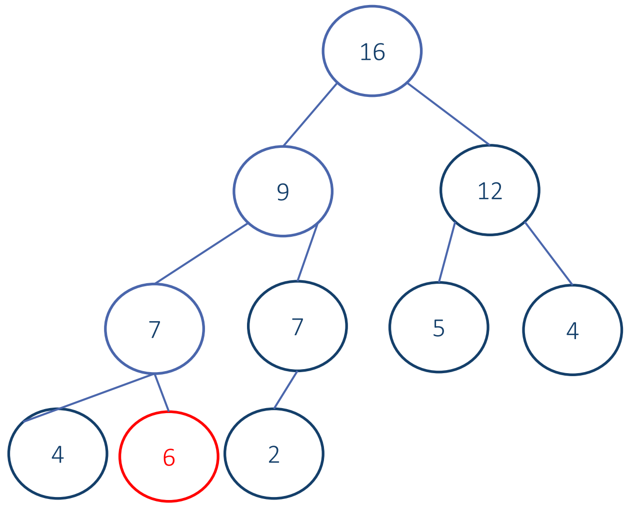
Heapify cima para baixo

- *Heapify* cima para baixo (ou *sink*)
  - Se o pai for menor que o maior dos filhos

    Trocar pai com maior filho

    Avançar para baixo e repetir
  - Caso contrário

    Podemos parar



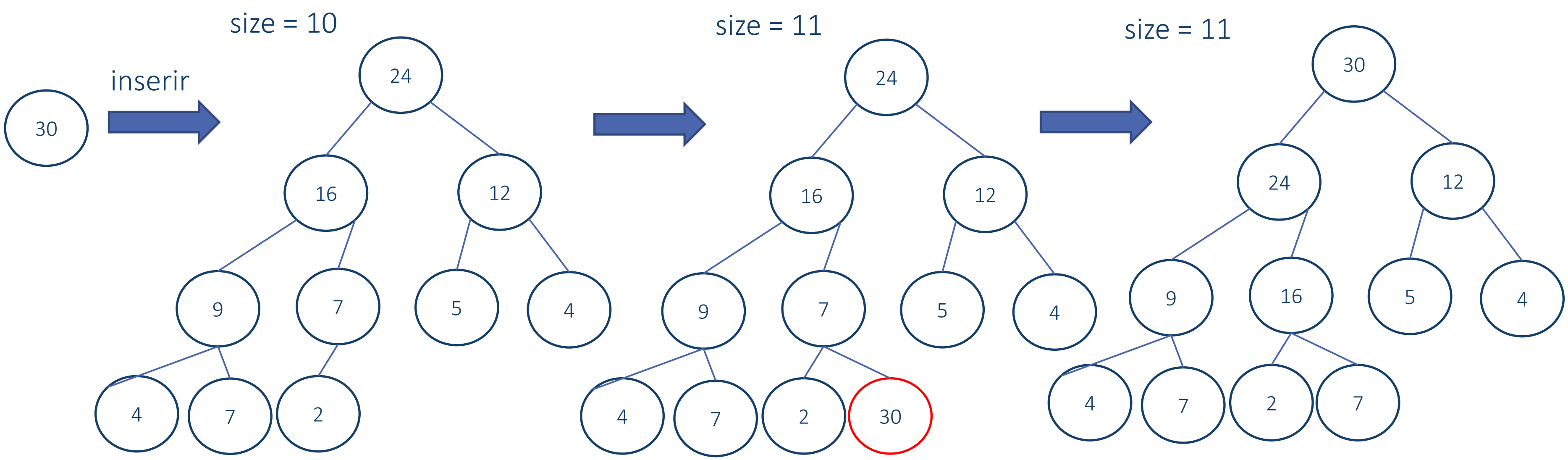
Heapify cima para baixo

- *Heapify* cima para baixo (ou *sink*)
  - Se o pai for menor que o maior dos filhos
    - *Trocar pai com maior filho*
    - *Avançar para baixo e repetir*
  - Caso contrário
    - *Podemos parar*

```
private static <T extends Comparable<T>>
  void heapifyTopDown(T[] a, int k, int n)
//n - last valid heap index
{
    int child = k*2;
    while(child <= n)
    {
        //select the biggest child
        if(child < n && less(a[child],a[child+1])) child++;
        //if the father is not smaller
        // we can stop here
        if(!less(a[k], a[child])) break;
        exchange(a, k, child);
        k = child;
        child = 2*k;
    }
}
```

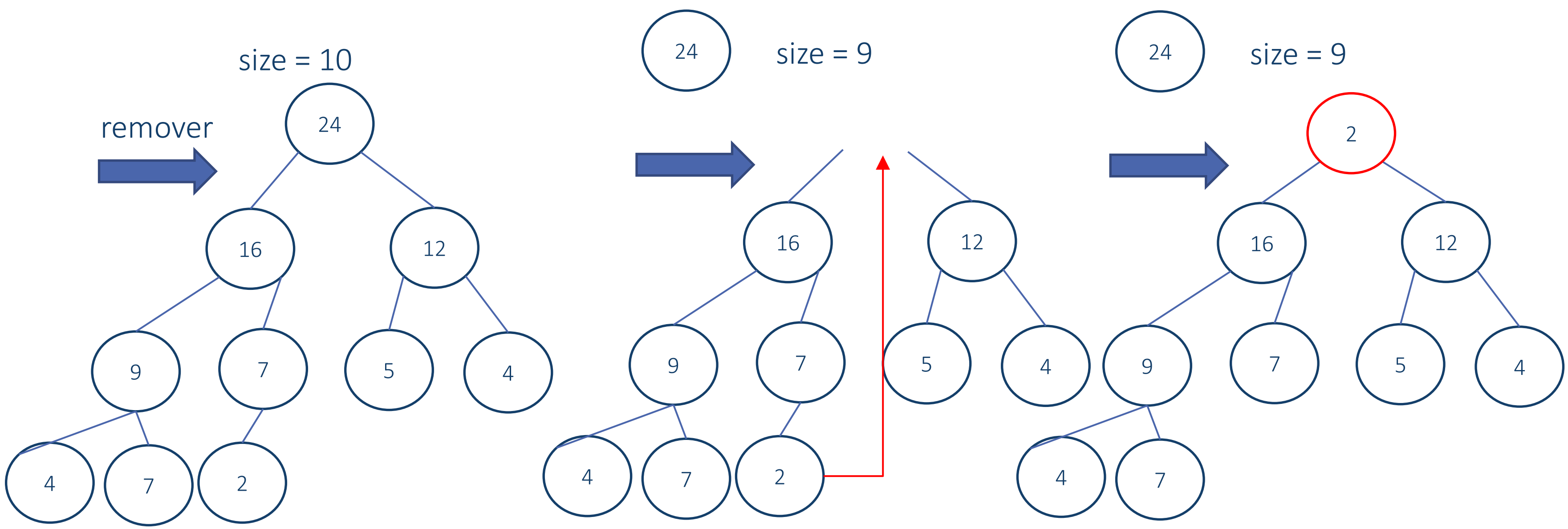
Fila Prioritária: Inserir elemento

- Inserir elemento no fim do *array*
- Aumentar o tamanho do *heap*
- *Heapify* baixo para cima a partir do elemento adicionado



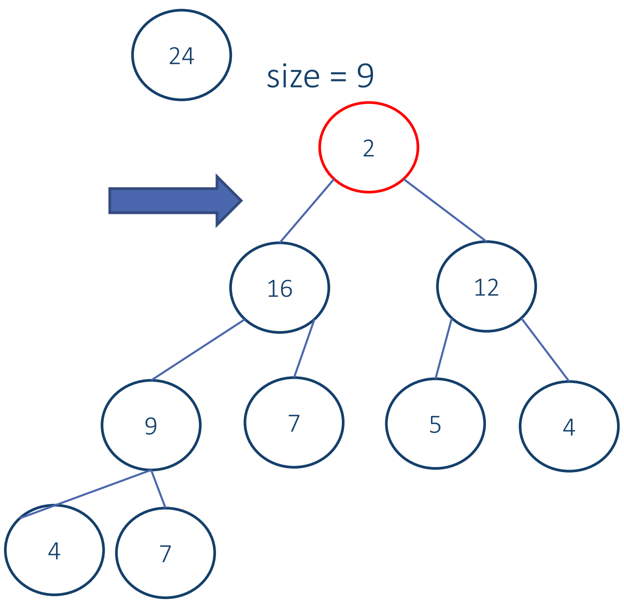
Fila Prioritária : Remover elemento

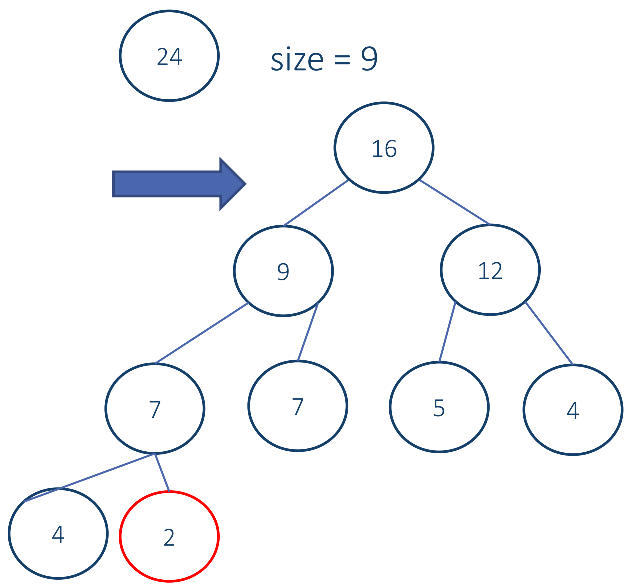
- Remover elemento da raiz do *heap*
- *Colocar o ultimo elemento da heap na raiz*
- *Diminuir o tamanho do heap*
- *Heapify cima para baixo a partir da nova raiz*



Fila Prioritária : Remover elemento

- Remover elemento da raiz do *heap*
- *Colocar o ultimo elemento da heap na raiz*
- *Diminuir o tamanho do heap*
- *Heapify cima para baixo a partir da nova raiz*





Complexidade Temporal

- Inserir
  - comparações

    $\sim log_2\, n$
  - trocas

    $\sim log_2\, n$

```
private static <T extends Comparable<T>>
void heapifyBottomUp(T[] a, int k)
{
    while(k > 1 && less(a[k/2],a[k]))
    {
        exchange(a, k/2, k);
        k = k/2;
    }
}
```

Complexidade Temporal

- Remover
  - comparações

    $\sim 2 \log_2 n$
  - trocas

    $\sim \log_2 n$

```java
private static <T extends Comparable<T>> void
heapifyTopDown(T[] a, int k, int n)
//n - last valid heap index
{
    int child = k*2;
    while(child <= n)
    {
        //select the biggest child
        if(child < n && less(a[child],a[child+1])) child++;
        //if the father is not smaller
        // we can stop here
        if(!less(k, child)) break;
        exchange(a, k, child);
        k = child;
        child = 2*k;
    }
}
```

Heapsort

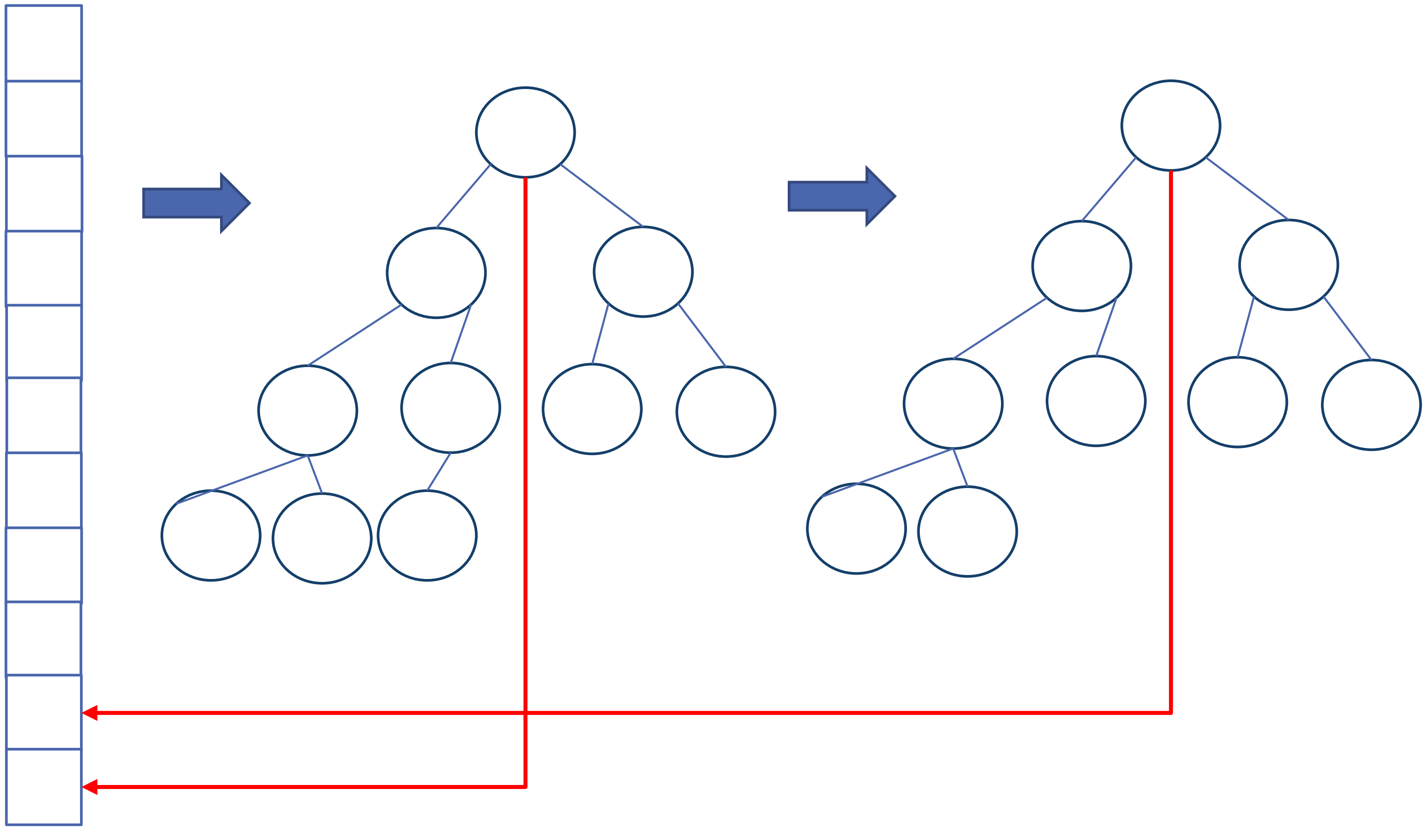
Heapsort

- Ideia:
  - tirar partido de uma *heap*
  - *para implementar um algoritmo de ordenação*

  - *1) colocar os items a ordenar numa heap*
  - *2) remover o máximo da heap (um de cada vez) e colocar no fim do array*

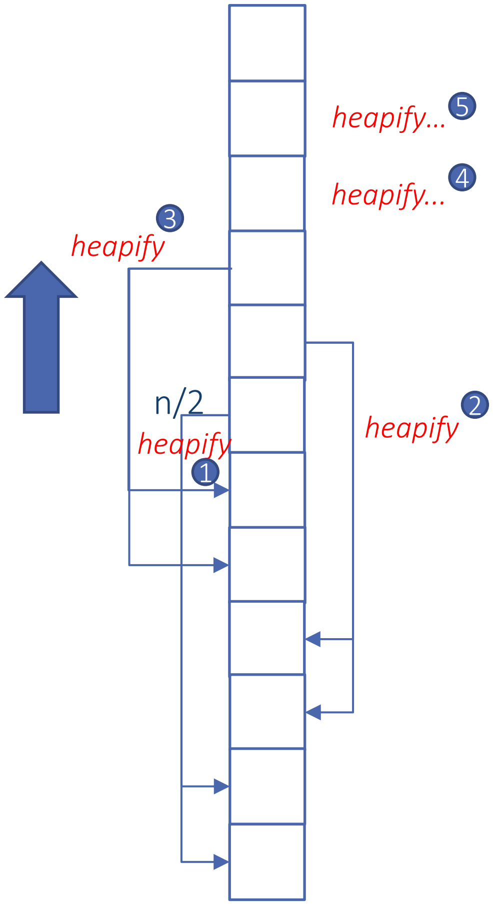
Heapsort

- *1) colocar os items a ordenar numa heap*
- *2) remover o máximo da heap (um de cada vez) e colocar no fim do array*



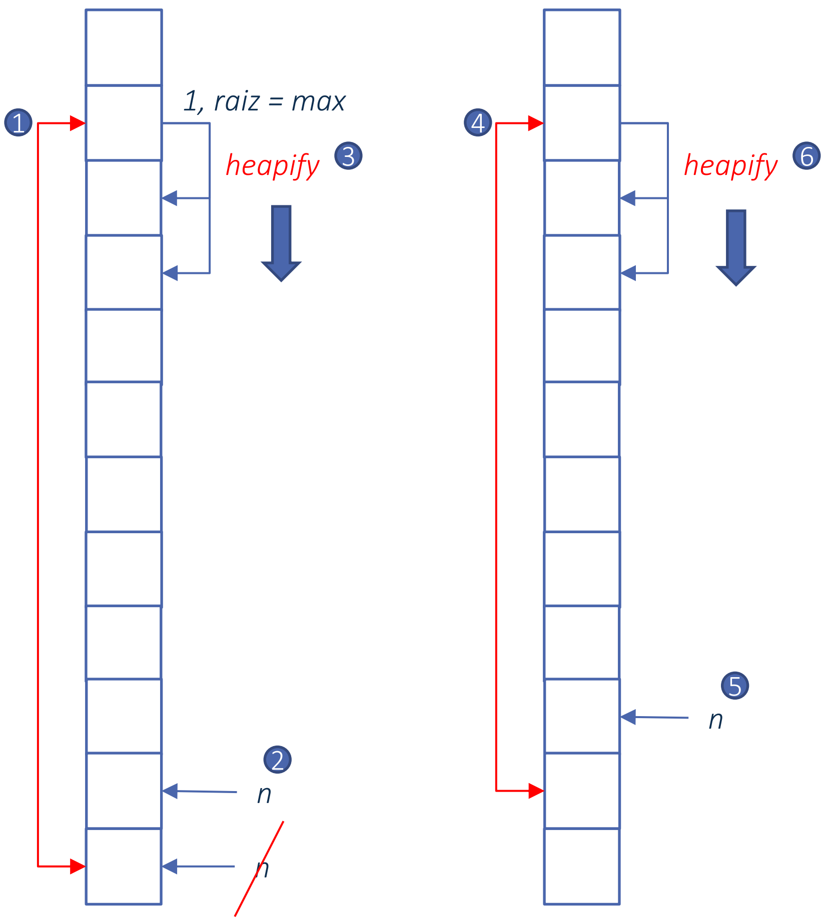
Heapsort no local

- Na realidade não precisamos de um array adicional
- Podemos garantir as propriedades de um heap no array original
  - Ao avançarmos de n/2 para cima
  - Fazendo heapify top-down a cada elemento



Heapsort no local

- Para reconstruir array ordenado
- Trocar raiz heap com o fim do array
- Diminuir tamanho heap
- Heapify Top-down



Heapsort

```java
public static <T extends Comparable<T>> void sort(T[] a)
{
    //does not sort position 0 of array
    int n = a.length;
    //enforces heap properties in the array
    for(int k = n/2; k >= 1; k--)
    {
        heapifyTopDown(a,k,n-1);
    }

    //reorganizes the array,
    //removing the maximum from the heap
    //and putting it at the end of the array
    while(n > 1)
    {
        exchange(a, 1, --n);
        heapifyTopDown(a, 1, n);
    }
}
```

problema: não ordena a posição 0 do array

1) organiza o array de acordo com um heap

2) usa o heap para ordenar o array

Heapsort com índice 0

- Equivalente ao anterior
  - Mas sempre que acedemos ao array
  - Acedemos à posição anterior

```
public static <T extends Comparable<T>> void sort(T[] a)
{
    int n = a.length;

    //enforces heap properties in the array
    for(int k = n/2; k >= 1; k--)
    {
        heapifyTopDown(a,k,n);
    }

    //reorganizes the array,
    while(n > 1)
    {
        exchange(a, 0, --n);
        heapifyTopDown(a, 1, n);
    }
}
```

Heapsort com índice 0

- Equivalente ao anterior
  - Mas sempre que acedemos ao array
  - Acedemos à posição anterior

```
private static <T extends Comparable<T>> void heapifyTopDown(T[] a, int k, int n)
{
    int child = k*2;
    while(child <= n)
    {
        //select the biggest child (if available)
        if(child < n && less(a[child-1],a[child])) child++;
        //if the father is not smaller than the biggest child
        // we can stop here
        if(!less(a[k-1], a[child-1])) break;
        exchange(a, k-1, child-1);
        k = child;
        child = 2*k;
    }
}
```

Complexidade Temporal

```
public static <T extends Comparable<T>> void sort(T[] a)
{
    int n = a.length;

    //enforces heap properties in the array
    for(int k = n/2; k >= 1; k--)
    {
        heapifyTopDown(a,k,n);
    }

    //reorganizes the array,
    while(n > 1)
    {
        exchange(a, 0, --n);
        heapifyTopDown(a, 1, n);
    }
}
```

$1\ T_{heapifyTD}(n) + 2\ T_{heapifyTD}(n/2) + 4\ T_{heapifyTD}(n/4) + \ldots$
$= 2 \log_2 n + 4 \log_2 n/2 + 8 \log_2 n/4 + \ldots$
$= 2 (\log_2 n + 2 \log_2 n/2 + 4 \log_2 n/4 + \ldots)$
$= 2\ n$

Complexidade Temporal

```java
public static <T extends Comparable<T>> void sort(T[] a)
{
    int n = a.length;

    //enforces heap properties in the array
    for(int k = n/2; k >= 1; k--)
    {
        heapifyTopDown(a,k,n);
    }

    //reorganizes the array,
    while(n > 1)
    {
        exchange(a, 0, --n);
        heapifyTopDown(a, 1, n);
    }
}
```

$1\ T_{heapifyTD}(n) + 2\ T_{heapifyTD}(n/2) + 4\ T_{heapifyTD}(n/4) + \ldots$
$= 2 \log_2 n + 4 \log_2 n/2 + 8 \log_2 n/4 + \ldots$
$= 2 (\log_2 n + 2 \log_2 n/2 + 4 \log_2 n/4 + \ldots)$
$= 2\ n$

~n

~2 $\log_2 n$

$T(n) \sim 2n + 2n \log_2 n$
$\sim 2n \log_2 n$

Complexidade Temporal

```
public static <T extends Comparable<T>> void sort(T[] a)
{
    int n = a.length;

    //enforces heap properties in the array
    for(int k = n/2; k >= 1; k--)
    {
        heapifyTopDown(a,k,n);
    }

    //reorganizes the array,
    while(n > 1)
    {
        exchange(a, 0, --n);
        heapifyTopDown(a, 1, n);
    }
}
```

No melhor caso, saímos sempre ao fim de 2 comparações:
1) Irmão c/ irmão
2) Pai c/ filho

Melhor caso: $\underbrace{2 + 2 + 2 + ... + 2}_{n/2} = 2*n/2 = n$

~n

Melhor caso: 2 n

Heapsort	Melhor caso	Pior caso	Aleatório	O
less/compare	*~3 n*	*~2 n* log_2 *n*	*~2 n* log_2 *n*	O(*n* log_2 *n*)

Sumário

Algoritmos ordenação

Propriedades de algoritmos de ordenação

- Inplace
  - algoritmo de ordenação que não necessita de muita memória extra para ordenar um array

    Pode gastar um valor constante c de memória extra

    Ou gastar uma quantidade de memória < n: ex: ~log n

- Stable
  - Algoritmo de ordenação que preserva a ordem relativa de elementos iguais

Sumário Algoritmos Ordenação

	inplace?	stable?	best	average	worst	remarks
selection	✔		$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	$\frac{1}{2} n^2$	n exchanges
insertion	✔	✔	n	$\frac{1}{4} n^2$	$\frac{1}{2} n^2$	use for small n or partially ordered
shell	✔		$n \log_3 n$	?	$c\, n^{3/2}$	tight code; subquadratic
merge		✔	$\frac{1}{2} n \lg n$	$n \lg n$	$n \lg n$	$n \log n$ guarantee; stable
quick	✔		$n \lg n$	$2 n \ln n$	$\frac{1}{2} n^2$	$n \log n$ probabilistic guarantee; fastest in practice
3-way quick	✔		n	$2 n \ln n$	$\frac{1}{2} n^2$	improves quicksort when duplicate keys
heap	✔		$3 n$	$2 n \lg n$	$2 n \lg n$	$n \log n$ guarantee; in-place