

# Árvores Binárias IV

15/11/2023

# Ficheiro ZIP

- Está disponível no Moodle um **ficheiro ZIP** de suporte aos tópicos de hoje
- O tipo abstrato **MAX-Heap binária**, que permite instanciar filas com prioridade (**Priority-Queues**)

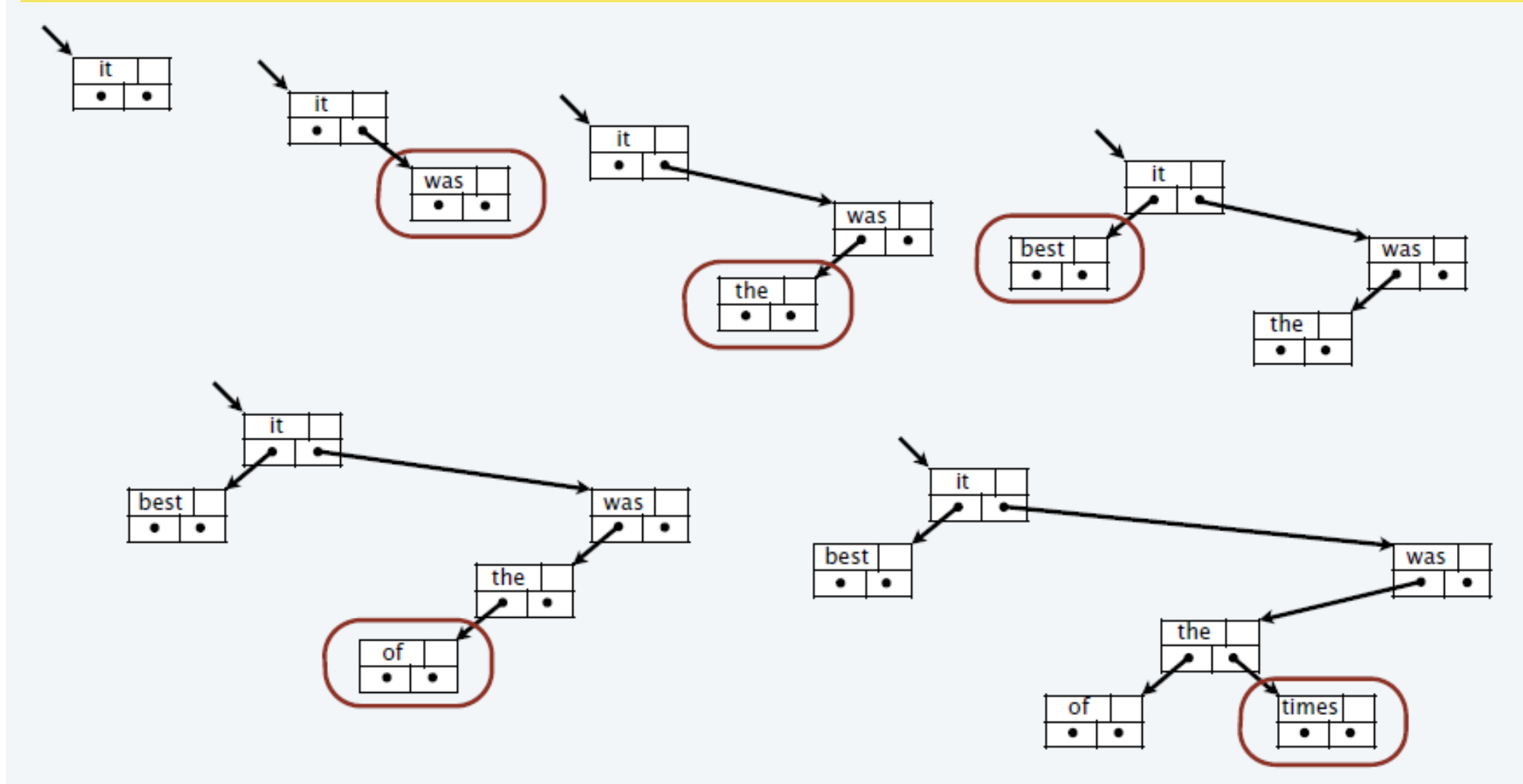
# Sumário

- Recap
- Filas com prioridade – “Priority Queues”
- Binary Heaps – “Amontoados Binários”
- O TAD **MIN-Heap**
- O TAD **MAX-Heap**
- O algoritmo **Heap-Sort**

Let's  
RECAP

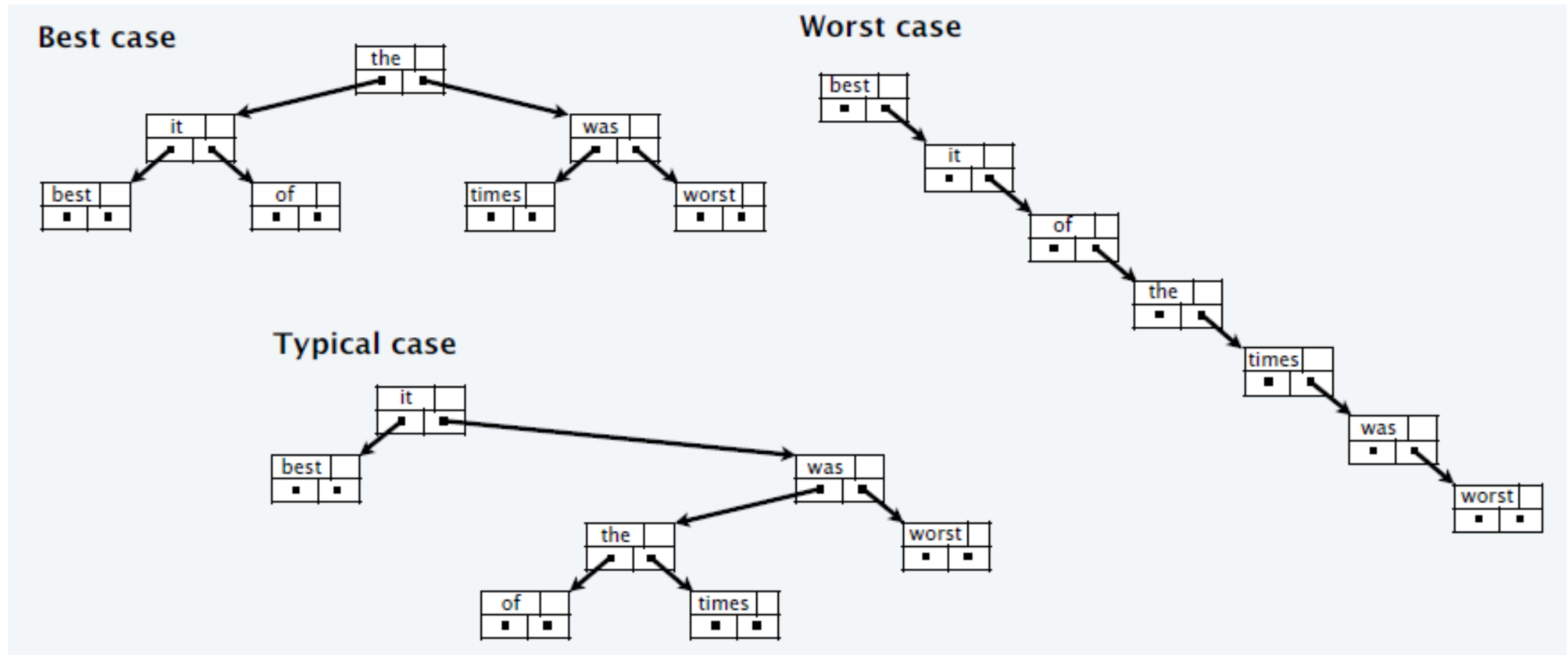
# Recapitulação

# ABPs – Adicionar como **folha**, manter **ordem**



[Sedgewick & Wayne]

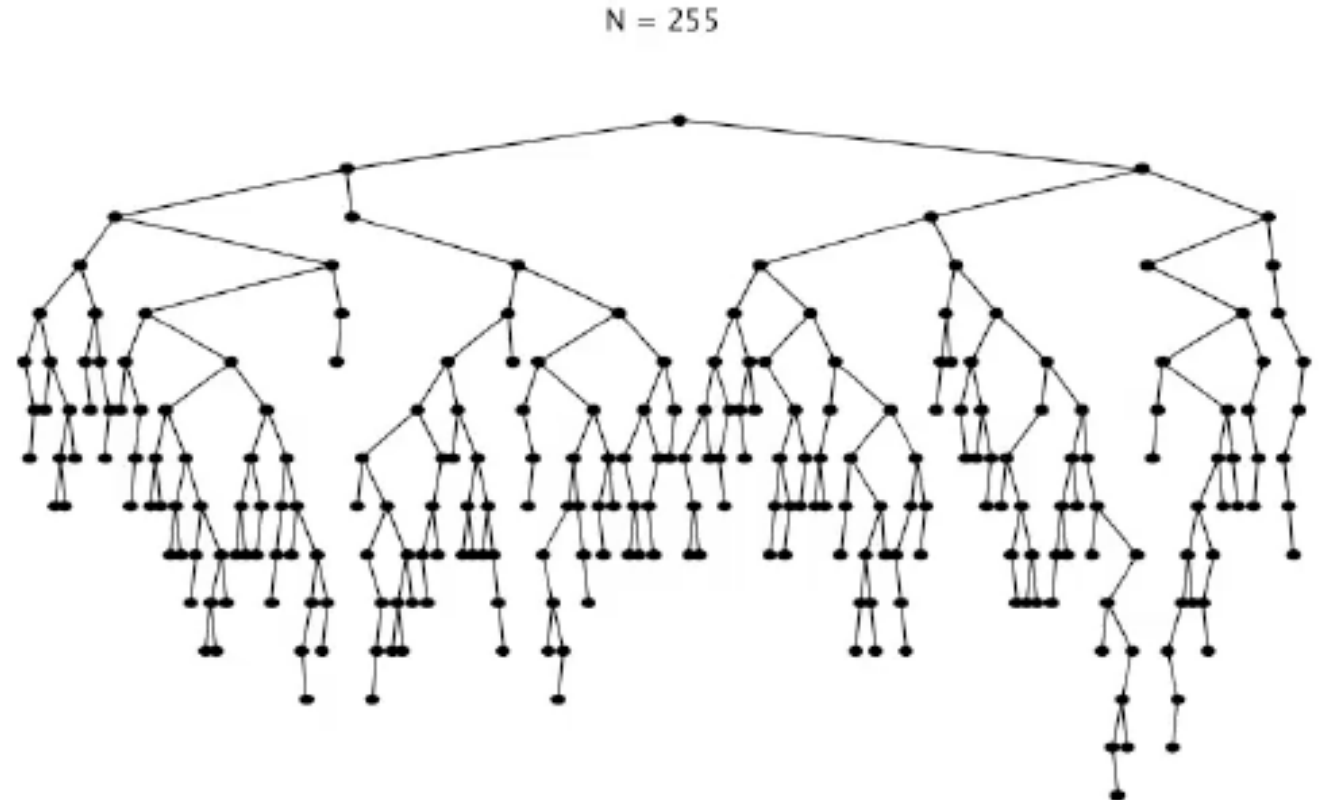
# Altura – Diferentes sequências de inserção



[Sedgewick & Wayne]

# Altura – Adição numa ordem aleatória

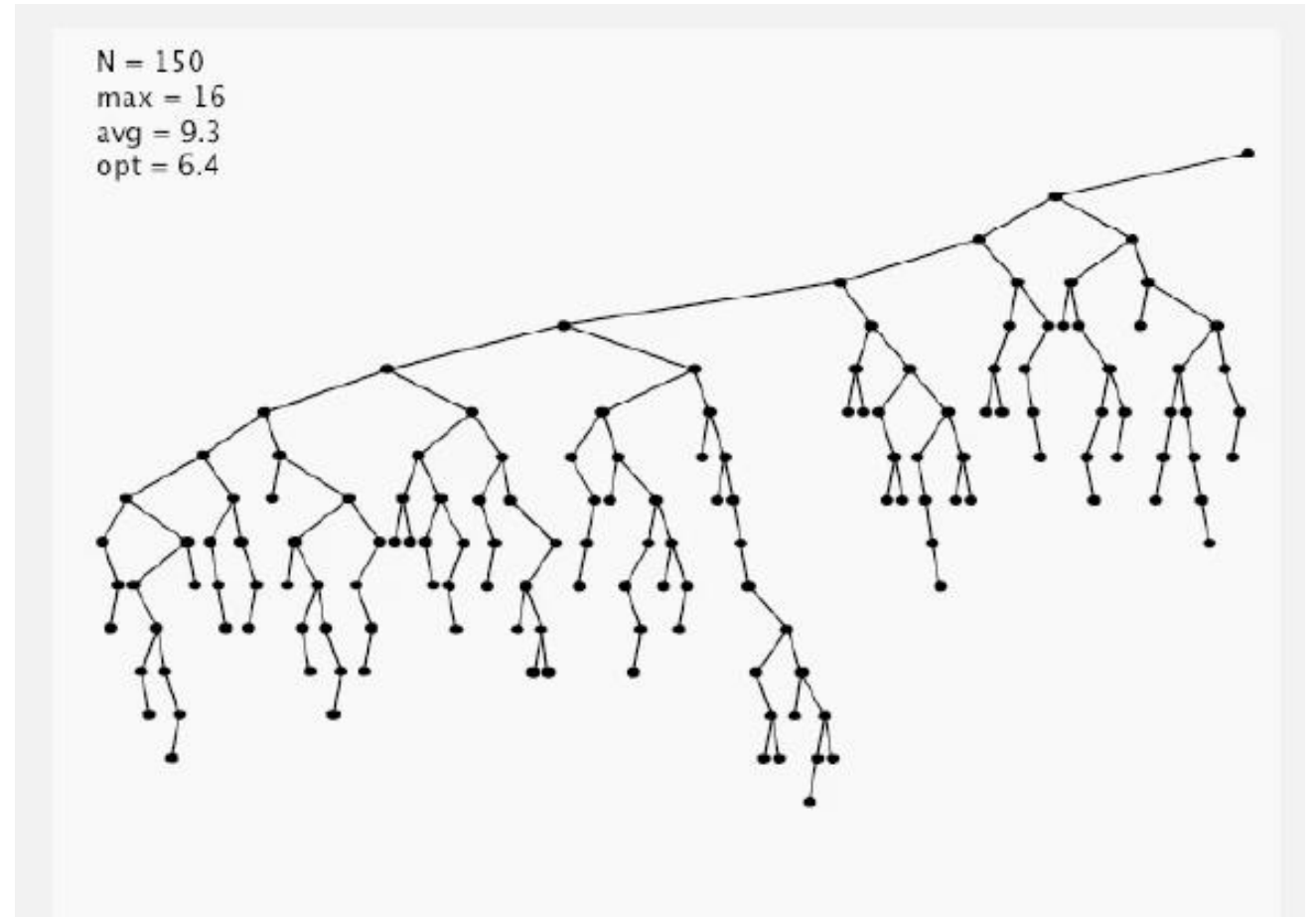
- Árvore **aprox.** equilibrada



[Sedgewick & Wayne]

# Após muitos apagamentos

- Árvore perde alguma “simetria” !!
- Consequências ?

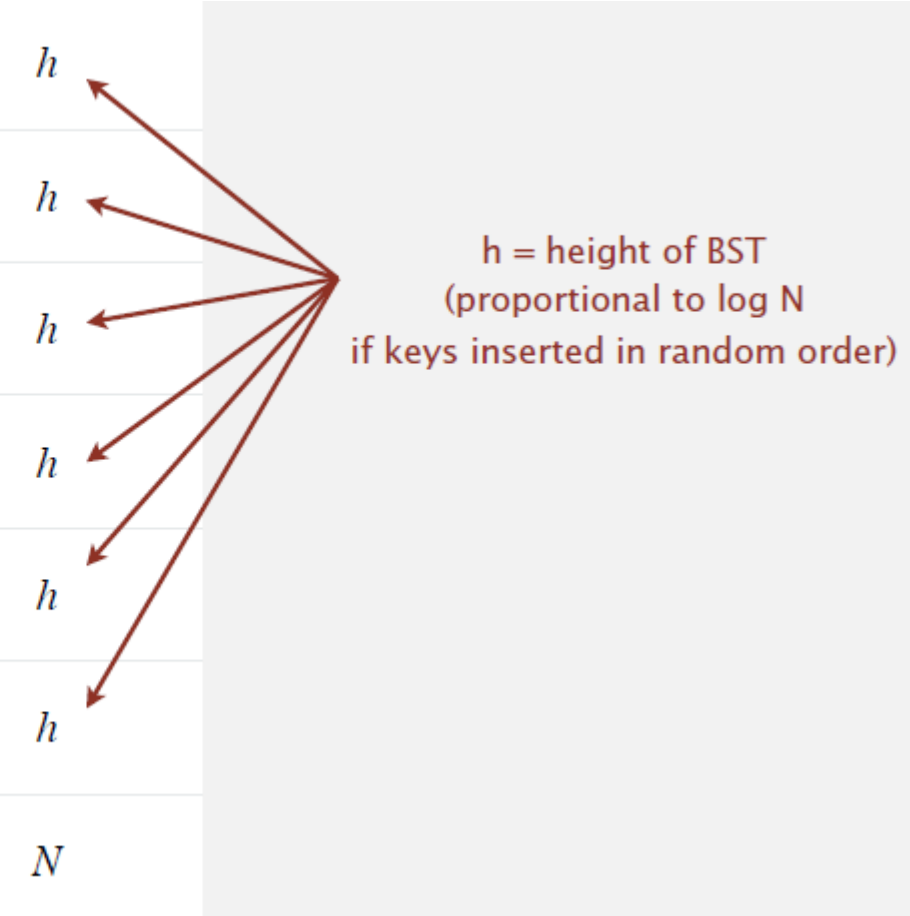


[Sedgewick & Wayne]



# Eficiência - Lista ligada / Array ordenado / ABP

search	$N$	$\lg N$	$h$
insert	$N$	$N$	$h$
min / max	$N$	1	$h$
floor / ceiling	$N$	$\lg N$	$h$
rank	$N$	$\lg N$	$h$
select	$N$	1	$h$
ordered iteration	$N \log N$	$N$	$N$



$h$  = height of BST  
(proportional to  $\log N$   
if keys inserted in random order)

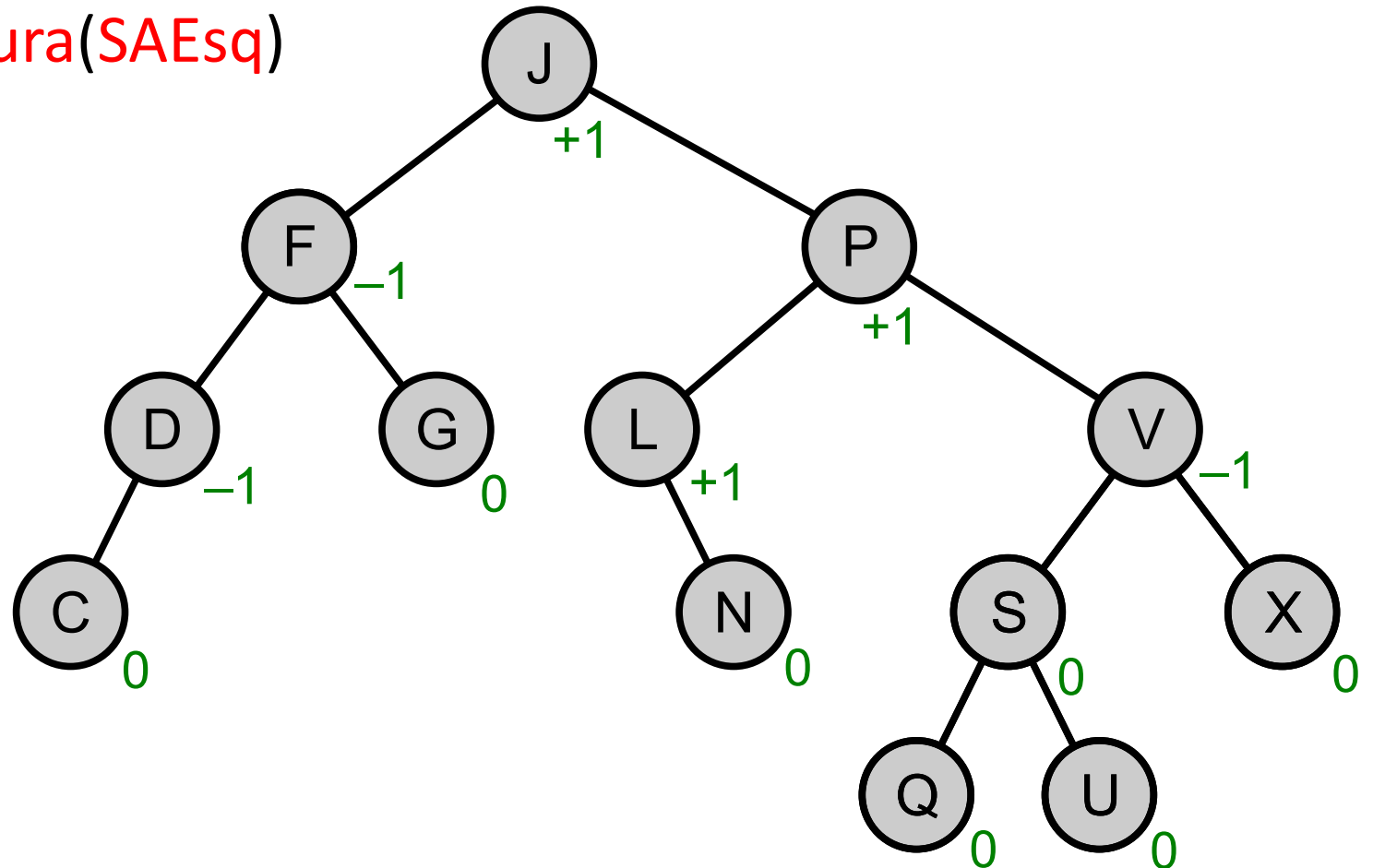
[Sedgewick & Wayne]

# Árvores equilibradas em altura

- **Esforço** computacional das operações habituais sobre ABPs depende do **comprimento do caminho** a partir da raiz da árvore
- **Evitar** que uma ABP tenha uma **altura “exagerada”**, para assegurar um bom “comportamento” – **Altura  $\in O(\log n)$**
- **O que fazer ?**
- Assegurar que, para cada nó, a **altura** das suas duas **subárvores** não é “muito diferente” – **Critério de equilíbrio**

# Fator de equilíbrio de um nó

- $F = \text{altura}(\text{SADir}) - \text{altura}(\text{SAEsq})$

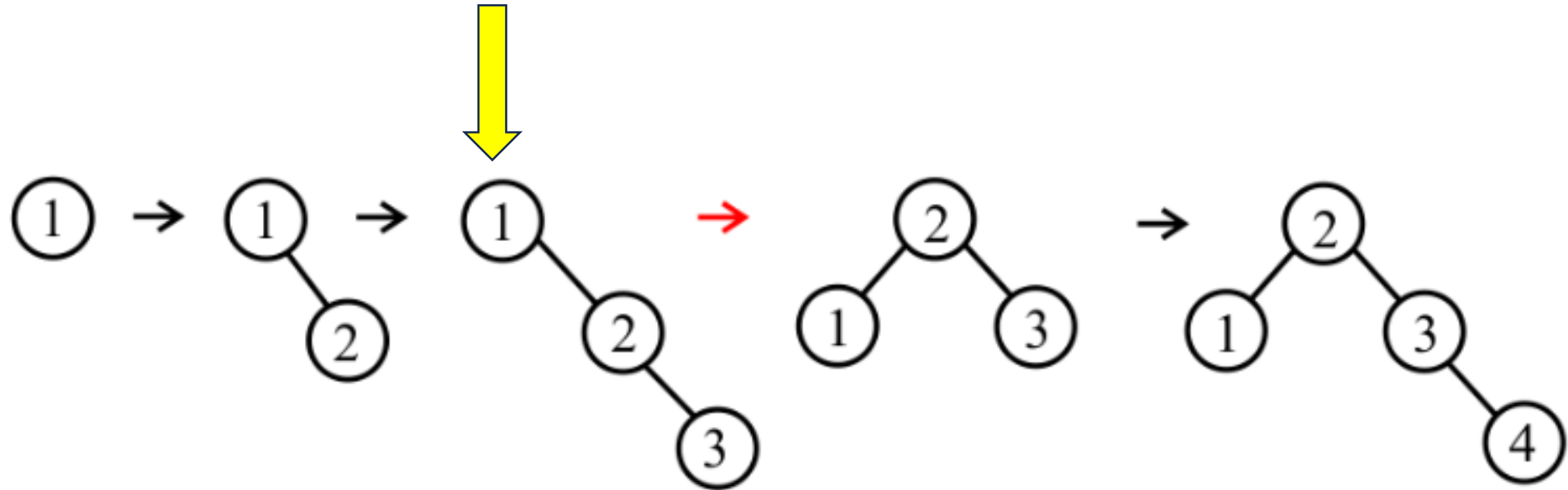


[Wikipedia]

# Quando fazer ? / Como fazer ?

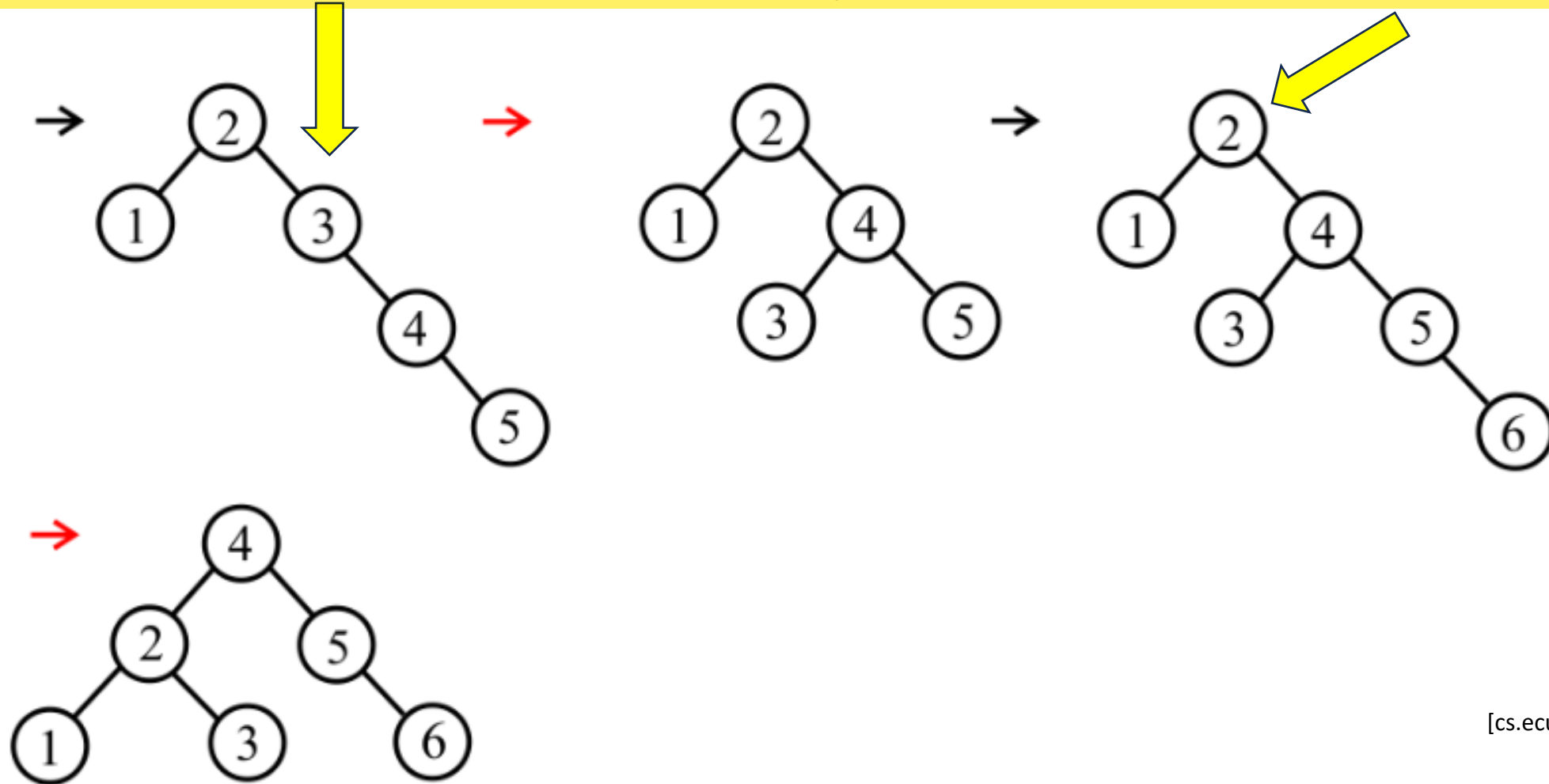
- **Assegurar o critério de equilíbrio** sempre que se adiciona ou remove um nó :  $F = -1, 0, +1$
- Reposicionar nós / subárvores quando **falha** :  $F = -2, +2$
- MAS, **manter o critério de ordem** da ABP !!
- 4 tipos de **operações de rotação**
- Apenas **trocas de ponteiros**
- Basta fazer a verificação / rotações ao longo do **caminho** entre a raiz e o nó – **traceback**

# Árvore AVL – Inserir + Equilibrar, se necessário



[cs.ecu.edu]

# Árvore AVL – Inserir + Equilibrar, se necessário



[cs.ecu.edu]

# Filas com Prioridade

## - Priority Queues

# Fila com prioridade

- Coleções

- Inserir e apagar elementos; que elemento apagar ?

- **STACK** : Apagar o **último** elemento inserido (**pop**)

- **QUEUE** : Apagar o **primeiro** elemento (**dequeue**)

- **PRIORITY QUEUE** : Apagar o elemento de **maior** (ou **menor**) **prioridade**

- Items devem ser **comparáveis** !!

<i>operation</i>	<i>argument</i>	<i>return value</i>
<i>insert</i>	P	
<i>insert</i>	Q	
<i>insert</i>	E	
<i>remove max</i>		Q
<i>insert</i>	X	
<i>insert</i>	A	
<i>insert</i>	M	
<i>remove max</i>		X
<i>insert</i>	P	
<i>insert</i>	L	
<i>insert</i>	E	
<i>remove max</i>		P

[Sedgewick & Wayne]



# Array não-ordenado vs array ordenado


operation	argument	return value	size	contents (unordered)					contents (ordered)								
insert	P		1	P					P								
insert	Q		2	P	Q				P	Q							
insert	E		3	P	Q	E			E	P	Q						
remove max		Q	2	P	E				E	P							
insert	X		3	P	E	X			E	P	X						
insert	A		4	P	E	X	A		A	E	P	X					
insert	M		5	P	E	X	A	M	A	E	M	P	X				
remove max		X	4	P	E	M	A		A	E	M	P					
insert	P		5	P	E	M	A	P	A	E	M	P	P				
insert	L		6	P	E	M	A	P	L	A	E	L	M	P	P		
insert	E		7	P	E	M	A	P	L	E	A	E	E	L	M	P	P
remove max		P	6	E	M	A	P	L	E	A	E	E	L	M	P		

A sequence of operations on a priority queue

# Eficiência computacional

implementation	insert	del max	max
unordered array	1	$n$	$n$
ordered array	$n$	1	1
goal	$\log n$	$\log n$	$\log n$

order of growth of running time for priority queue with  $n$  items



[Sedgewick & Wayne]

# Binary Heaps

## - Filas com Prioridade

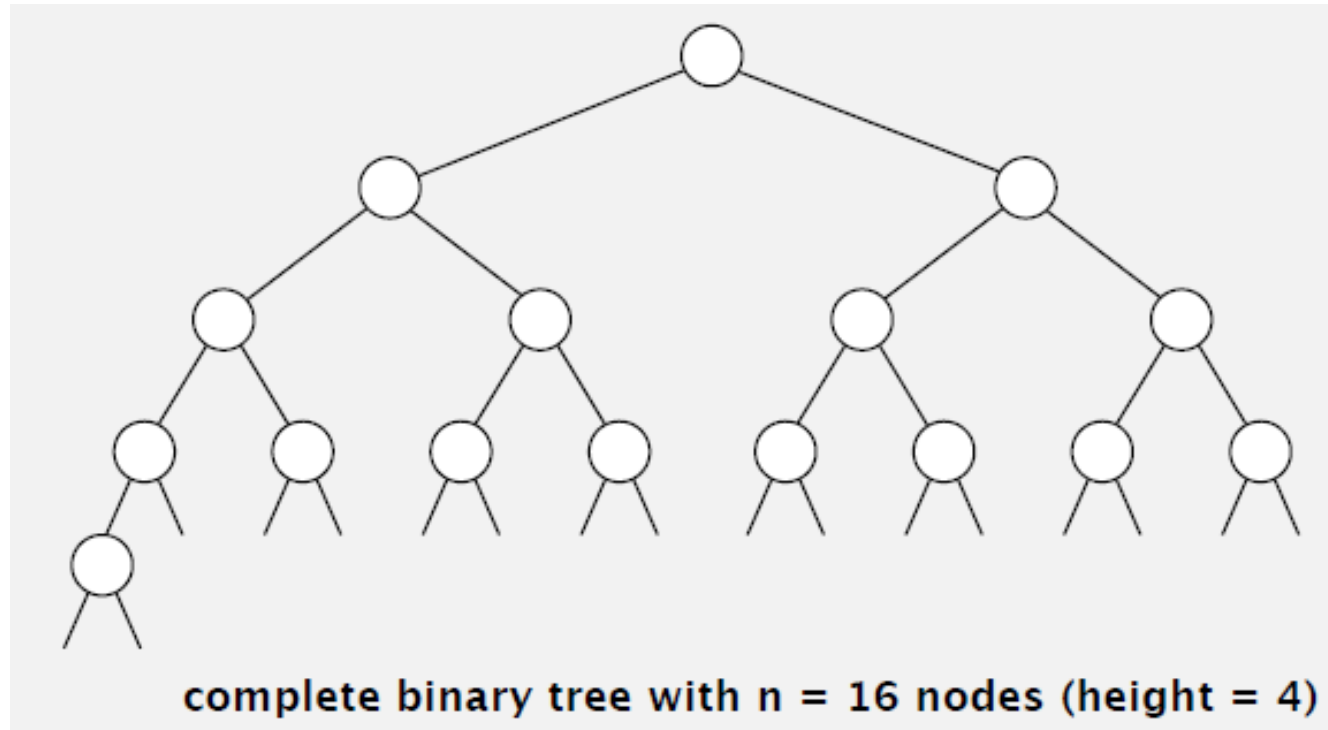
# Binary Heaps – “Amontoados Binários”

- Representar **filas com prioridade** usando **árvores binárias completas**
- Com um **critério de ordem/prioridade**
- **Elementos** da heap habitualmente **armazenados por níveis**, num **array**
- Acesso aos **filhos** e ao **pai de um nó** através de **índices**
- **Não** são utilizados **ponteiros !!**
- **Eficiência !!**

# Binary Heaps – Operações habituais

- Adicionar um elemento
- Consultar o elemento de maior/menor prioridade
- Apagar o elemento de maior/menor prioridade
- ...
- Não há acesso aleatório !!

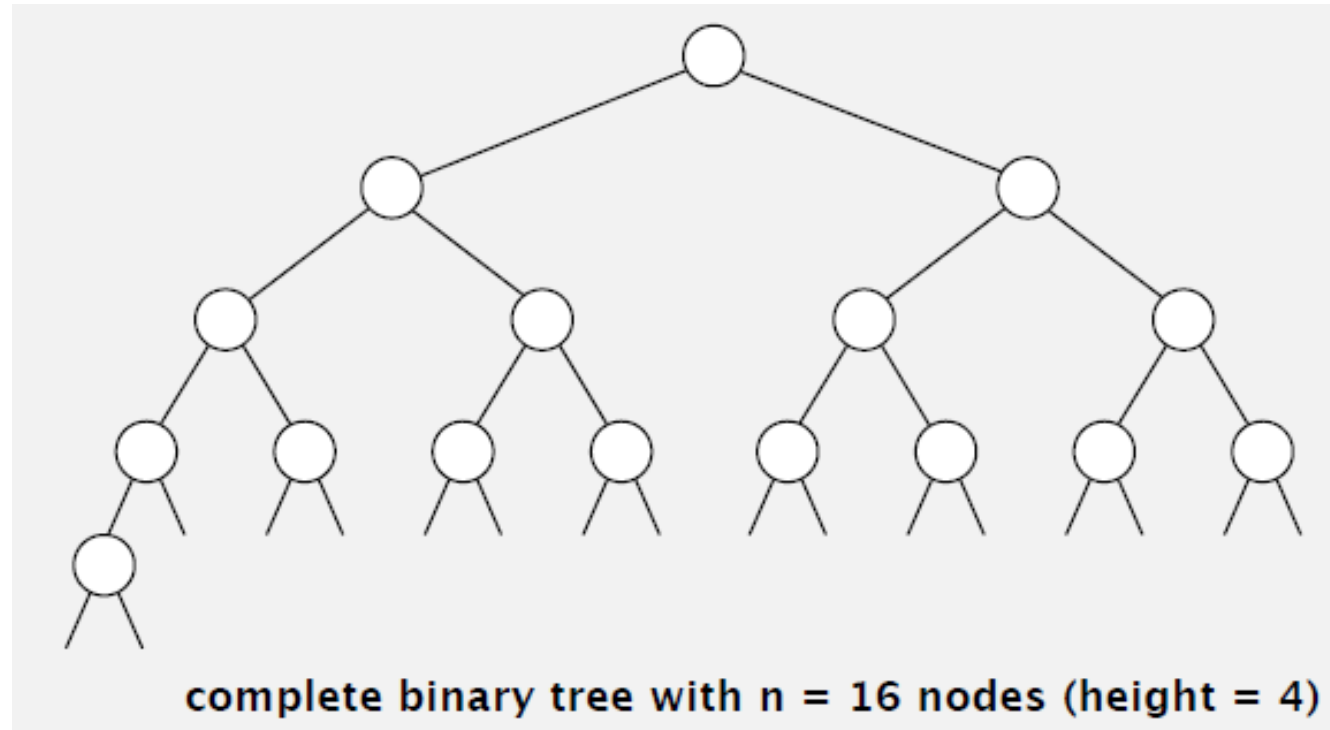
# Árvore binária completa



[Sedgewick & Wayne]

- Árvore **perfeitamente equilibrada**, com a possível exceção do **último nível**, que tem os nós (folhas) “ancorados à esquerda”

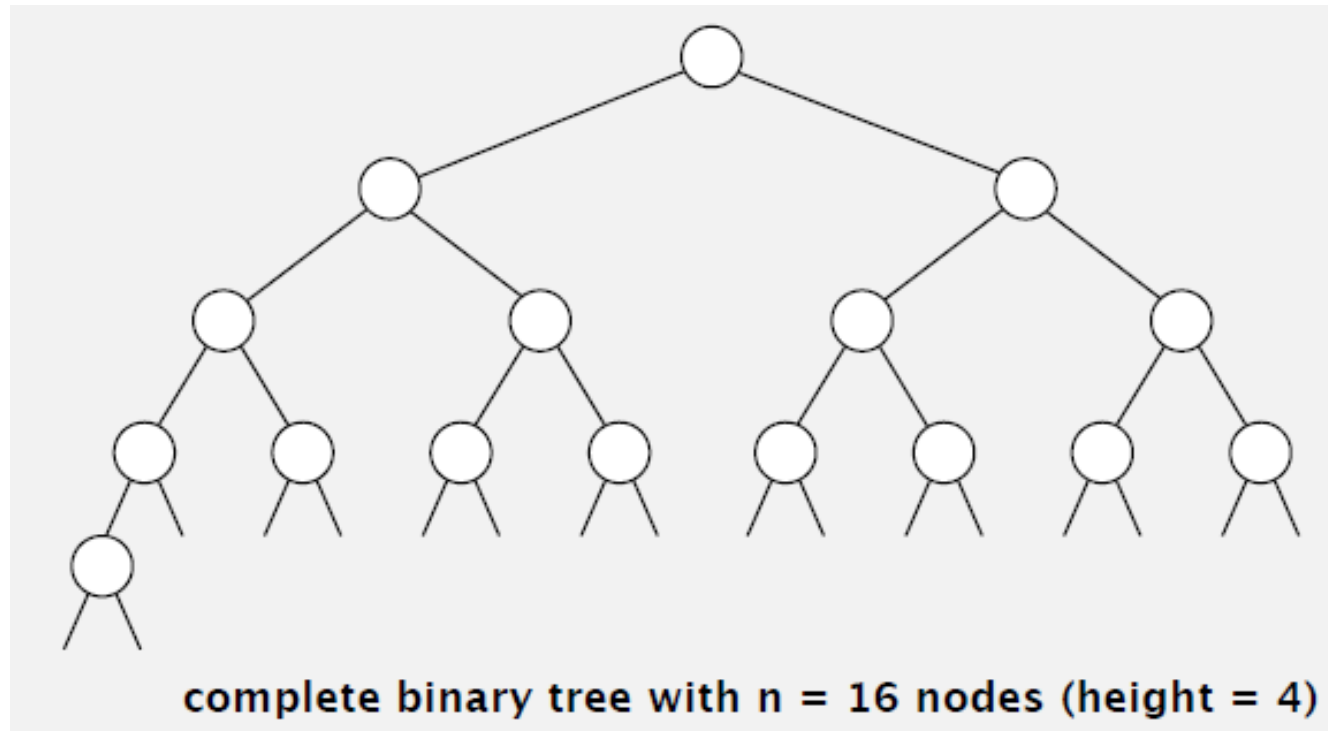
# Árvore binária completa



[Sedgewick & Wayne]

- A altura de uma árvore completa com  $n$  nós é  $\text{floor}(\log_2 n)$ 
  - A altura aumenta apenas quando  $n = 2^k$

# Árvore binária completa



[Sedgewick & Wayne]

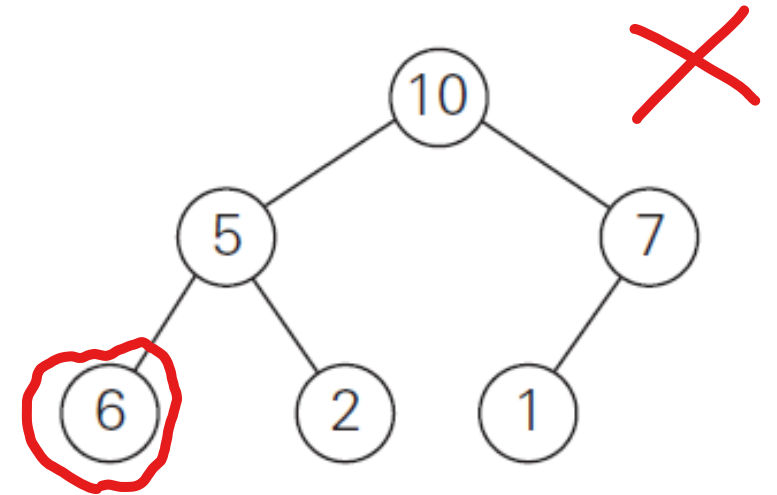
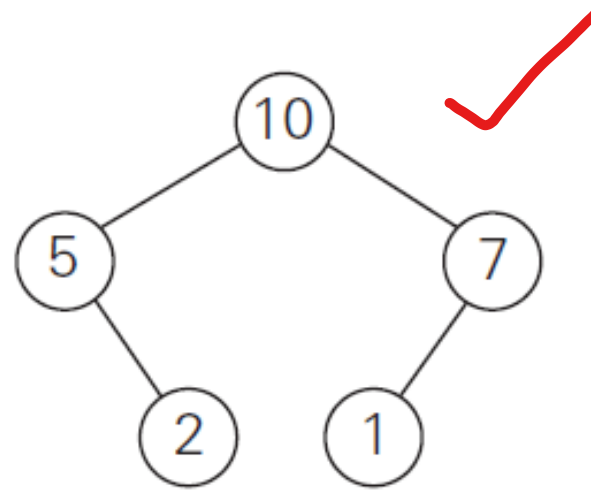
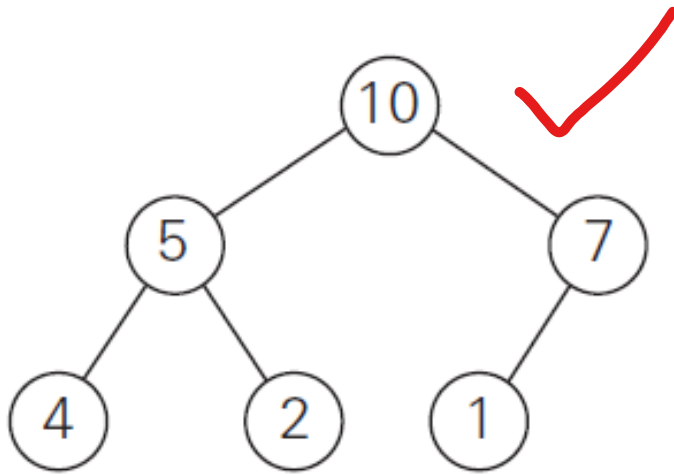
- Nº de folhas =  $\text{ceil}(n / 2)$
- Nº de nós que não são folhas =  $\text{floor}(n / 2)$



# Critérios de ordem

- **MIN-HEAP** : O valor/chave de um nó não é superior ao do seus filhos
- A sequência de valores em qualquer caminho da raiz da árvore até uma folha é não-decrescente
- **MAX-HEAP** : O valor/chave de um nó não é inferior ao do seus filhos
- A sequência de valores em qualquer caminho da raiz da árvore até uma folha é não-crescente
- Podem existir elementos/chaves repetidos !

# São MAX-HEAPS ?

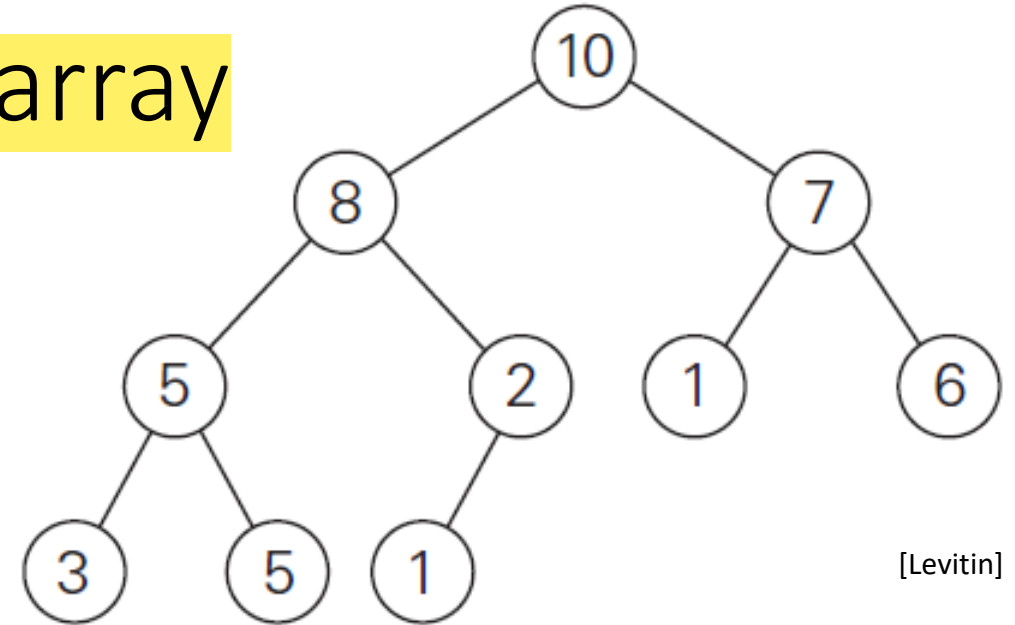


[Levitin]

# Representação usando um array

0	1	2	3	4	5	6	7	8	9
10	8	7	5	2	1	6	3	5	1

folhas



- Armazenar de modo contíguo, da esquerda para a direita, num array
- $\text{LeftChild}(i) = 2 \times i + 1$ , se existir Da forma, LeftChild de (i) está no índice...
- $\text{RightChild}(i) = 2 \times (i + 1)$ , se existir
- $\text{Parent}(i) = (i - 1) \text{ div } 2$ , se  $i > 0$

# Eficiência computacional

- Consultar o elemento de maior/menor prioridade  $O(1)$
- Adicionar um elemento  $O(\log n)$ 
  - Pode ser necessário reorganizar a heap
- Apagar o elemento de maior/menor prioridade  $O(\log n)$ 
  - Pode ser necessário reorganizar a heap
- No pior caso, é necessário percorrer o caminho mais longo definido na heap !!

# MIN-Heaps

# MIN-Heap

```
// The type for MinHeap structures
typedef struct _Heap MinHeap;

// The type for item comparator functions
typedef int (*compFunc)(const void* p1, const void* p2);

// The type for item printer functions
typedef void (*printFunc)(void* p);

// CREATE/DESTROY

MinHeap* MinHeapCreate(int capacity, compFunc compF, printFunc printF) ;

void MinHeapDestroy(MinHeap** pph) ;
```

# MIN-Heap

```
// The heap data structure
struct _Heap {
    void** array;
    int capacity;
    int size;
    compFunc compare;
    printFunc print;
};
```

# MIN-Heap

```
// GETTERS

int MinHeapCapacity(MinHeap* ph) ;

int MinHeapSize(MinHeap* ph) ;

int MinHeapIsEmpty(MinHeap* ph) ;

int MinHeapIsFull(MinHeap* ph) ;

void* MinHeapGetMin(MinHeap* ph) ;
```



# MIN-Heap

```
// MODIFY

void MinHeapInsert(MinHeap* ph, void* item) ;

void MinHeapRemoveMin(MinHeap* ph) ;

// CHECK/VIEW

int MinHeapCheck(MinHeap* ph) ;

void MinHeapView(MinHeap* ph) ;
```

# MAX-Heaps

# MAX-Heap

```
MaxHeap* MaxHeapCreate(int capacity, compFunc compF, printFunc printF) {
    MaxHeap* h = (MaxHeap*)malloc(sizeof(MaxHeap)); // alloc heap header
    if (h == NULL) abort();
    h->array = (void**)malloc(capacity * sizeof(void*)); // alloc array
    if (h->array == NULL) {
        free(h);
        abort();
    }
    h->capacity = capacity;
    h->size = 0;
    h->compare = compF;
    h->print = printF;
    return h;
}
```

# MAX-Heap

```
void MaxHeapDestroy(MaxHeap** pph) {  
    MaxHeap* ph = *pph;  
    if (ph == NULL) return;  
    free(ph->array);  
    free(ph);  
    *pph = NULL;  
}
```

# MAX-Heap

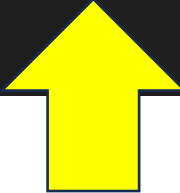
```
int MaxHeapCapacity(const MaxHeap* ph) { return ph->capacity; }

int MaxHeapSize(const MaxHeap* ph) { return ph->size; }

int MaxHeapIsEmpty(const MaxHeap* ph) { return ph->size == 0; }

int MaxHeapIsFull(const MaxHeap* ph) { return ph->size == ph->capacity; }

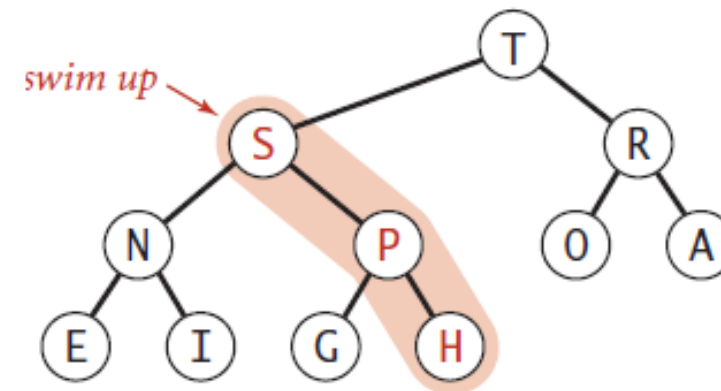
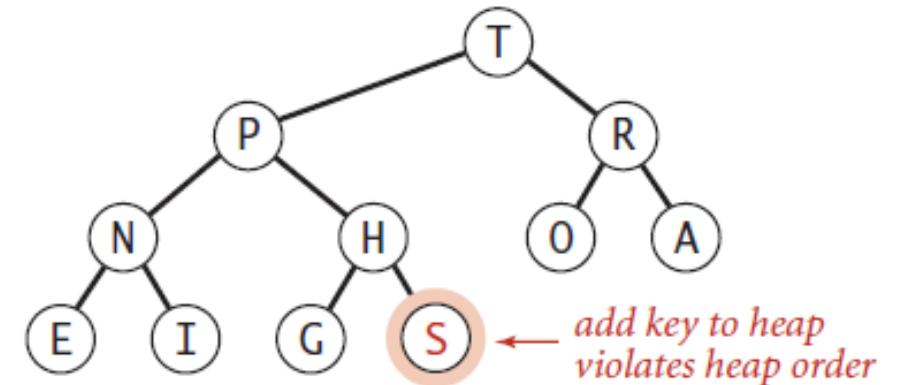
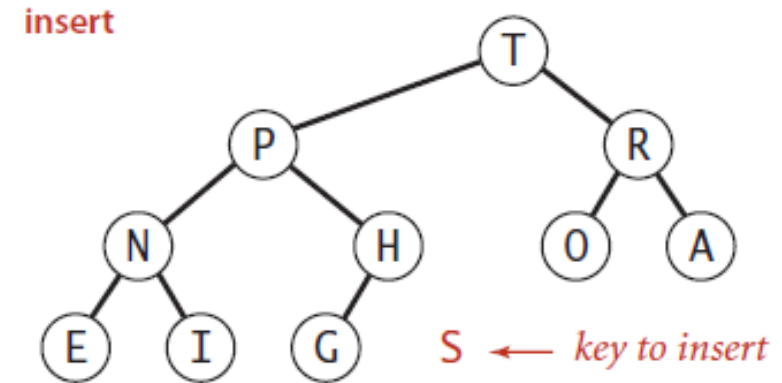
void* MaxHeapGetMax(const MaxHeap* ph) {
    assert(!MaxHeapIsEmpty(ph));
    return ph->array[0];
}
```



# MAX-Heap – Funções auxiliares

```
// n is the index of a node (n in [0, size[).  
// _child(n, 1) is the index of the first child of node n, if < size.  
// _child(n, 2) is the index of the second child of node n, if < size.  
static inline int _child(int n, int c) { return 2 * n + c; }  
  
// _parent(n) is the index of the parent node of node n, if n>0.  
static inline int _parent(int n) {  
    assert(n > 0);  
    return (n - 1) / 2;  
}
```

# Adicionar um elemento



[Sedgewick & Wayne]

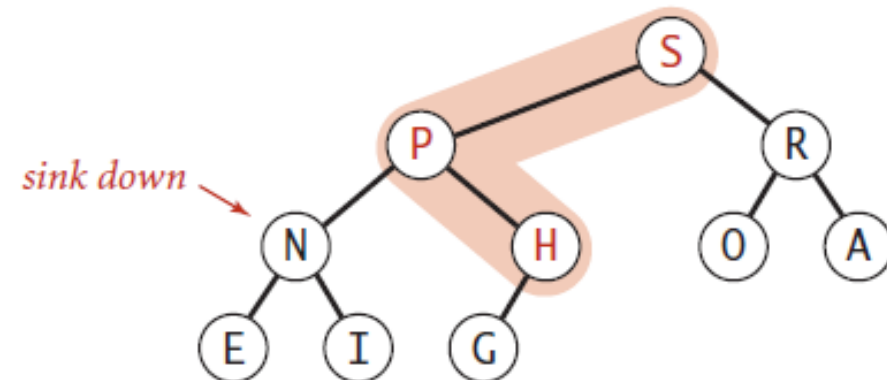
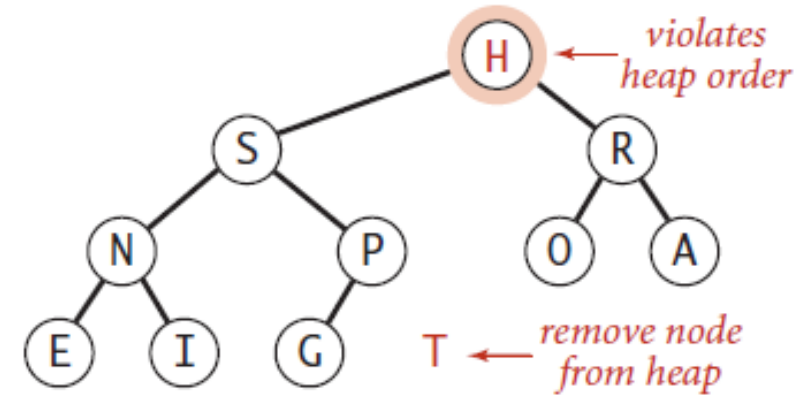
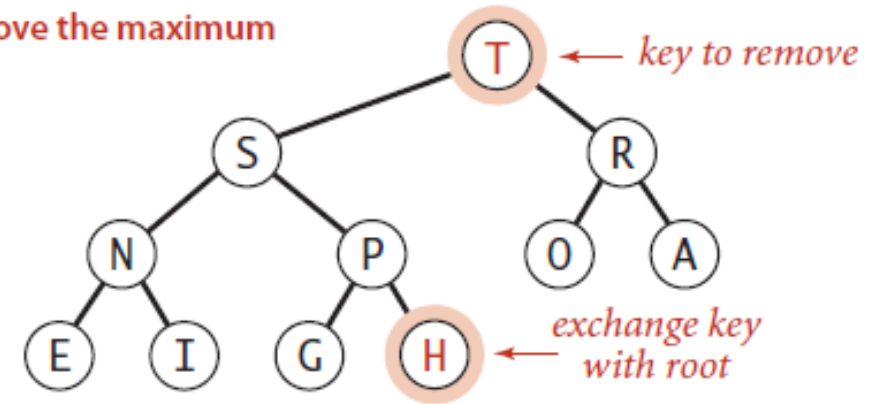
# MAX-Heap – Adicionar e posicionar elemento

```
void MaxHeapInsert(MaxHeap* ph, void* item) {  
    assert(!MaxHeapIsFull(ph));  
    // start at the first vacant spot (just after the last occupied node)  
    int n = ph->size;  
    while (n > 0) {  
        int p = _parent(n);  
        // if item not larger than _parent, then we've found the right spot!  
        if (ph->compare(item, ph->array[p]) <= 0) break;  
        // otherwise, move down the item at node p to open up space for new item  
        ph->array[n] = ph->array[p];  
        // update  
        n = p; // p is the new vacant spot  
    }  
    ph->array[n] = item; // store item at node n  
    ph->size++;  
}
```



# Remover o maior

remove the maximum



[Sedgewick & Wayne]

# MAX-Heap – Remover e reorganizar

```
void MaxHeapRemoveMax(MaxHeap* ph) {  
    assert(!MaxHeapIsEmpty(ph));  
  
    ph->size--; // NOTE: we're decreasing the size first!  
    int n = 0; // the just emptied spot... must fill it with largest child  
    while (1) {  
        // index of first child  
        int max = _child(n, 1); // first child (might not exist)  
  
        if (!(max < ph->size)) break; // if no second child, stop looking  
  
        // if second child is larger, choose it  
        if (ph->compare(ph->array[max + 1], ph->array[max]) > 0) {  
            max = max + 1;  
        }  
    }  
}
```

# MAX-Heap – Remover e reorganizar

```
// if largest child is not larger than last, stop looking
if (!(ph->compare(ph->array[max], ph->array[ph->size]) > 0)) break;

// move largest child to fill empty _parent spot
ph->array[n] = ph->array[max];

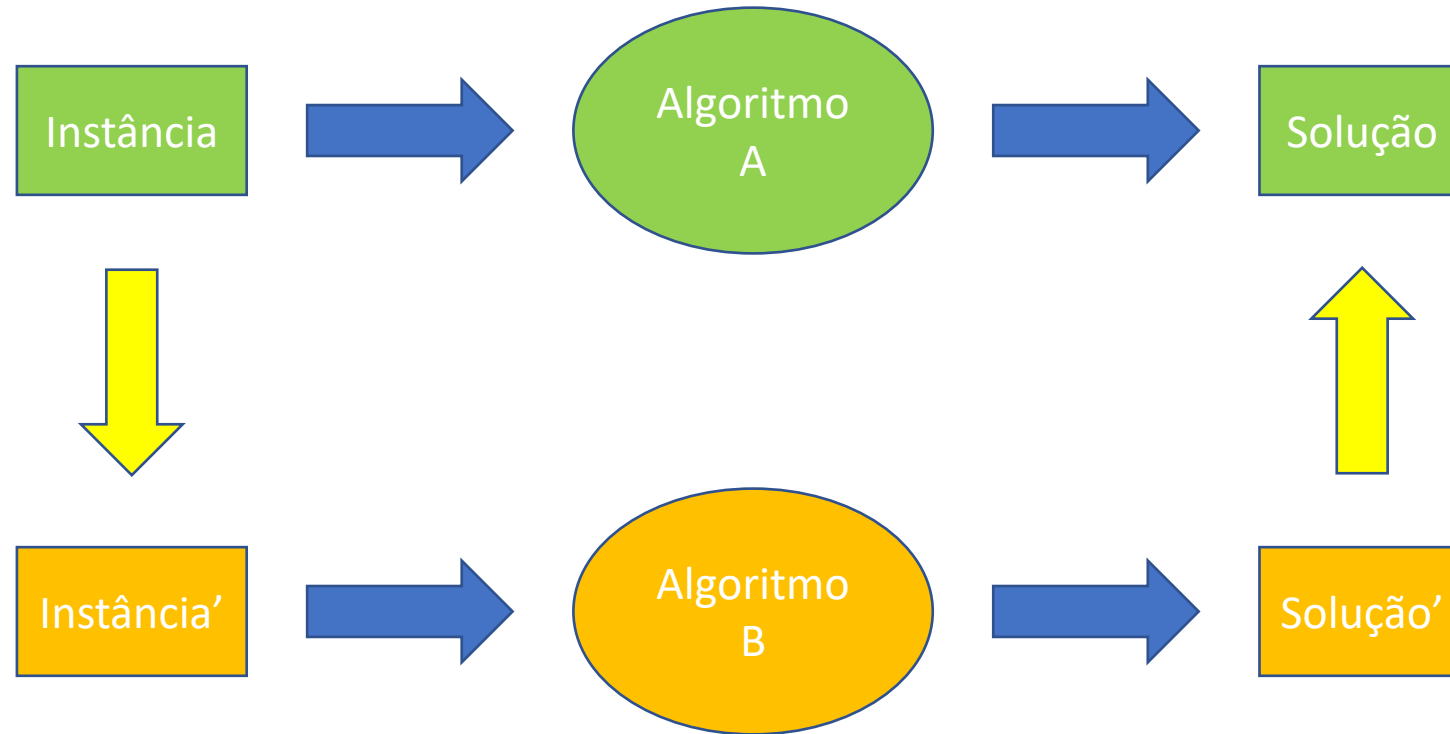
n = max; // now, the largest child spot was just emptied!
}

// move last element to emptied spot
ph->array[n] = ph->array[ph->size];

// mark last element as vacant
ph->array[ph->size] = NULL;
}
```

# Algoritmo Heap-Sort

# A estratégia Transform-and-Conquer



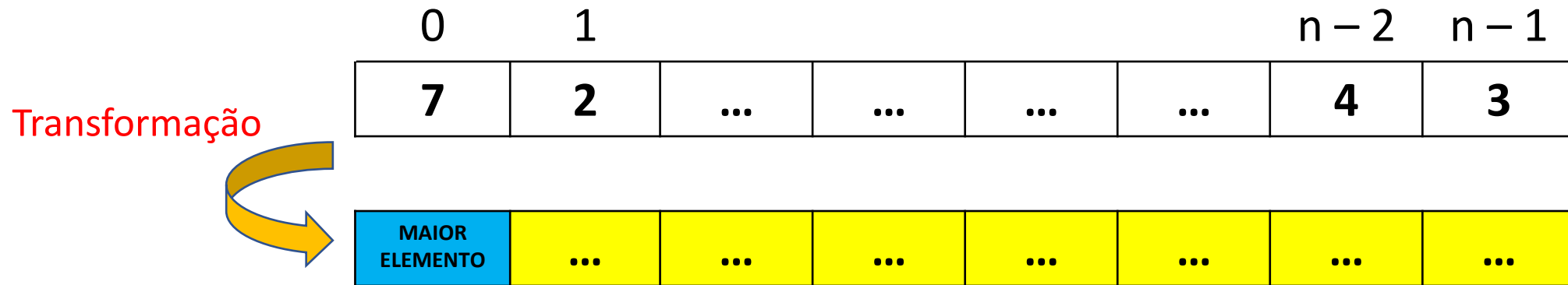
# Transform-and-Conquer

- **Objetivo** : baixo custo computacional !
- **1º passo** : Transformação
- **Modificar a instância dada**, para que seja mais fácil resolver o problema proposto
- **2º passo** : Conquista
- **Resolver a instância modificada** e obter a solução desejada

# T&C – Como ordenar um array ?

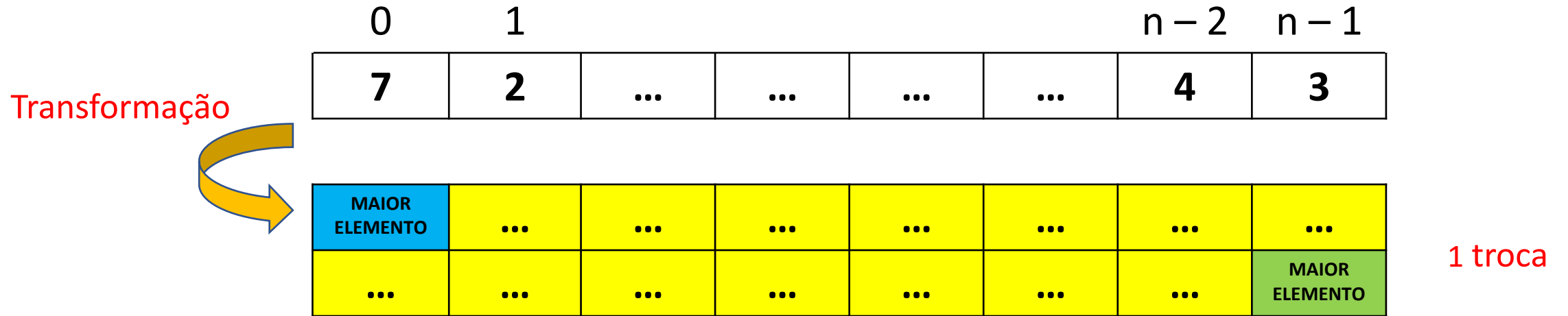
0	1					$n - 2$	$n - 1$
<b>7</b>	<b>2</b>	...	...	...	...	<b>4</b>	<b>3</b>

# T&C – Como ordenar um array ?

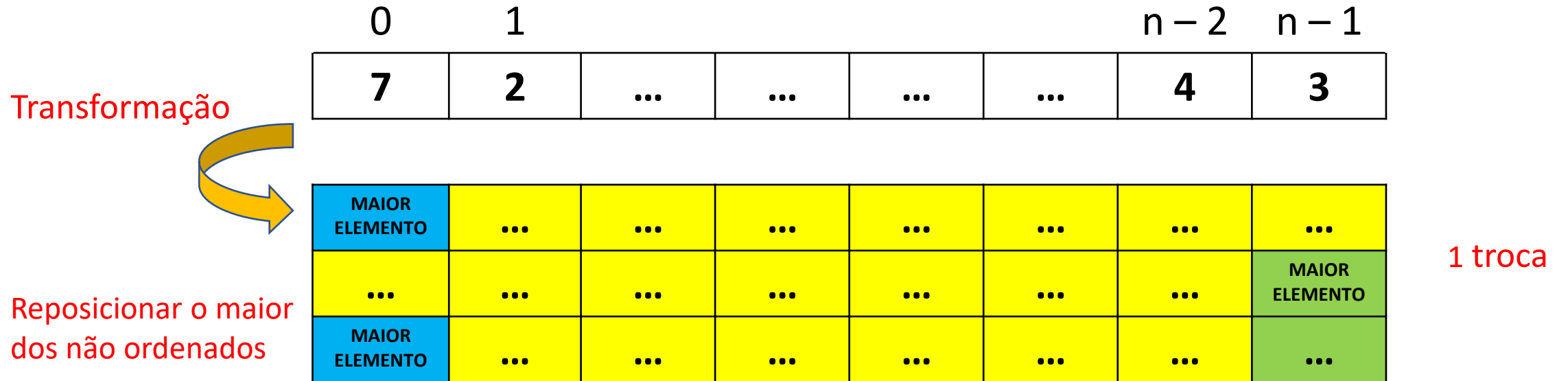




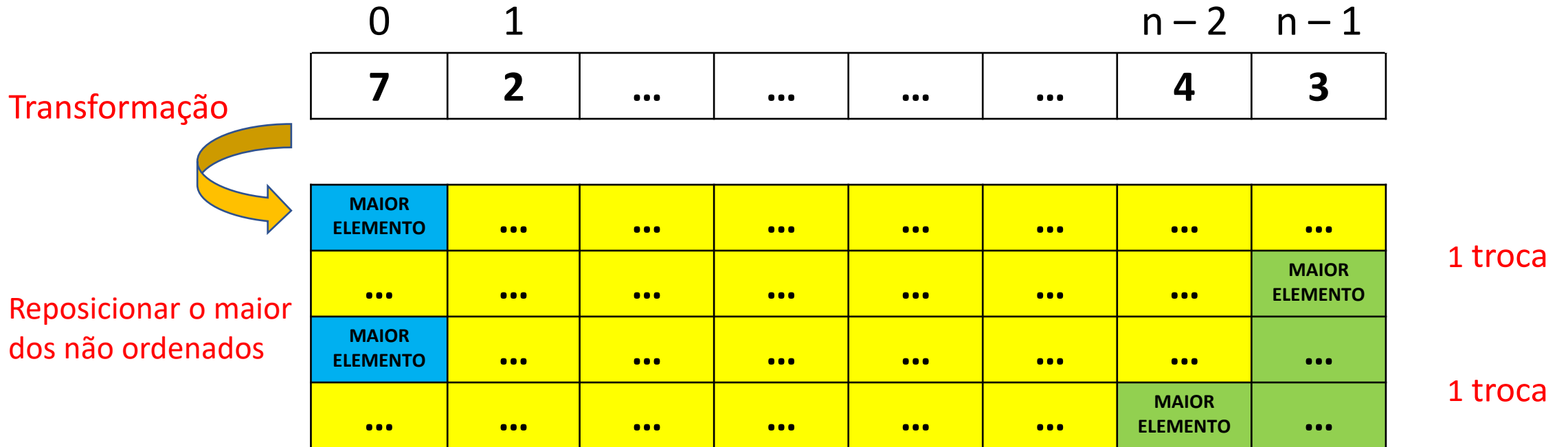
# T&C – Como ordenar um array ?



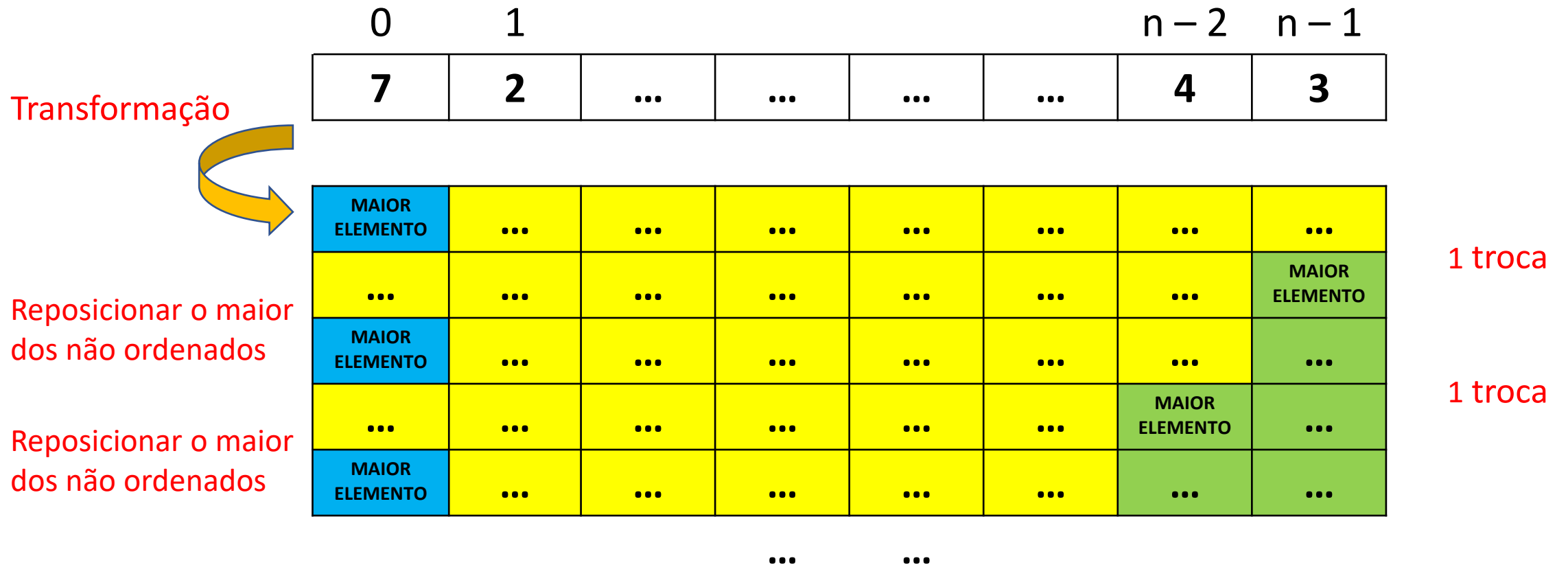
# T&C – Como ordenar um array ?



# T&C – Como ordenar um array ?



# T&C – Como ordenar um array ?



# T&C – Como ordenar um array ?

- **Objetivo** : baixo custo computacional !
- Como **obter** o sucessivamente **o maior elemento** de um conjunto, **sem manter ordenado** esse conjunto de elementos ?
- **Solução** : usar uma representação alternativa – **MAX-HEAP**
- E **não usar espaço de memória adicional**, apenas o array dado

# Estratégia T&C

Dado um **array** de **n elementos**

Construir uma MAX-HEAP

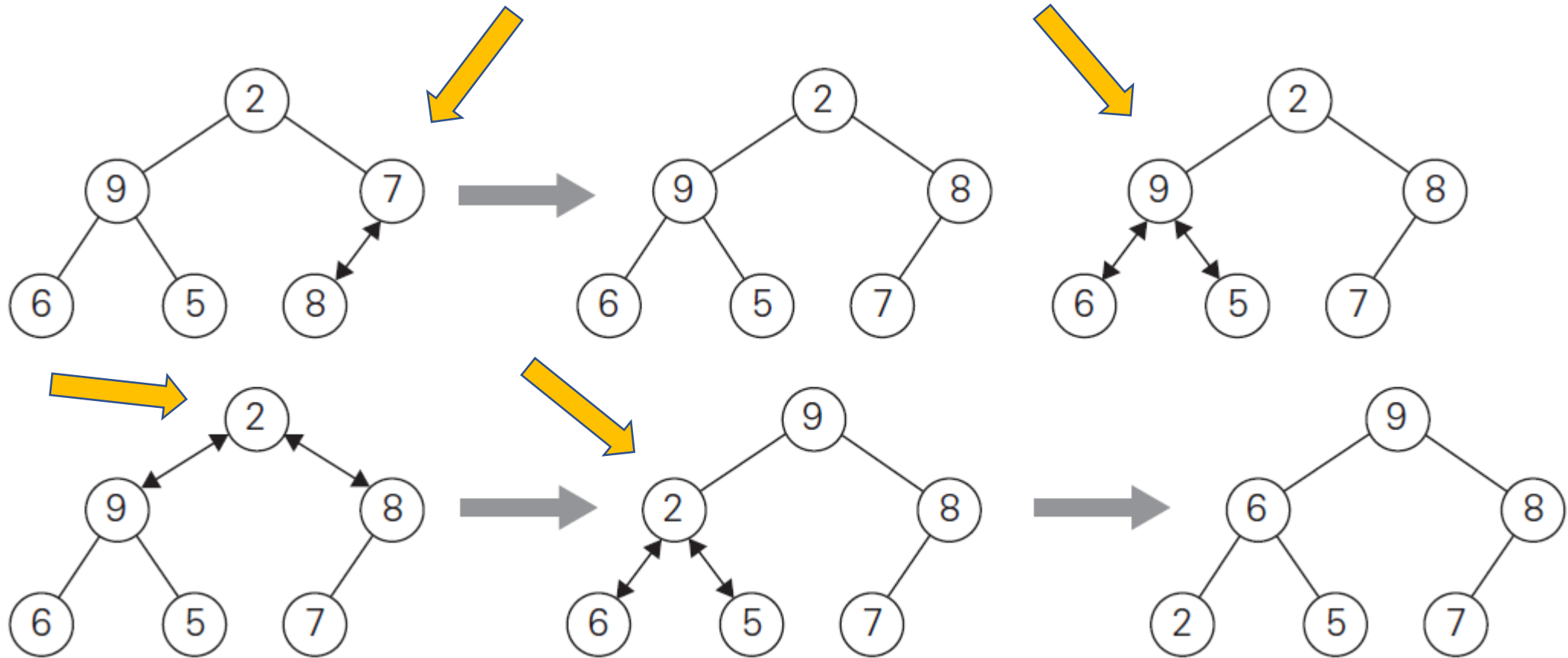
**Repetir  $(n - 1)$  vezes**

Levar o **maior elemento** da MAX-HEAP para **posição final** – **1 TROCA**

**Reorganizar** os elementos não ordenados para **MAX-HEAP** – **1 x fixHeap**

- Algoritmo **in-place** !!

# Construção da MAX-HEAP, dado o array



[Levitin]

# Construção da MAX-HEAP

```
void heapBottomUp( int a[], int n ) {  
    for(int i = n / 2 - 1; i >= 0; i-- )  
        fixHeap( a, i, n );  
}
```

// Para cada elemento  
// que não é folha,  
// reposicioná-lo,  
// se necessário



# Construção de uma MAX-HEAP

```
void fixHeap( int a[], int index, int n ) {  
    int child;  
    for( int tmp = a[index]; leftChild(index) < n; index = child ) {  
        child = leftChild(index);  
        if( child != (n - 1) && a[child + 1] > a[child] ) child++;  
        if( tmp < a[child] ) a[index] = a [child];  
        else break;  
    }  
    array[index] = tmp;  
}
```

// The largest  
// moves up,  
// if needed

// Final position

# Tarefa

0	1	2	3	4
<b>7</b>	<b>2</b>	<b>6</b>	<b>4</b>	<b>3</b>

- Transformar numa MAX-HEAP usando o algoritmo **heapBottomUp**

# Heap Sort

```
void heapSort( int a[], int n ) {  
    heapBottomUp( a, n );  
    for( int i = n - 1; i > 0; i-- ) {  
        swap( &a[0], &a[i] );  
        fixHeap( a, 0, i );  
    }  
}
```

// Só a[0] pode  
// necessitar de ser  
// reposicionado !!

# Tarefa

0	1	2	3	4
<b>7</b>	<b>2</b>	<b>6</b>	<b>4</b>	<b>3</b>

- Ordenar usando o algoritmo **Heap-Sort**

# Tarefa

0	1	2	3	4	5
2	9	7	6	5	8

- Ordenar usando o algoritmo **Heap-Sort**