Heap (Cormen)

→ Runtime: $O(n \log n)$, like merge / quick

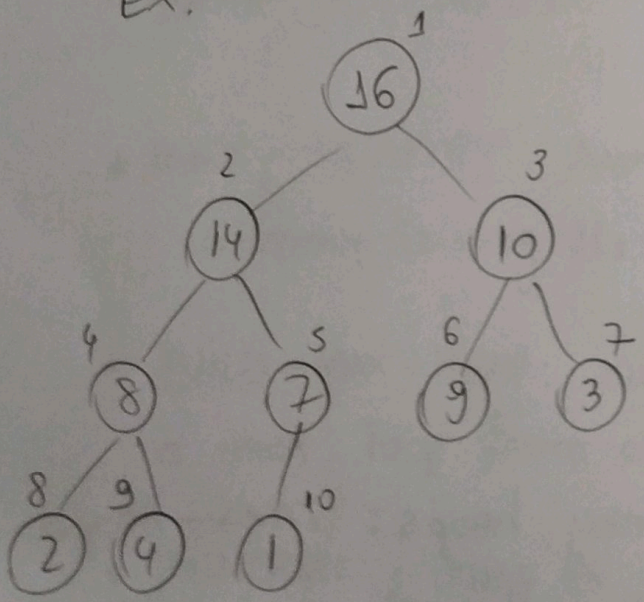
→ heapsort: like insertion sort, sorts in place, only a constant number of array elements are stored outside the input array at any time.

→ heapsort combines the better attributes of the two sorting algorithms we already discussed.

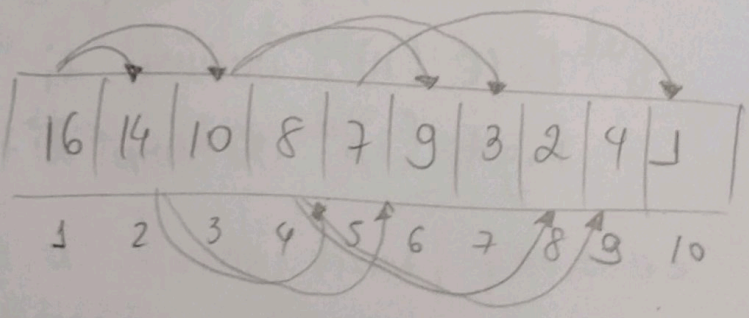
X.1 Heaps

↳ the (binary) heap data structure is an array that can view as a nearly complete binary tree.

EX:



(a) heap as binary tree



(b) heap as an array

→ An array A that represents a heap is an object with two attributes:

A.lenght → gives the number of elements in the array

A.heap.size → how many elements in the heap are stored within array A

Only the elements in $A[1 \ldots heap.size]$ are valids, where $0 \leq heap.size \leq lenght$

→ the root of the tree is $A[1]$, and given the index i of a node, we can easily compute the indices of its parent, left child, and right child:

Parent (i): return $\lfloor i/2 \rfloor$
Left (i): return $2i$
Right (i): return $2i+1$

→ There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property

- In a max-heap, the max heap property is that for every node i other than root:

$$A[parent(i)] \geq A[i]$$

the value of a node is at most the value of its parent.

- in a min-heap:

$$A[parent(i)] \leq A[i]$$

and the smallest element in a min-heap is the root.

→ For heap sort, we use max-heaps. So, we will need some functions:

- MAX-HEAPIFY: runs in $O(\lg n)$ time, is the key to maintaining the max-heap property
- BUILD-MAX-HEAP: runs in a linear time, produces a max heap from an unordered input array
- HEAPSORT: runs in $O(n \lg n)$ time, sorts an array in place

- MAX-HEAP-INSERT, HEAP-EXTRACT-MAX, HEAP-INCREASE-KEY, HEAP-MAXimum procedures: run in $O(\lg n)$ time, allow the heap data structure to implement a priority queue

X.2 Maintaining the heap Property

→ MAX-HEAPIFY

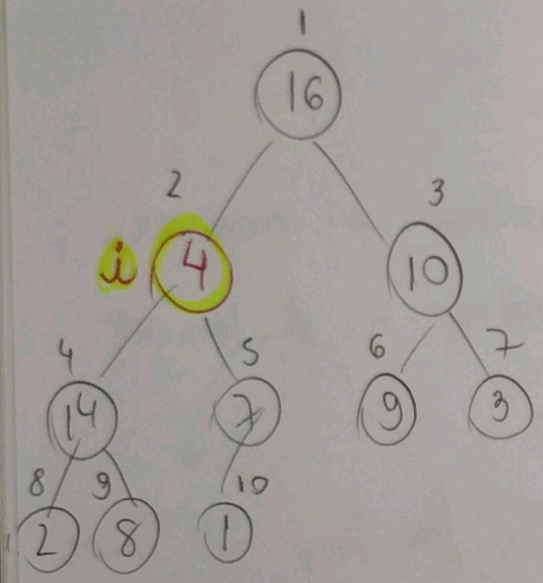
→ Inputs:
- A array
- i index into the array

→ it lets the value of A[i] "float down" in the max-heap so that the subtree rooted at index i obeys the max-heap property

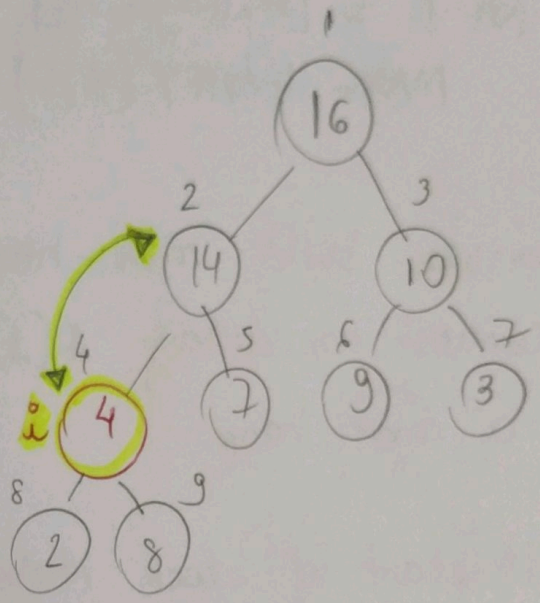
```
MAX-HEAPIFY (A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if ≤ A.heapsize and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
```

```
8     if largest ≠ i
9         exchange A[i] with A[largest]
10        MAX-HEAPIFY(A, largest)
```

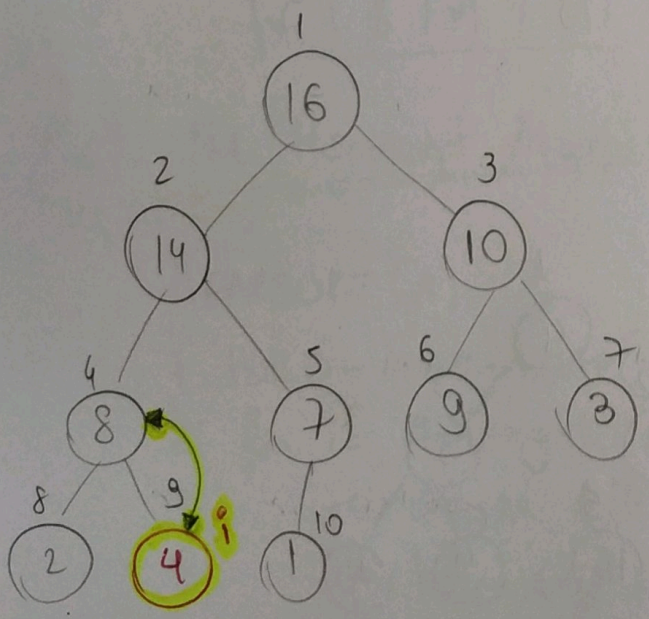
EX:



(a) initial configuration with A[2] = i violating the max heap property



(b) the max heap property is restored for node 2, swapping A[2] with A[4].
However node 4 breaks the property



(c) Swapping A[4] and A[9], node 4 is fixed up

X.3 Building a heap

```
BUILD-MAX-HEAP(A)
1 A.heap-size = A.lenght
2 for i = ⌊A.length/2⌋ downto 1
3     MAX-HEAPIFY(A,i)
```

→ Can use build-max-heap in a bottom-up manner to convert an array $A[1 \ldots n]$, where $n = A.lenght$ into a max heap

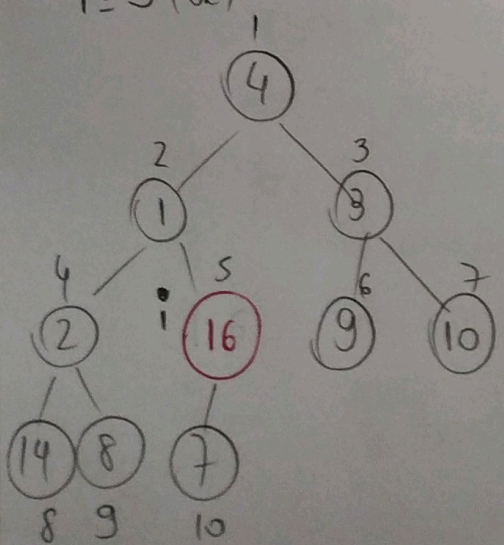
→ At the start of each iteration of the for loop of lines 2-3, each node $i+1, i+2, \ldots, n$ is the root of max-heap.

EX:

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

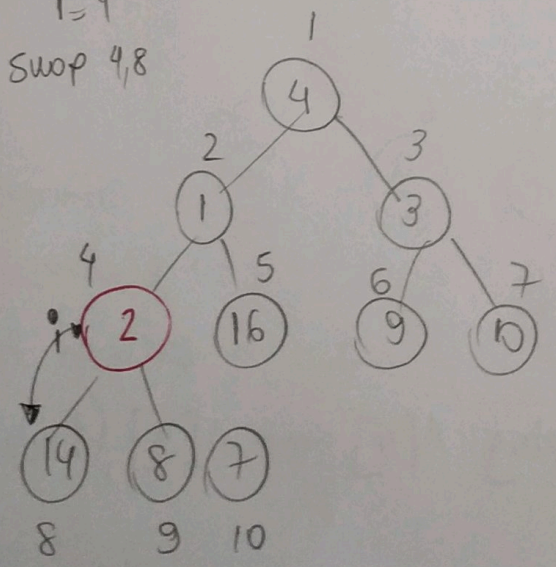
10/2 = 5

i = 5 (ok)

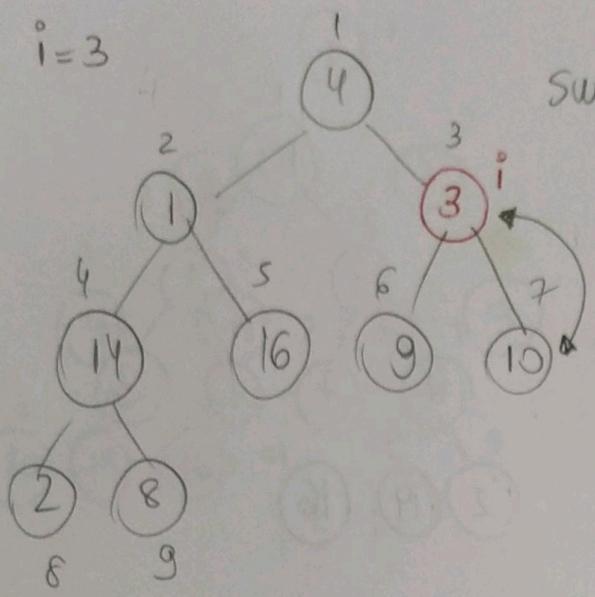


i = 4

swap 4,8

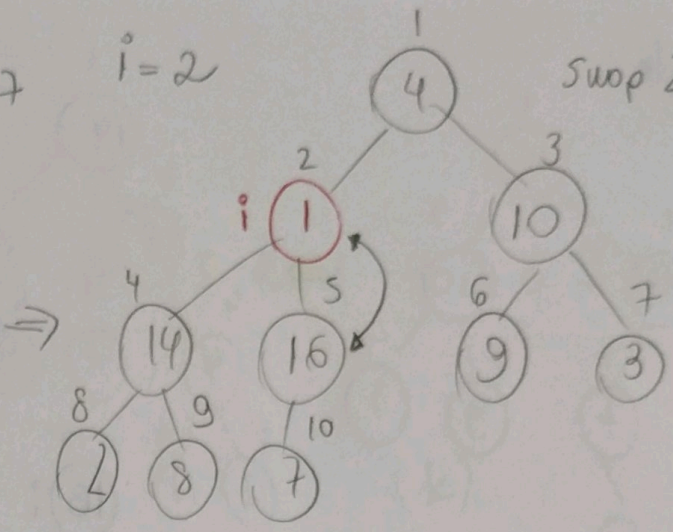


i=3



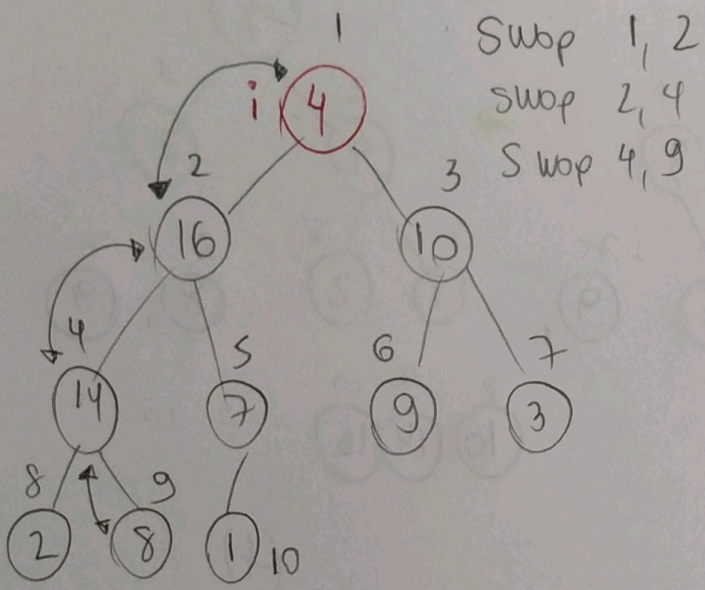
Swap, 3,7

i=2



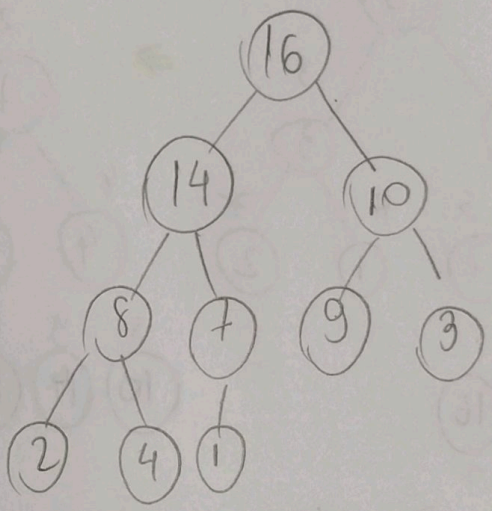
Swap 2,5

i=1



Swap 1, 2
swap 2, 4
Swap 4, 9

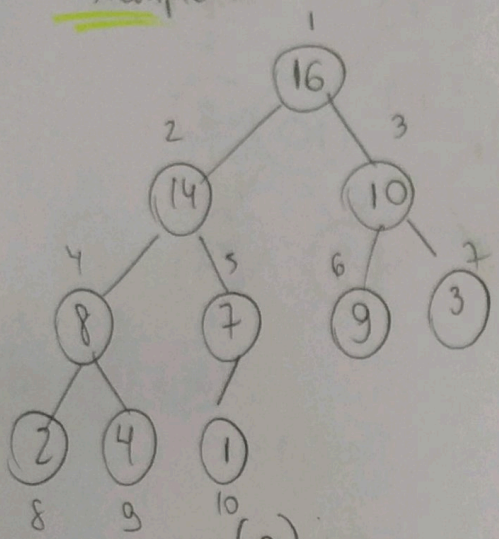
Final



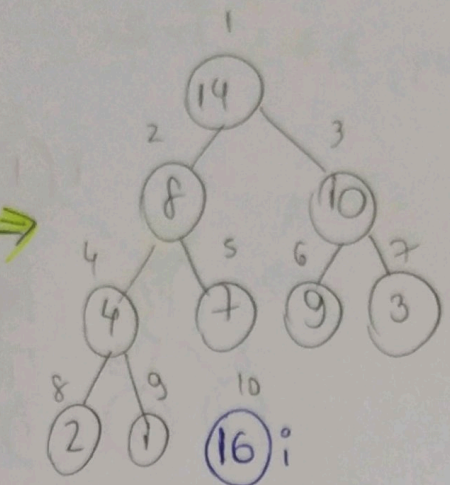
X.4 the Heapsort Algorithm

```
HEAPSORT(A)
1 BUILD-MAX-HEAP(A)
2 for i = A.lenght downto 2
3     exchange A[1] with A[i]
4     A.heap-size = A.heap-size - 1
5     MAX-HEAPIFY(A,1)
```

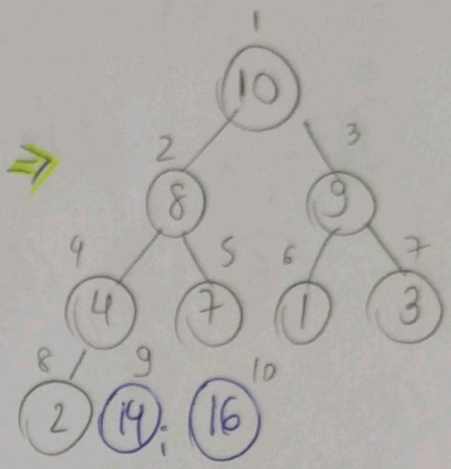
Example :



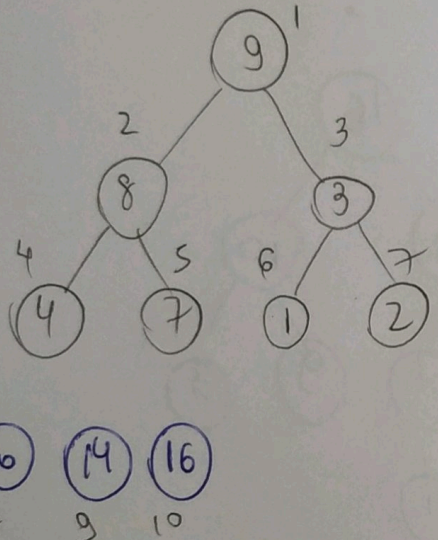
(a)
Creates Heap



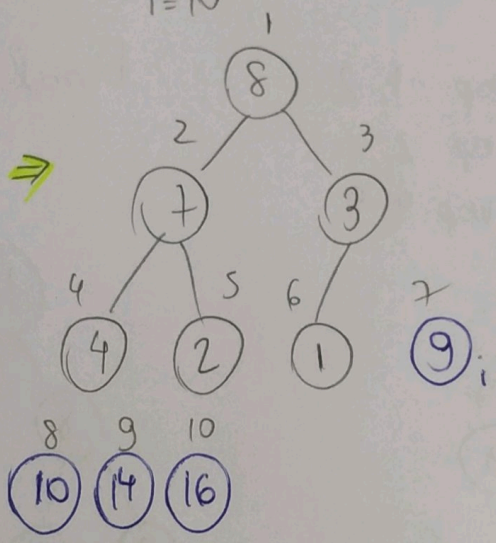
(b)
i = 10



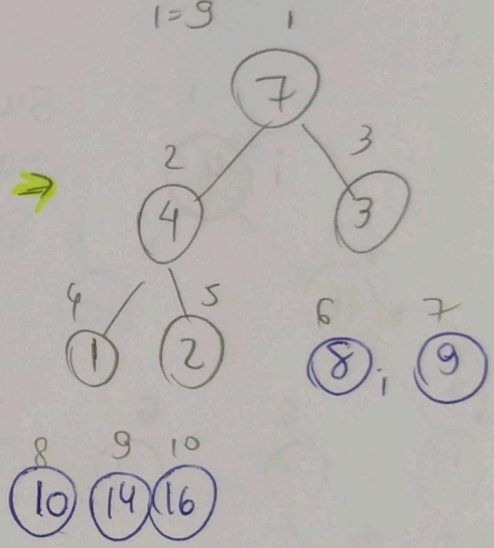
(c)
i = 9



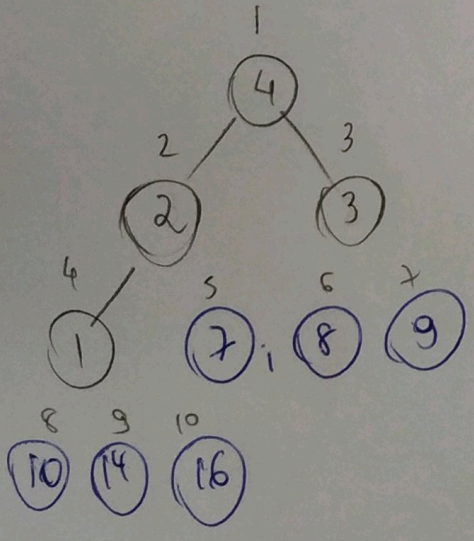
(d)
i = 8



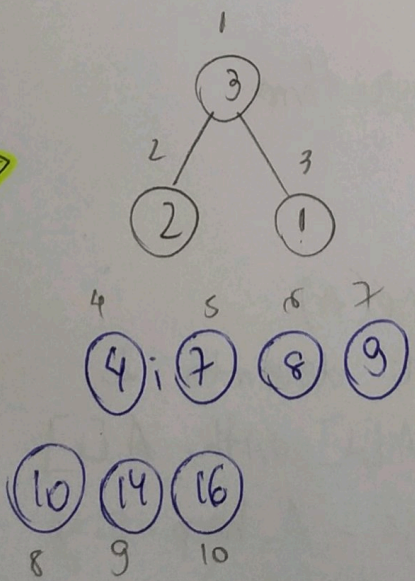
(e)
i = 7



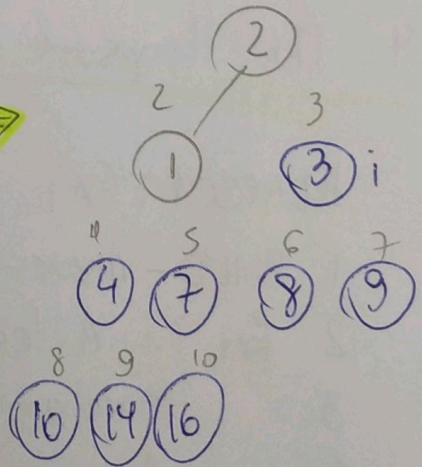
(f)
i = 6



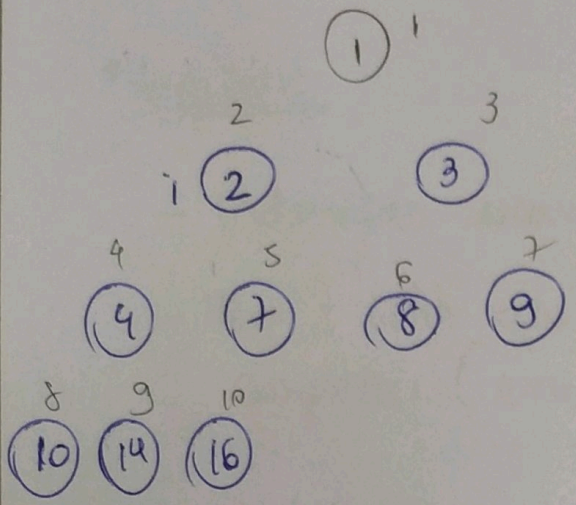
(g)
i = 5



(h)
i = 4



(i)
i = 3



(j)

$i = 2$

$A = [1\ 2\ 3\ 4\ 7\ 8\ 9\ 10\ 14\ 16]$

(k)

the resulting sorted array

• Heap Sort

- ordenção usando heap
- Heap : vetor que simula uma árvore binária completa (exceção do último nível)

- todo elemento "pai" do vetor possui dois elementos como "filhos"

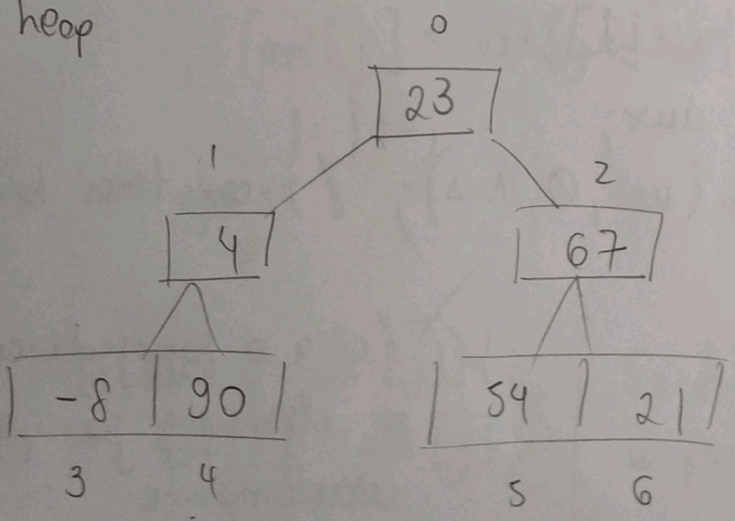
pai	→	filhos
i		2i + 1
		2i + 2

EX:

0	1	2	3	4	5	6
23	4	67	-8	90	54	21

vetor

vetor como heap



elemento 23 (id=0), filhos 4 (2*0+1), id 1
67 (2*0+2), id 2

→ Código (Principal)

```
void heapSort (int *vet, int N){
    int i, aux;
    // cria heap a partir dos dados, da
    // metade do vetor (N-1)/2 até o começo 0
    for(i=(N-1)/2 ; i>=0 ; i--){
        criaHeap(vet, i, N-1);
    }
    // reconstruir heap
    for(i=N-1 ; i>=1 ; i--){
        // pegar o maior elemento da heap e colocar na
        // sua posição correspondente no final do array
        aux = vet[0];
        vet[0] = vet[i];
        vet[i] = aux;
        criaHeap(vet, 0, N-1); // reconstroi heap
    }
}
```

// cria o heap desconsiderando o elemento já posicionado corretamente

- Código possui 2 funções:
  - → Heap Sort
  - → Cria Heap

A cada iteração:
- → elemento do topo da heap (máximo) é acessado
- → esse elemento é colocado na última posição válida do vetor

- Função cria heap (Heapify):

```
void cria heap (int *vet, int i, int f){
    int aux = vet[i];
    int j = i*2 +1;
    while (j<=f){
        if(j<f){
            if(vet[j] < vet[j+1]){
                j=j+1;
            }
        }
        if(aux < vet[j]){
            vet[i] = vet[j];
            i=j ;
            j=2*i +1;
        }
```

Pai tem 2 filhos. Quem é o maior?

Se filho é maior que o pai?

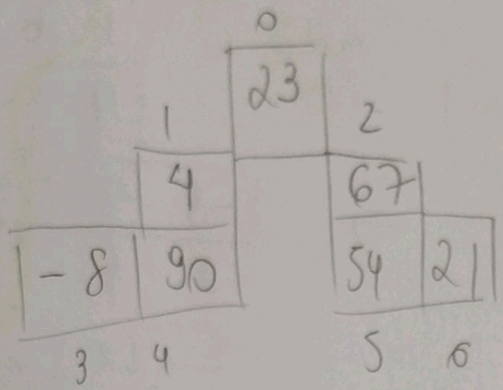
Filho se torna o pai!

```
        else {
            j = f+1;
        }
    }
    vet[i] = aux;
}
```

Antigo pai ocupa o lugar do último filho analisado

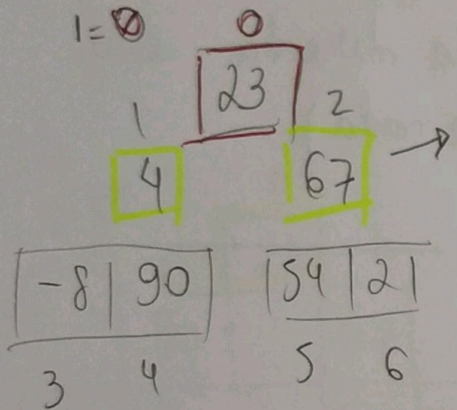
Exemplo: V =	0	1	2	3	4	5	6
	23	4	67	-8	90	54	21

Cria heap(V,0,6)



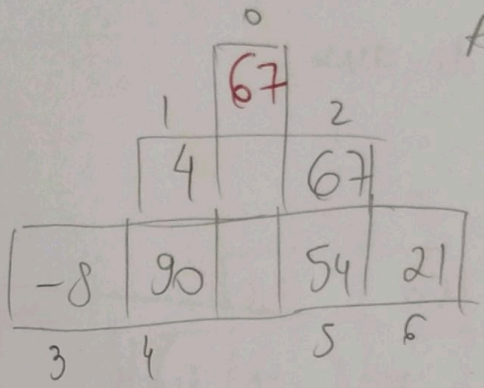
Aux = 23

i=0



→ maior filho

⇒



Aux = 23

"procura maior filho de i(0)"

- 67 > aux

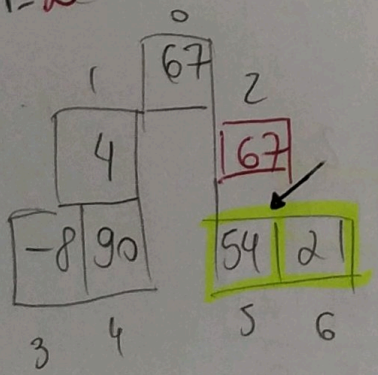
pai recebe valor do filho

pai (v[i]), filho (v[j])

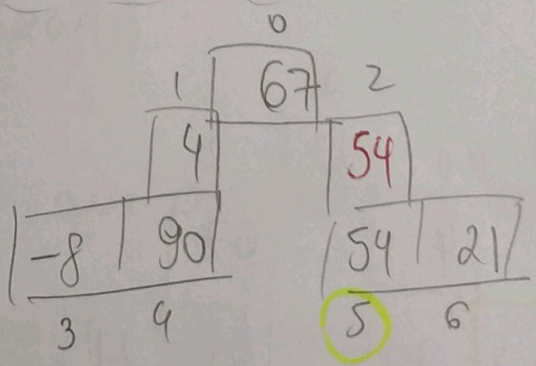
i = recebe 2

j = recebe 5

i=2



Aux = 23



Procura maior filho de i(2)

- 54 > aux

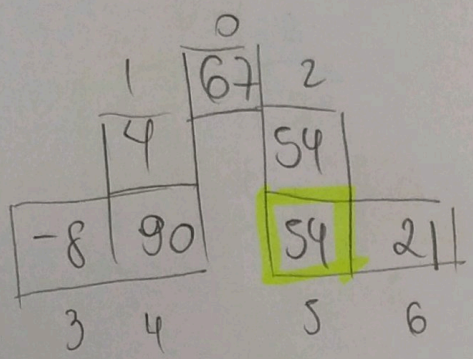
pai (v[i]) recebe valor do filho (v[j])

i = recebe 5

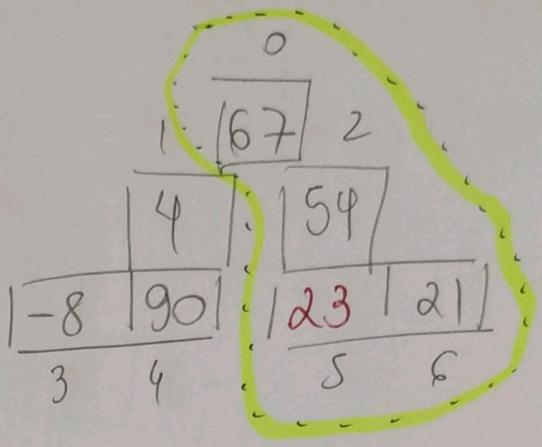
j = recebe 11

j=1, j > fim : terminar

aux = 23



⇒



posição i = 5 recebe o valor de aux

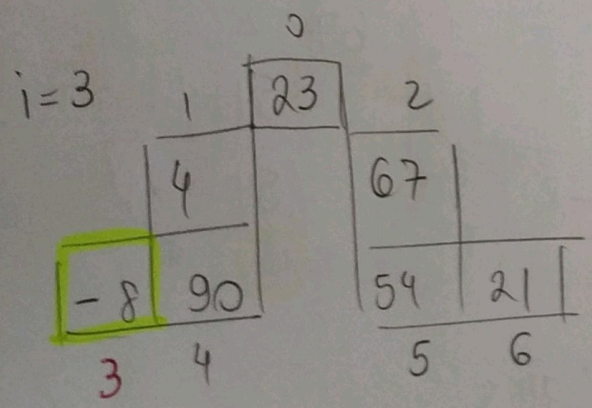
Resultado final (porção da direita, está ordenada)

```
void heapSort (vet, N){
    for(i = (N-1)/2; i >= 0; i--){
        cria heap(vet, i, N-1);
    }
}
```

] cria heap a partir dos dados

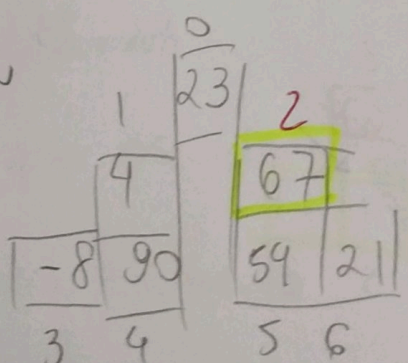
"Elemento pai, fica maior que os filhos"

i=3



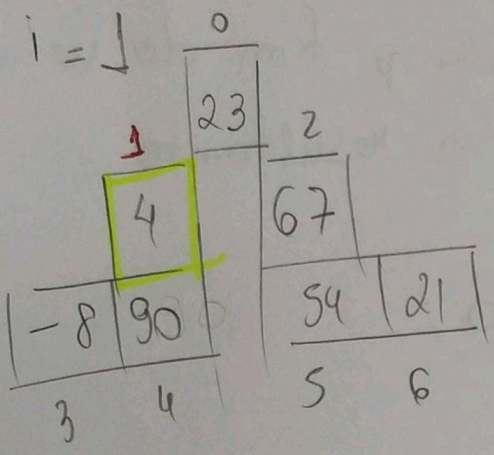
⇒

i=2



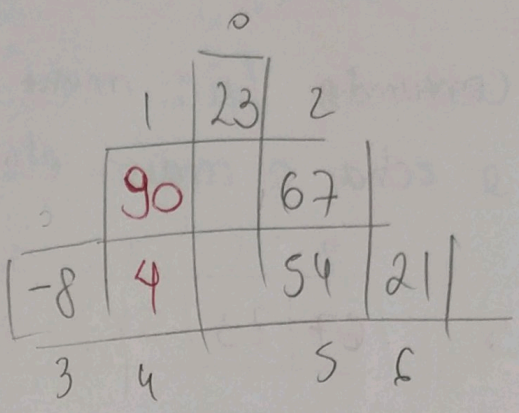
i=3 (7, 8) } não existem filhos, ñ faz nada

"Pai maior que os dois filhos, não faz nada" (06)



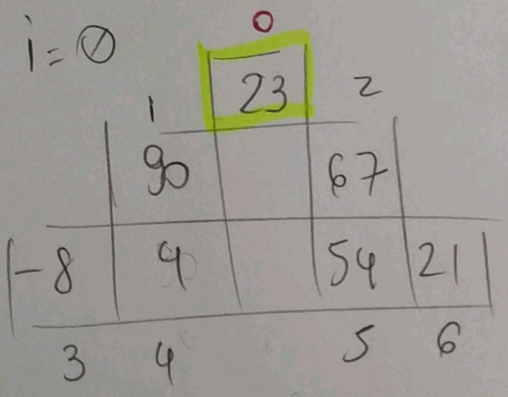
90 > 4

troca pai e filho ⇒



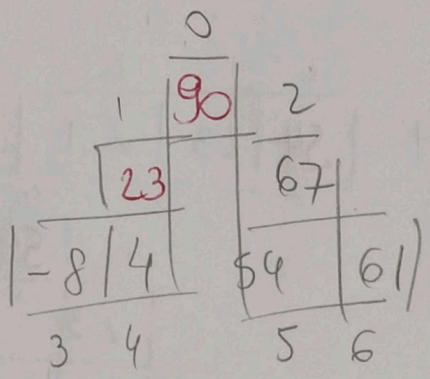
i = 1, filhos $2*1+1 = 3$

$2*1+2 = 4$



90 > 23

troca pai e filho



i = 0 (23)

filhos. $2*0+1 = 1$ (90)

$2*0+2 = 2$ (67)

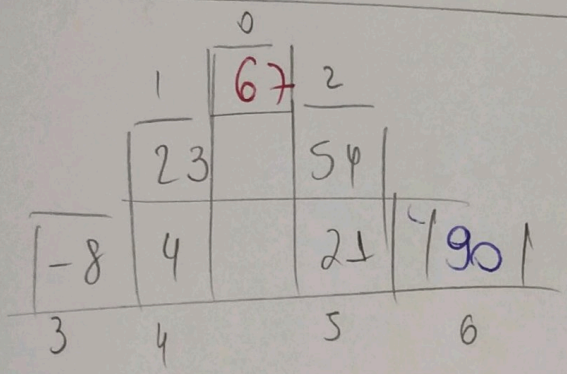
OBS: "todo pai é > que os dois filhos"

No topo, sempre o maior elemento

- 2º comando for: move maior elemento p fim do vetor e acha o maior elemento do vetor resultante

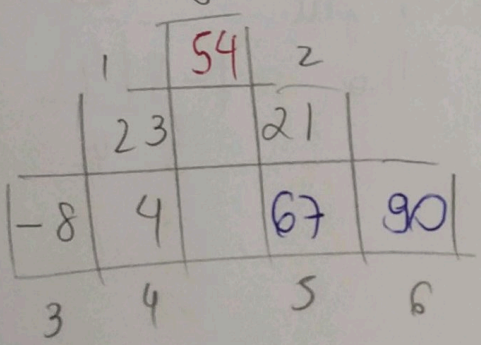
- i-1 = 5

0	1	2	3	4	5
67	23	54	-8	4	21	90



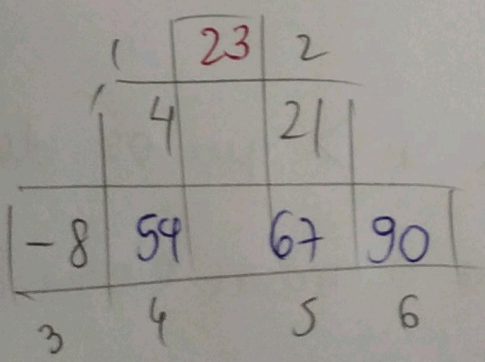
i-1 = 4

0	1	2	3	4		
54	23	21	-8	4	67	90



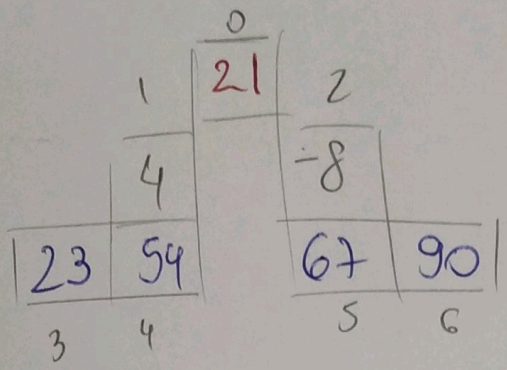
i-1 = 3

0	1	2	3			
23	4	21	-8	54	67	90



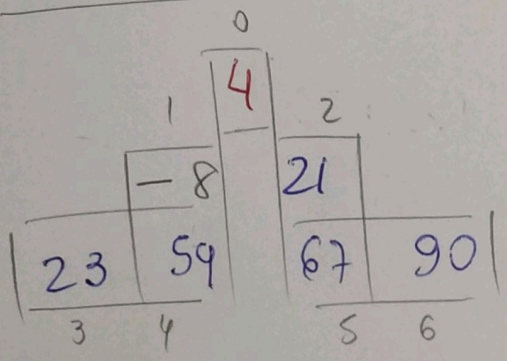
i - j = 2

21	4	-8	23	54	67	90
0	1	2				



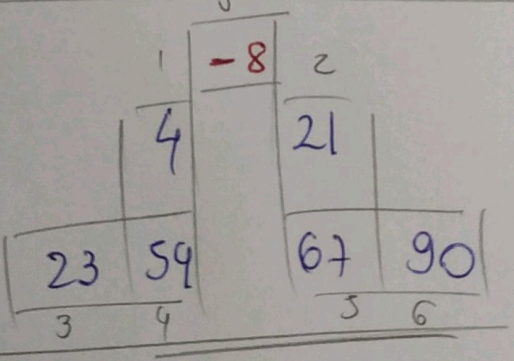
i - j = 1

4	-8	21	23	54	67	90
0	1					



i - j = 0

-8	4	21	23	54	67	90
0						



Ordenado!