MCTA001-13-Algoritmos e Estruturas de Dados I

Aula 08
Algoritmos Eficientes de Ordenação de Vetores

Marcio K. Oikawa
(marcio.oikawa@ufabc.edu.br)

1o. Quadrimestre de 2020

Introdução

- Os melhores algoritmos de ordenação de vetores conseguem realizar a operação em tempo proporcional a $nlog(n)$, onde n é o número de elementos do vetor.
- Exemplo: ordenar um vetor de 1.000.000 de registros
  - Bubble sort: 10^{12} unidades de tempo
  - Algoritmos mais eficientes: 2×10^7 unidades de tempo.
- Os algoritmos mais eficientes necessitam usar métodos mais sofisticados ou estruturas de dados auxiliares (ou ambos).
- Existem muitos algoritmos eficientes, mas os mais famosos são:
  - Mergesort (ordenação por intercalação).
  - Heapsort (ordenação com heap).
  - Quicksort (ordenação rápida).

Problema de Ordenação (*Sorting*)

- Considere um vetor $v[0..n-1]$ de n elementos, $n > 1$, dispostos de forma aleatória. Um algoritmo de ordenação deverá realizar permutações entre esses elementos de modo que, ao final, tenhamos válida a seguinte propriedade: $v[0] \leq v[1] \leq v[2] \leq \cdots \leq v[n-1]$
- Os algoritmos clássicos são iterativos, buscando ordenar parcialmente o vetor a cada iteração.

Ordenação por Intercalação (*Mergesort*)

- O Mergesort é um algoritmo recursivo.
- A cada passo recursivo, deseja-se dividir o vetor em dois sub-vetores.
- Após as divisões recursivas, o algoritmo rearraja os elementos do vetor, ordenando-os.

Mergesort

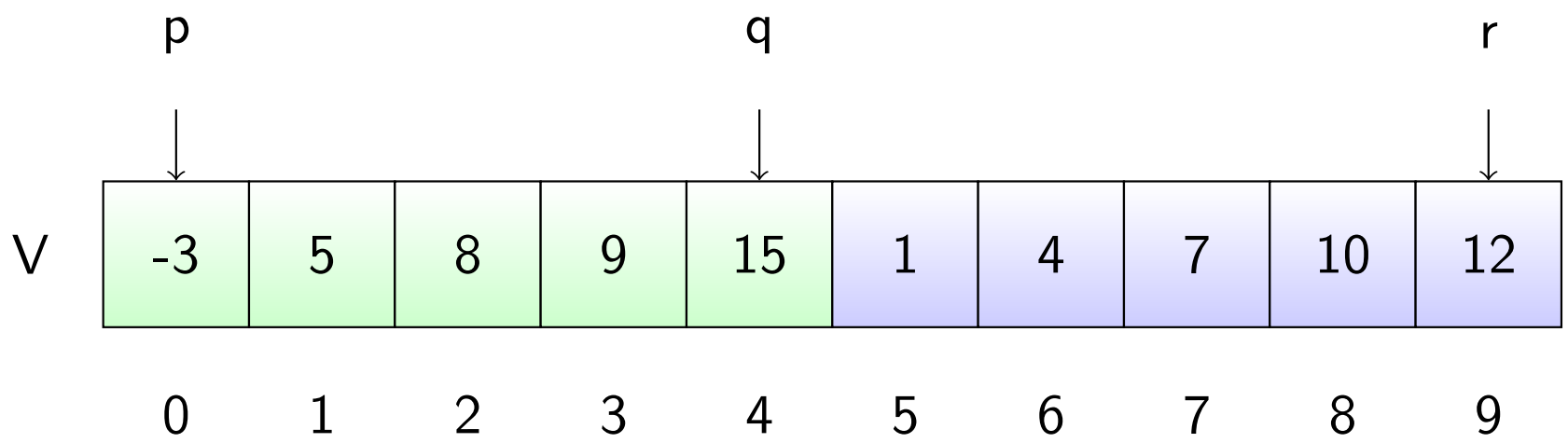
Implementação do *Mergesort*:

```
void mergesort (int* v, int p, int r){
    if (p < r-1) {
        int q = (p + r)/2;
        mergesort (v, p, q);
        mergesort (v, q+1, r);
        intercala (v, p, q, r);
    }
}
```

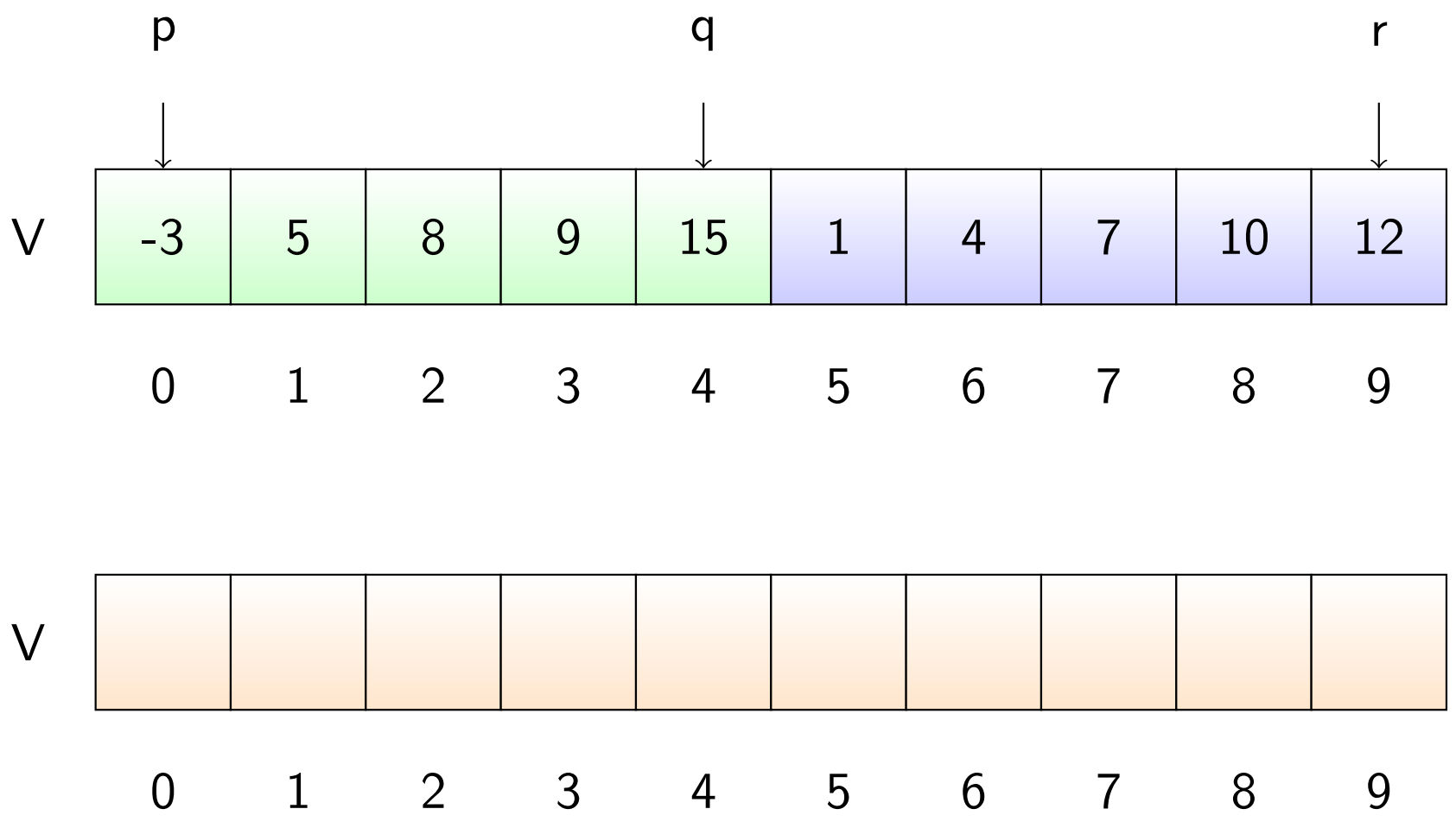
Ordenação por Intercalação (*Mergesort*)

- O segredo do Mergesort está na função "intercala". Vamos ver o seu funcionamento a seguir.

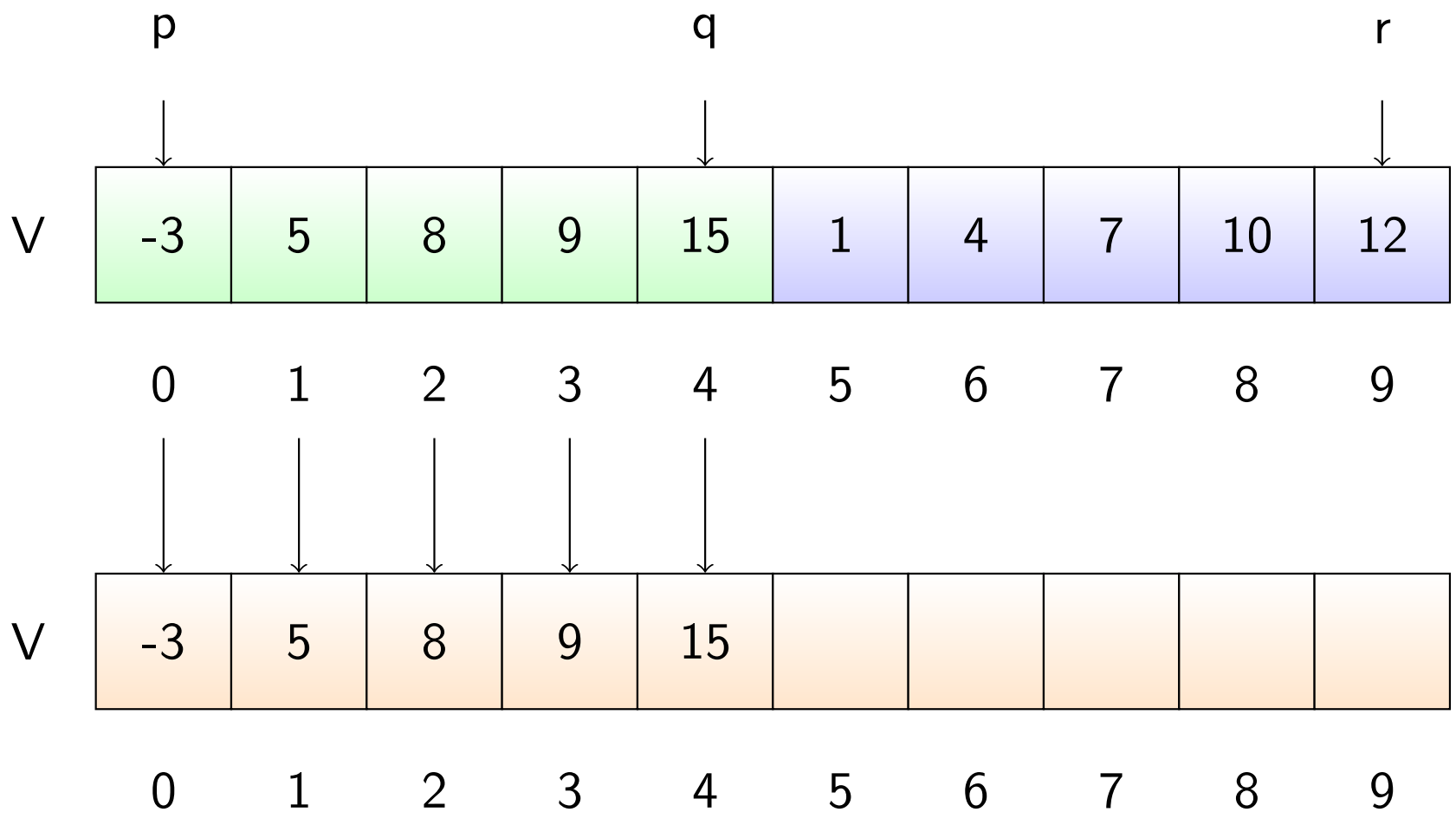
Função de intercalação *(intercala)*



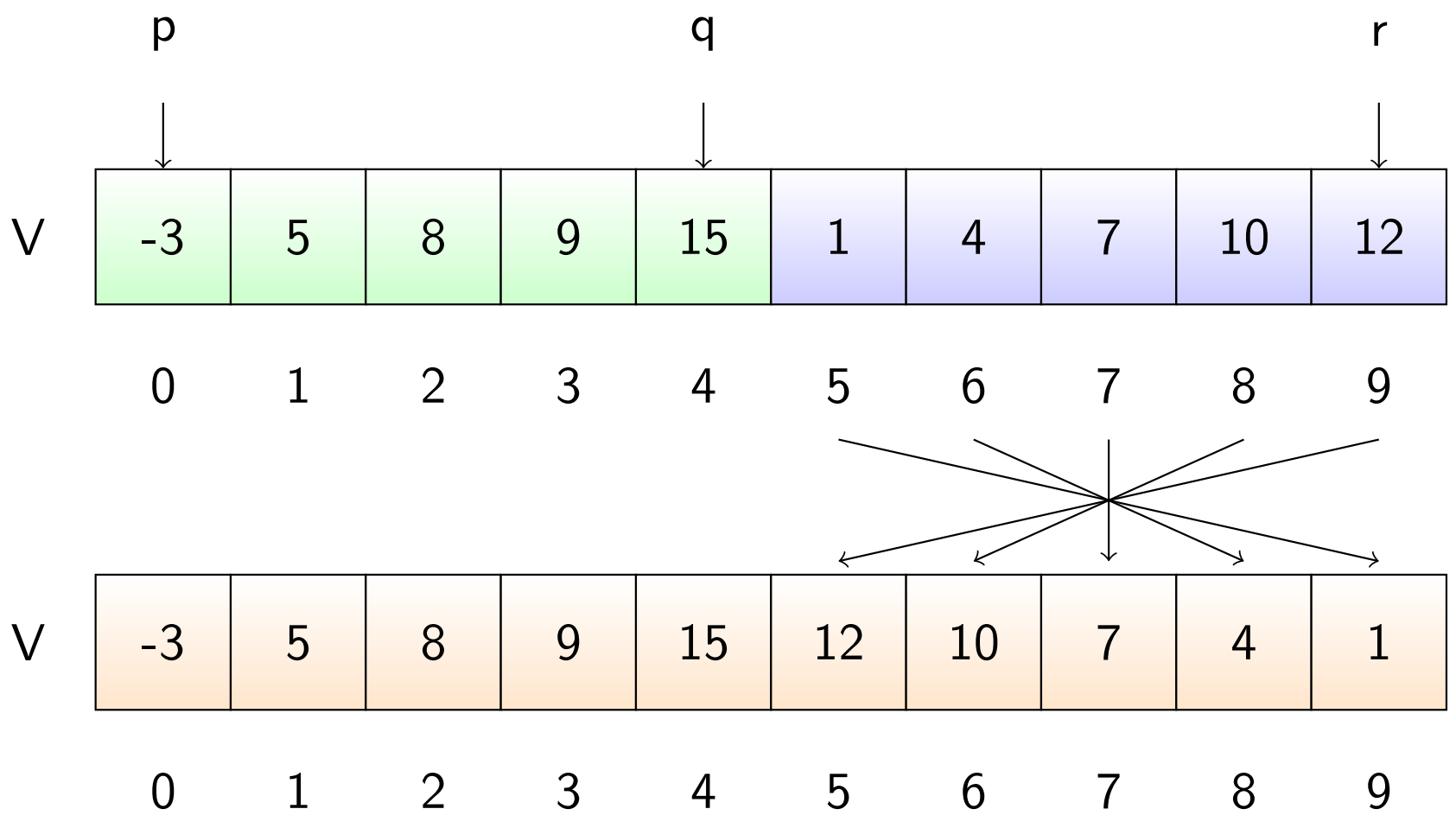
Função de intercalação *(intercala)*



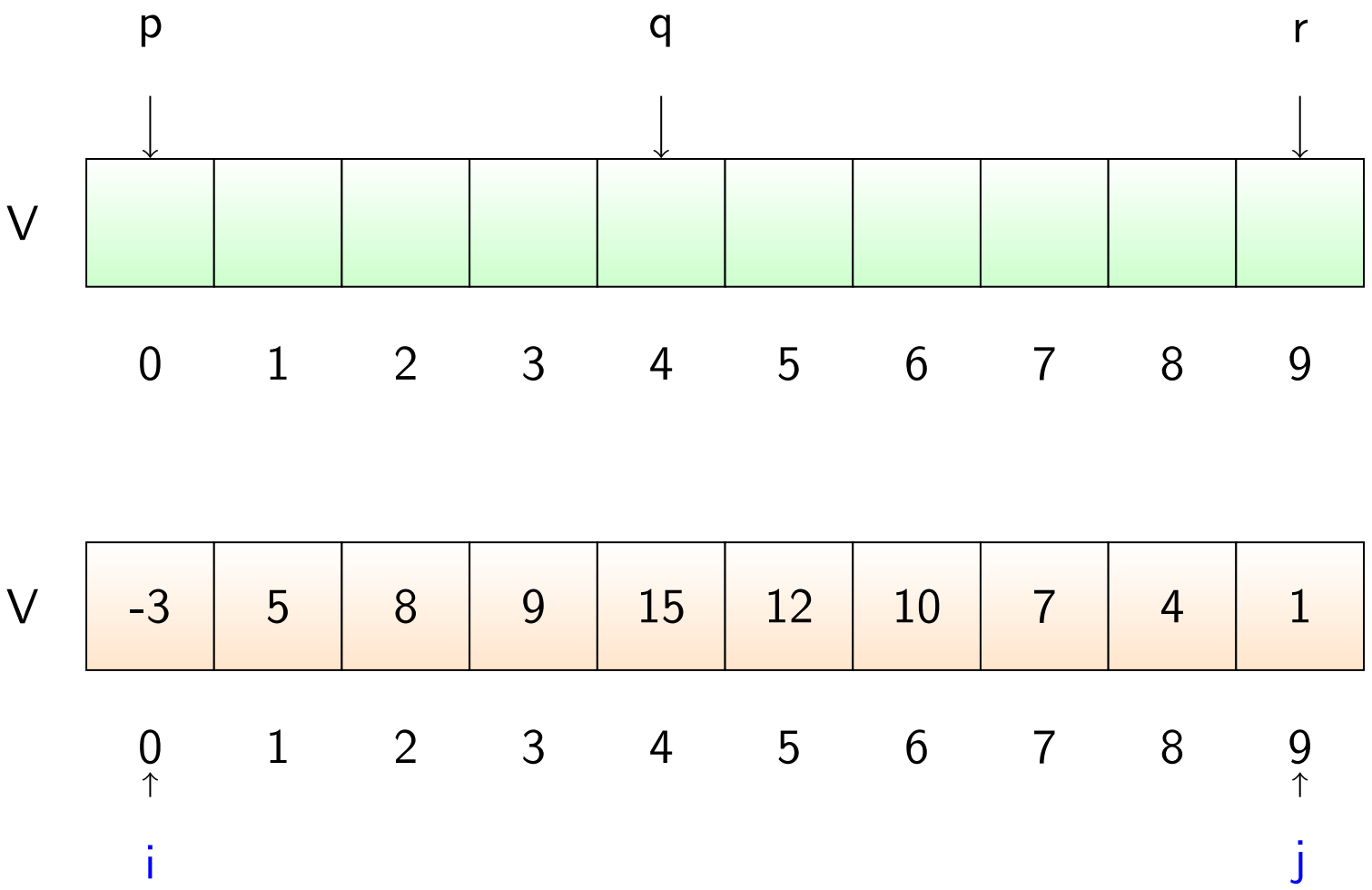
Função de intercalação *(intercala)*



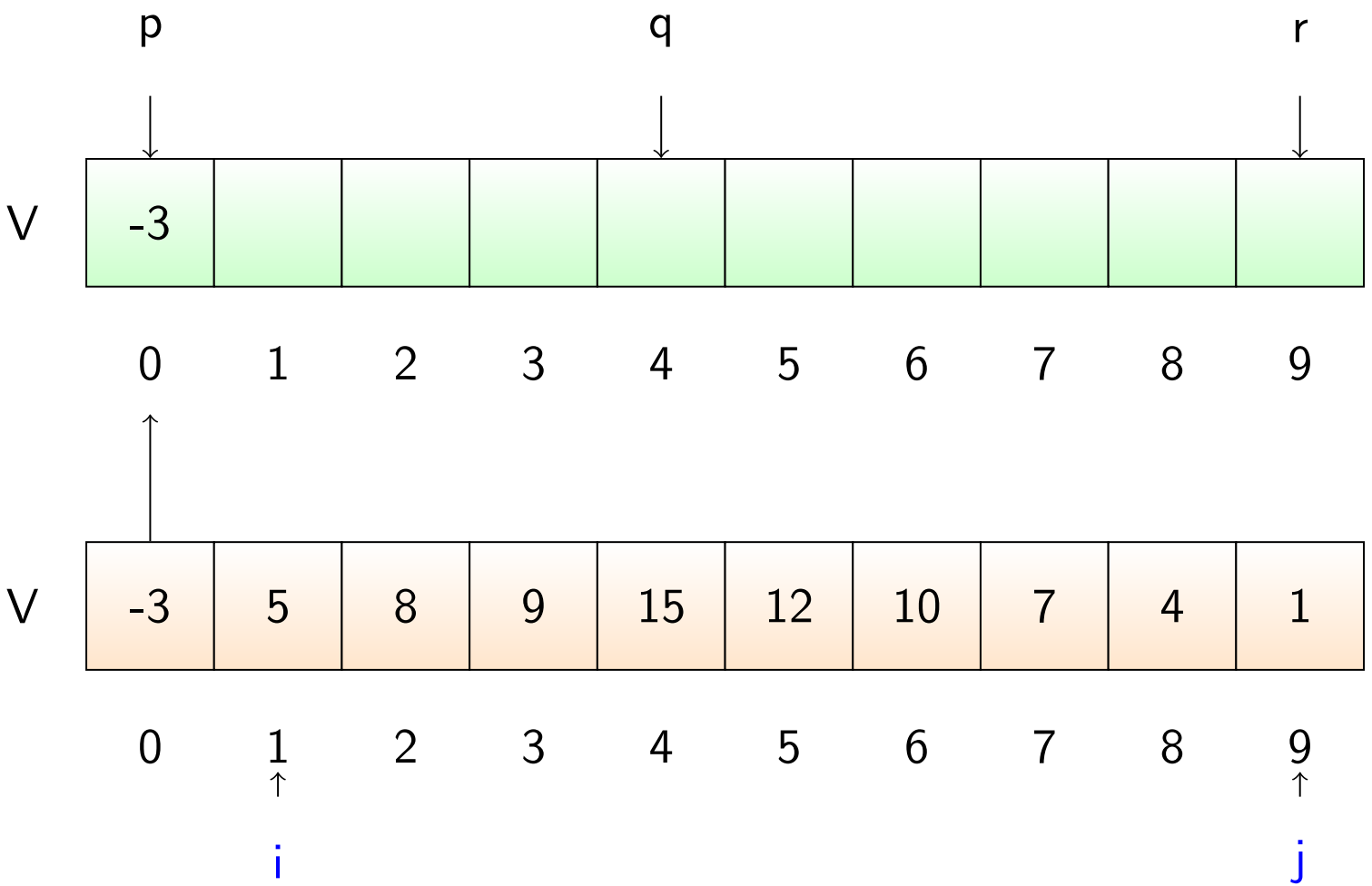
Função de intercalação *(intercala)*



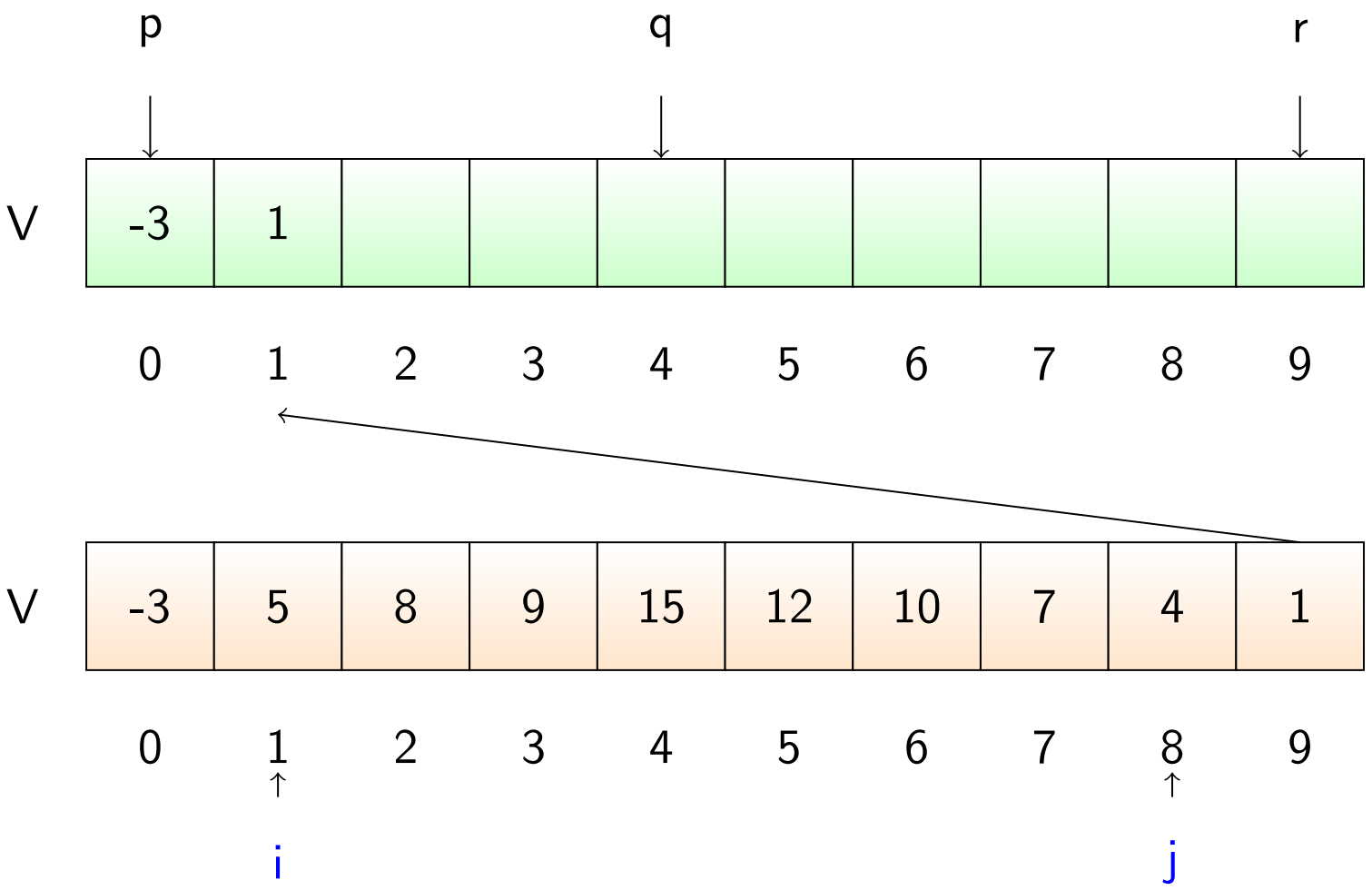
Função de intercalação *(intercala)*



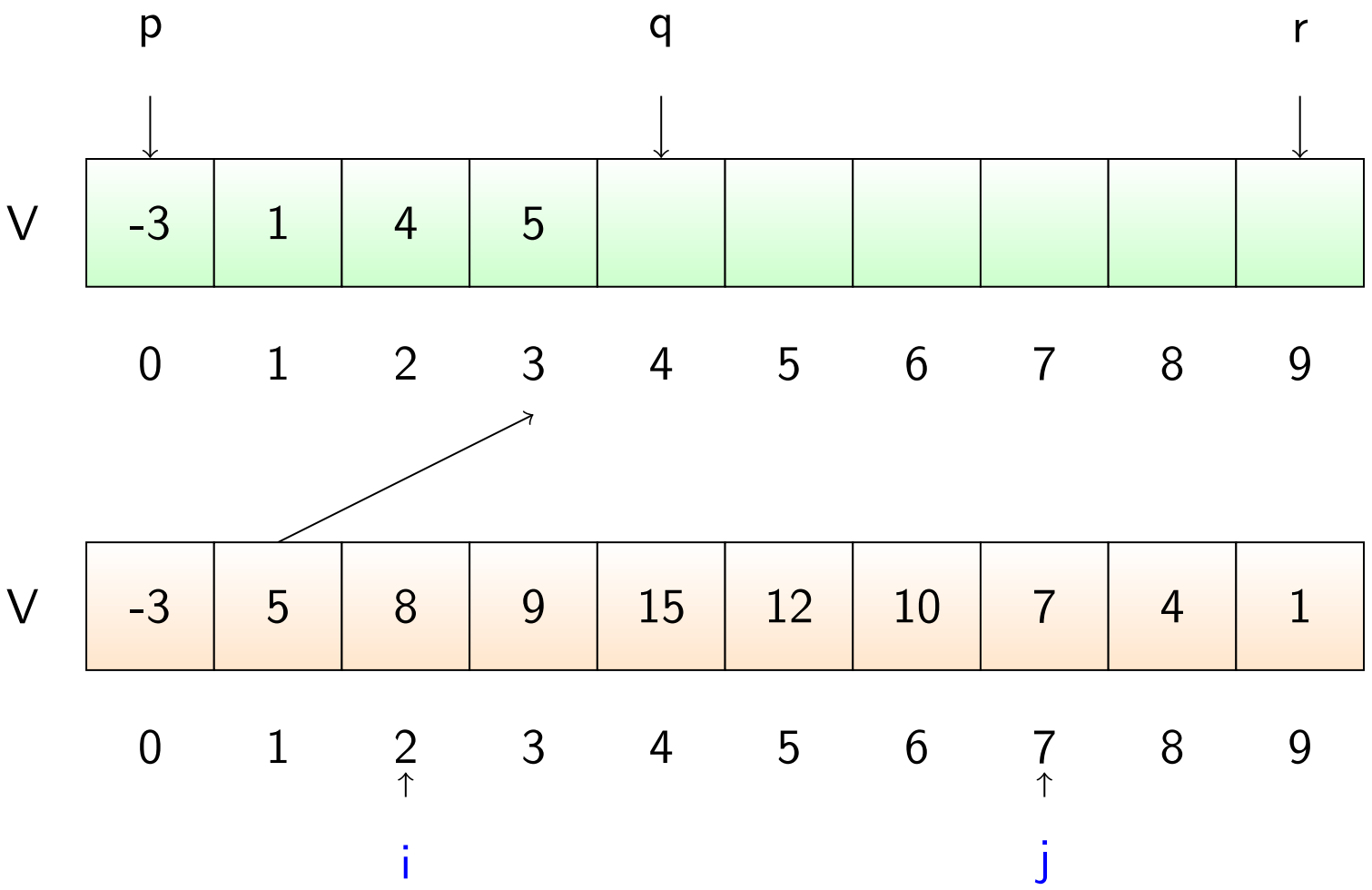
Função de intercalação *(intercala)*



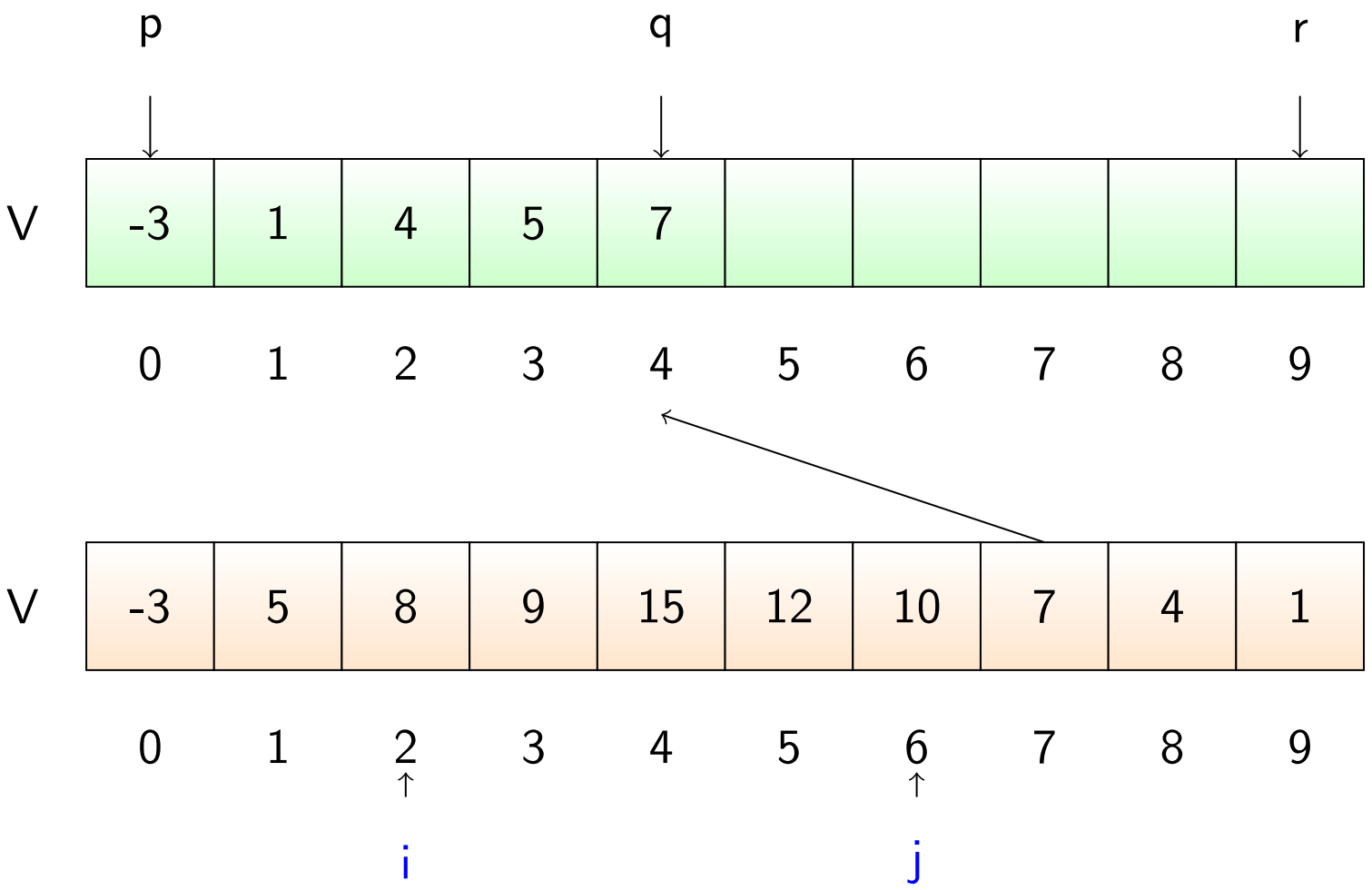
Função de intercalação *(intercala)*



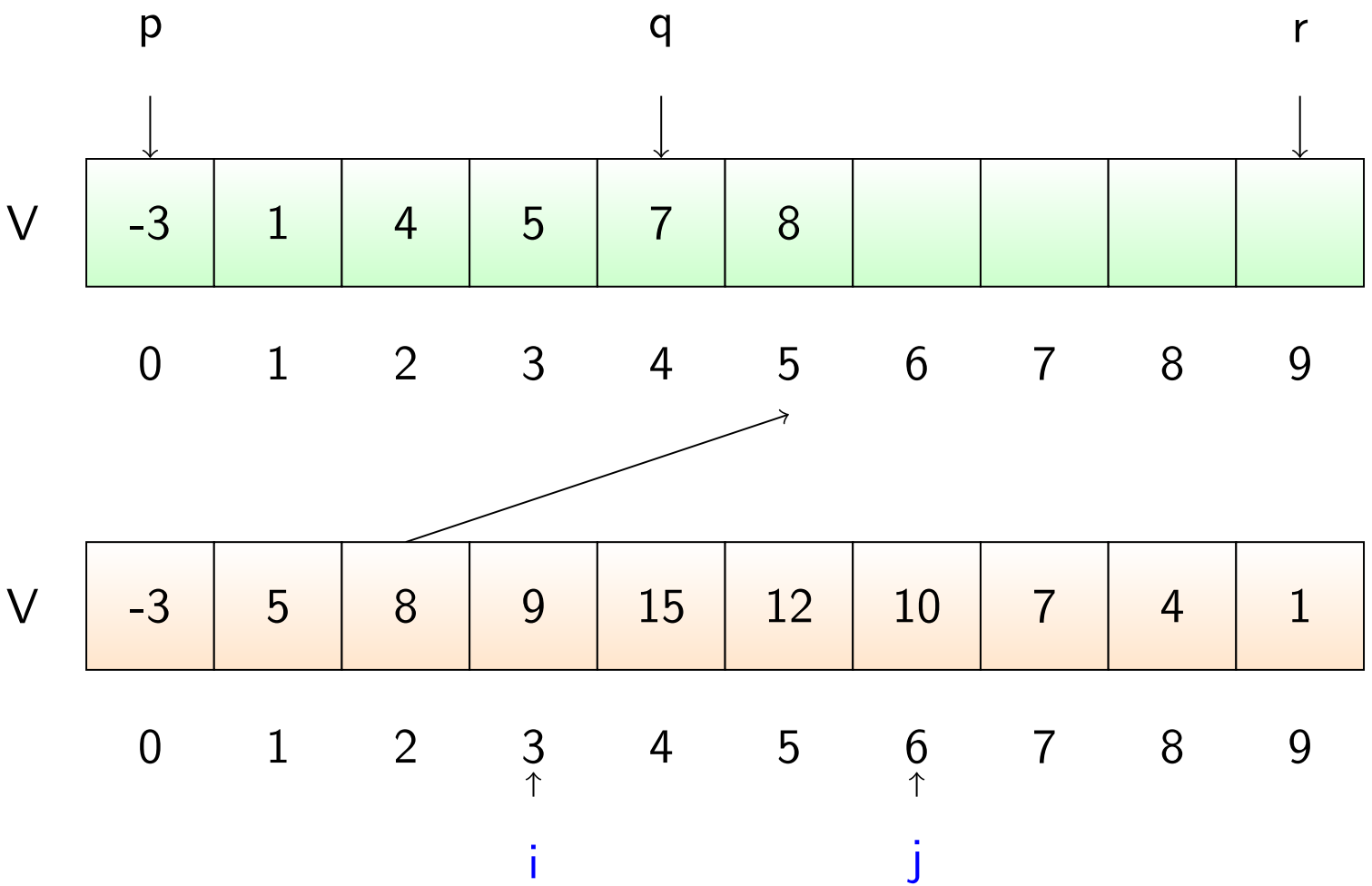
Função de intercalação *(intercala)*



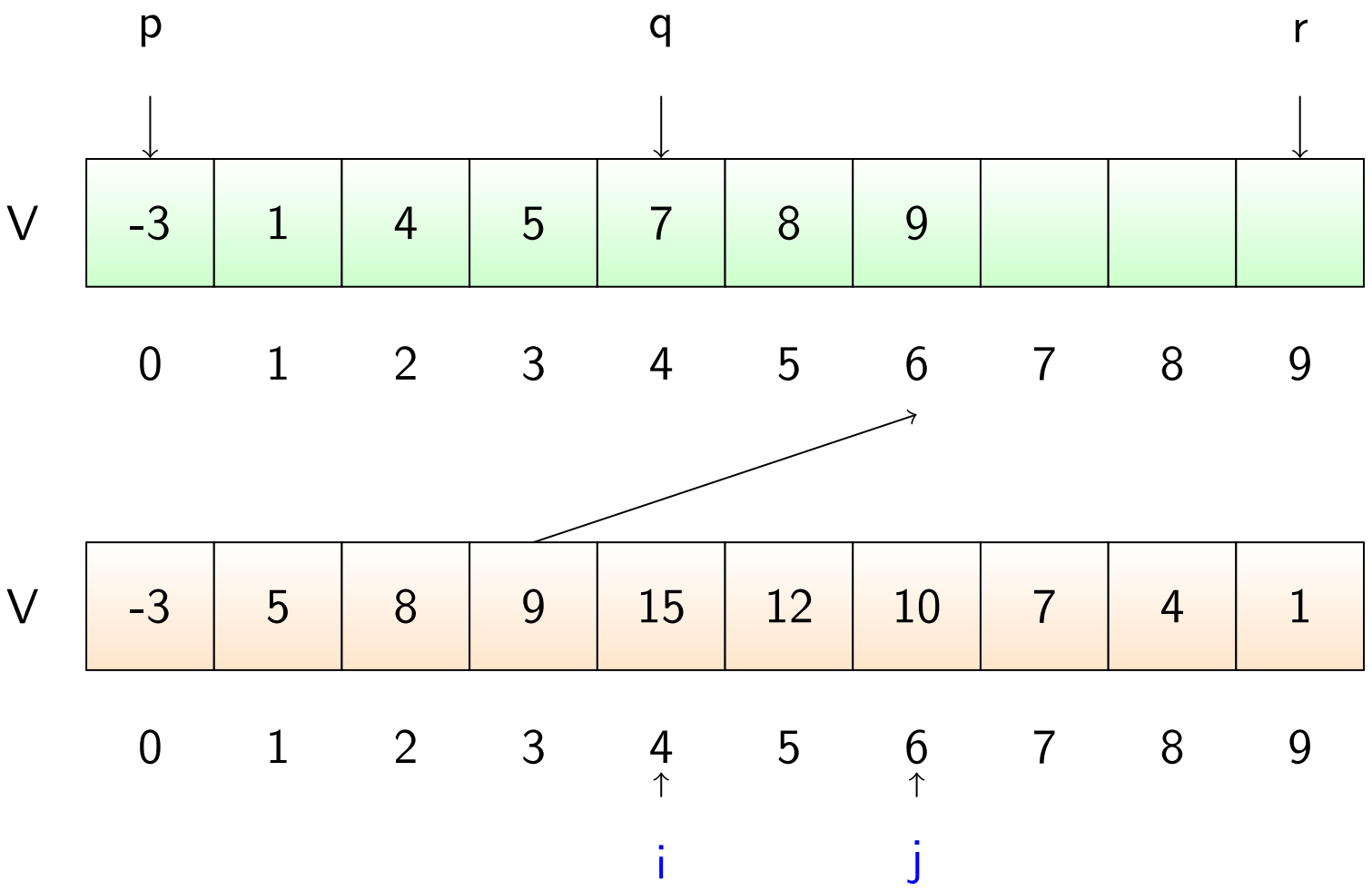
Função de intercalação *(intercala)*



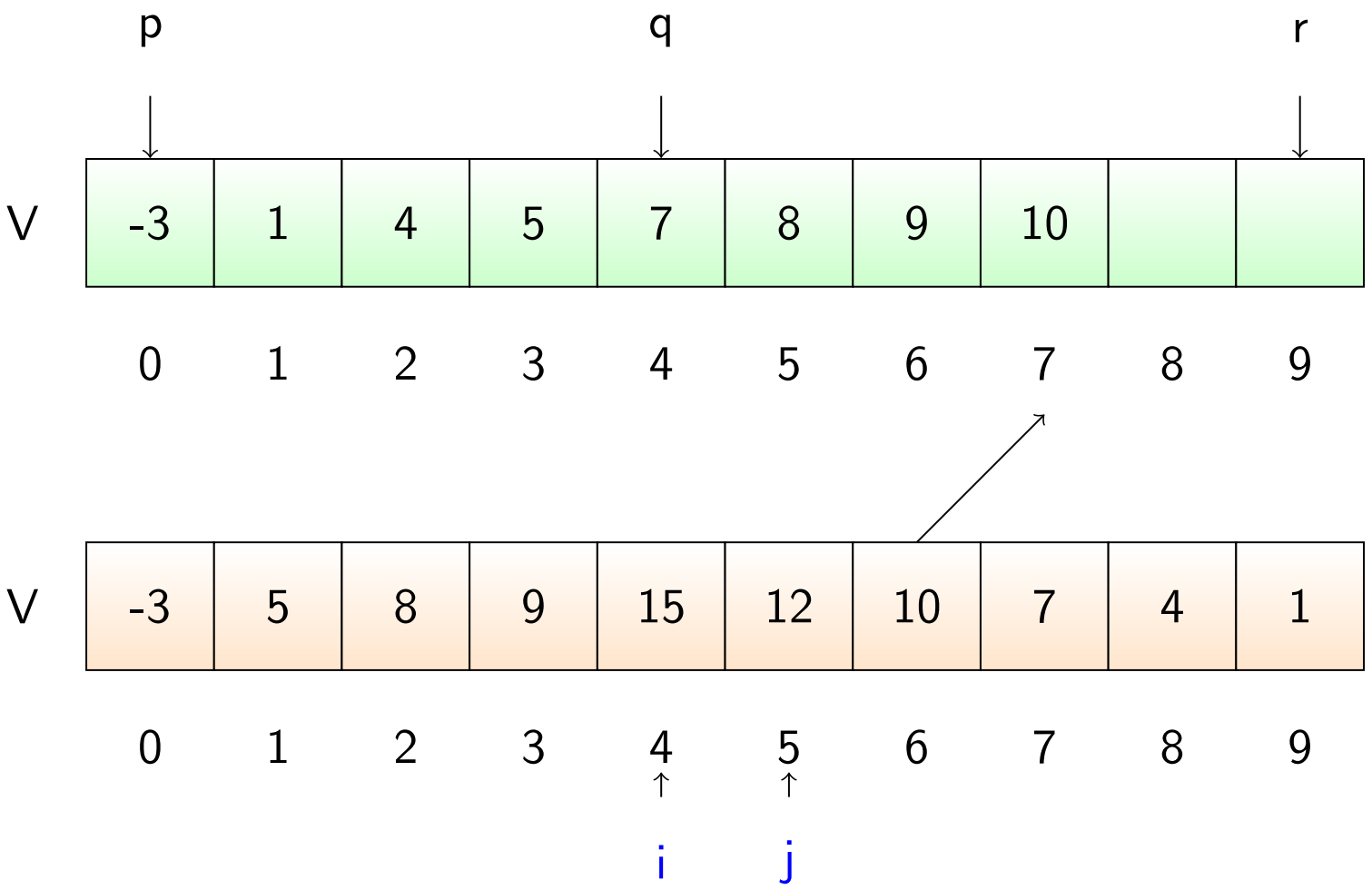
Função de intercalação *(intercala)*



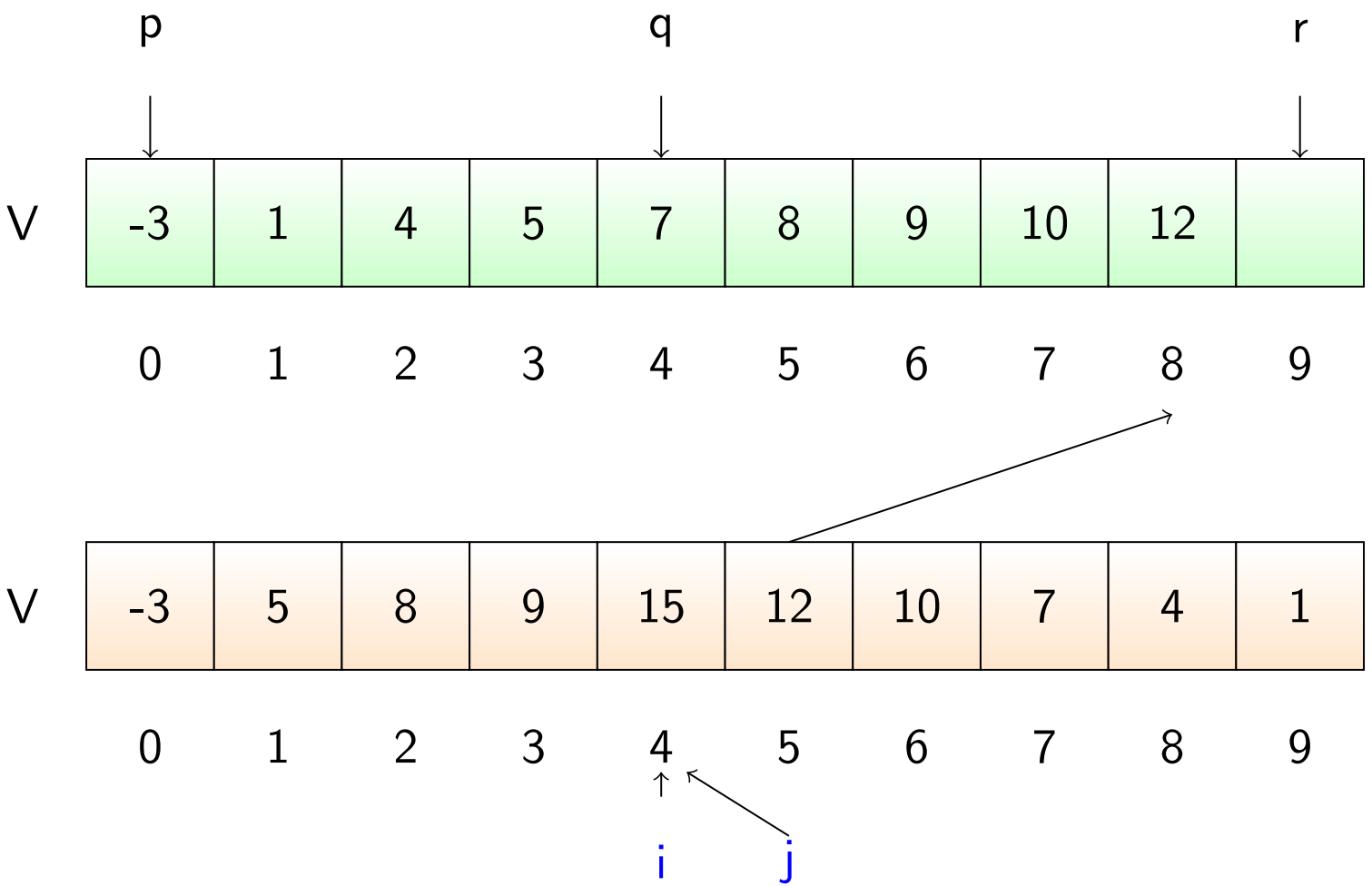
Função de intercalação *(intercala)*



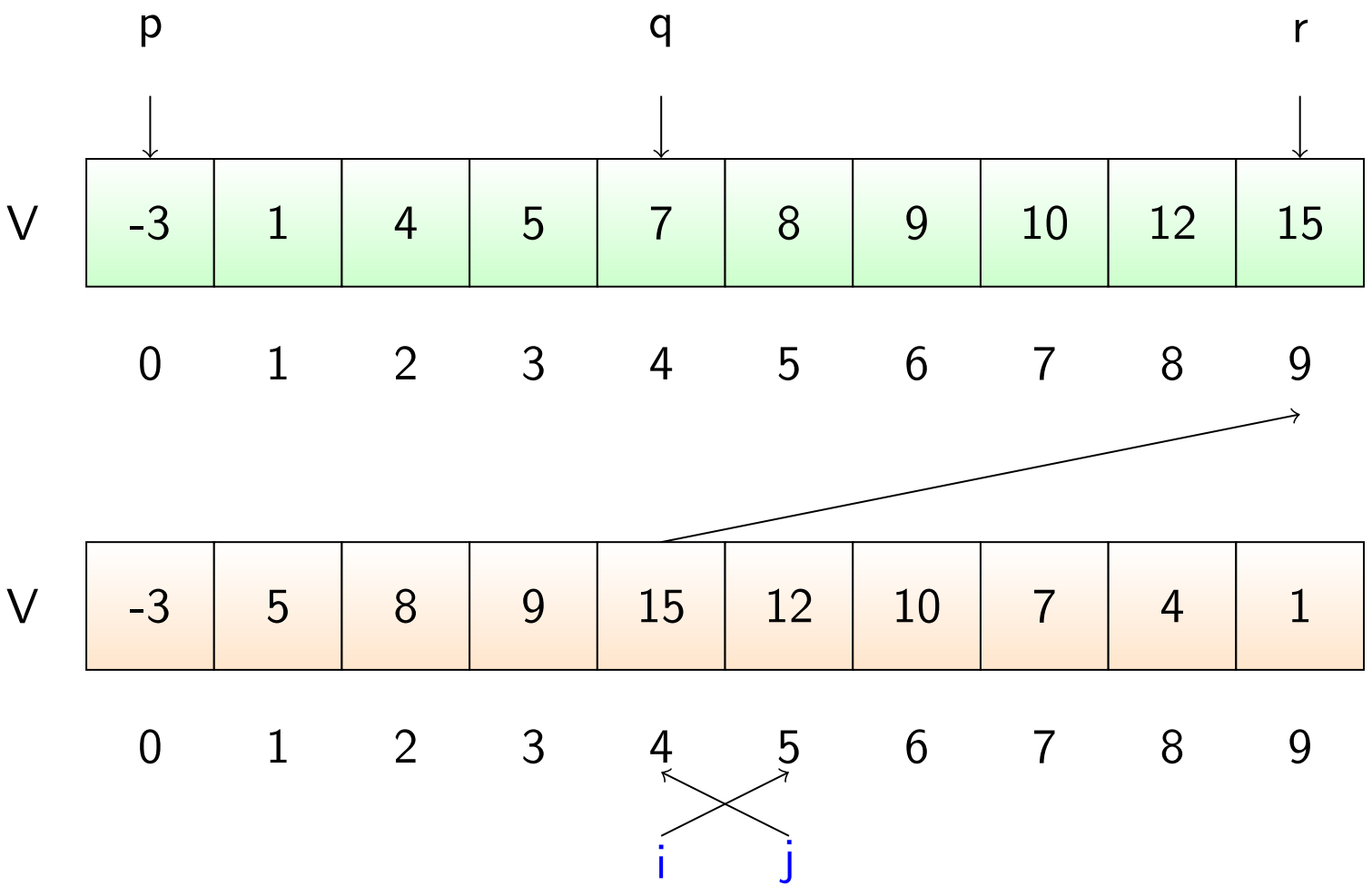
Função de intercalação *(intercala)*



Função de intercalação *(intercala)*



Função de intercalação *(intercala)*



Mergesort

Onde está a mágica do *Mergesort*?

```
void mergesort (int* v, int p, int r){
   if (p < r-1) {
      int q = (p + r)/2;
      mergesort (v, p, q);
      mergesort (v, q+1, r);
      intercala (v, p, q, r);
   }
}
```

Mergesort

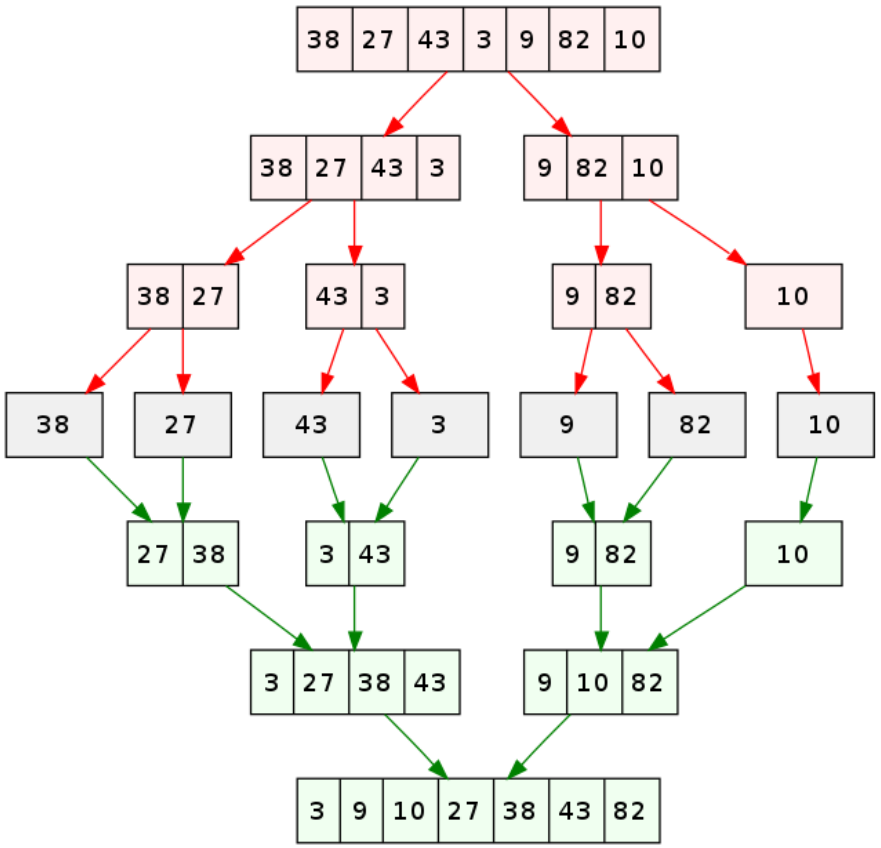


Figura: Chamadas recursivas do Mergesort (extraído de `https://www.wikiwand.com/en/Merge_sort`)

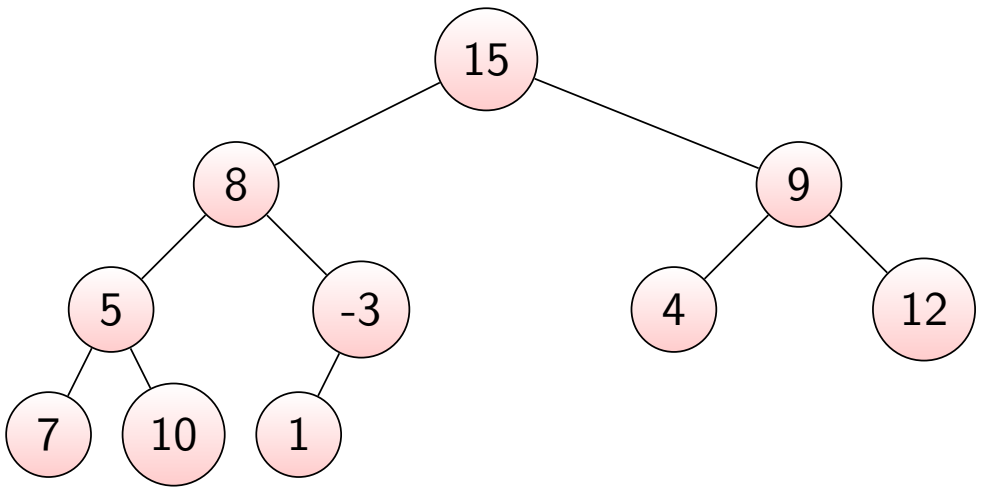
Heapsort

- O heapsort utiliza uma estrutura de dados chamada *heap*.
- A *heap* é um vetor com configuração similar a uma árvore binária.
- Considere um vetor $V[0..n-1]$ de n elementos.
  - Raiz = V[0]
  - Pai de V[i] = V$[\lfloor (i-1)/2 \rfloor]$, para $1 \leq i \leq n-1$.
  - Filho esquerdo de V[i] = V$[2i+1]$, para $i \leq (n-2)/2$.
  - Filho direito de V[i] = V$[2i+2]$, para $2 \leq (n-3)/2$.

Heapsort

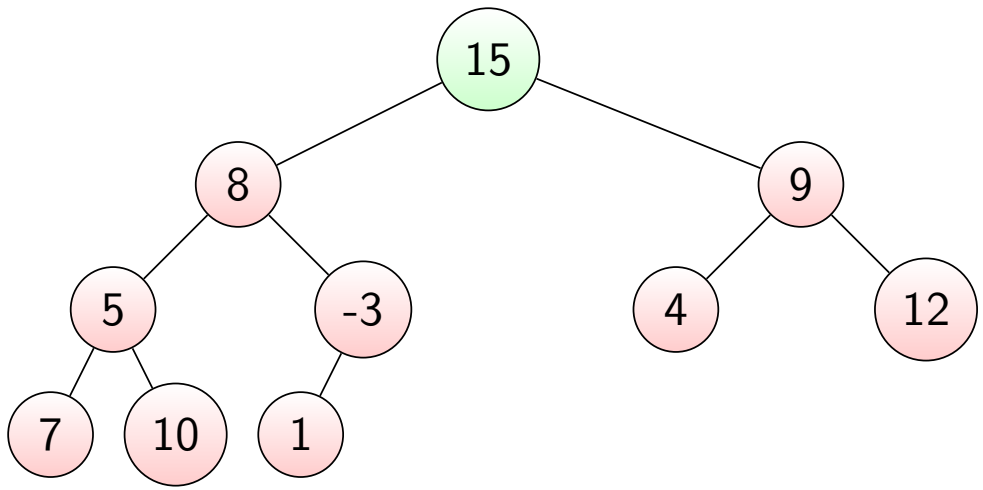
- Propriedade da heap:
  - $V[i] \geq V[2i+1]$ e $V[i] \geq V[2i+2]$.
  - O valor contido no índice pai é maior que o contido em seus filhos.
- Podemos dividir o heapsort em duas fases:
  - Fase 1: construção da heap.
  - Fase 2: ordenação.

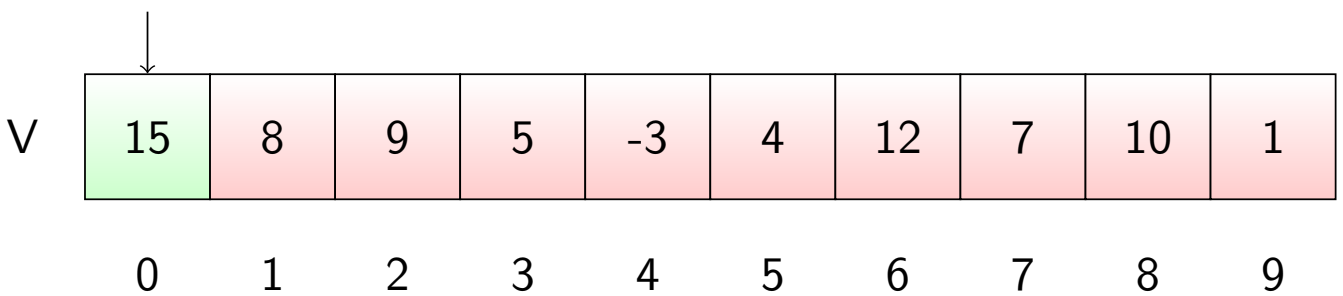
Heapsort *(Fase 1: construindo a heap)*



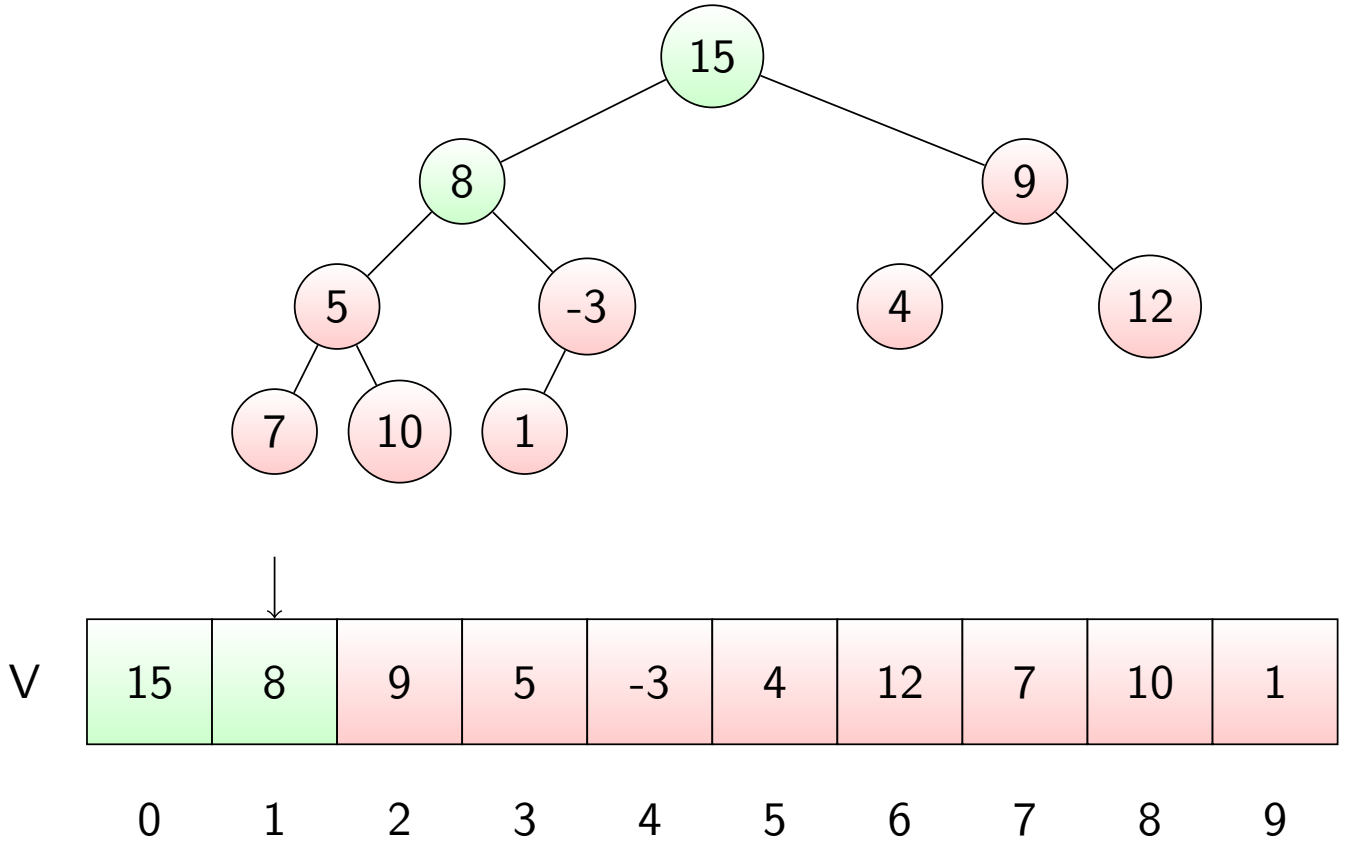
V	15	8	9	5	-3	4	12	7	10	1
	0	1	2	3	4	5	6	7	8	9

Heapsort *(Fase 1: construindo a heap)*

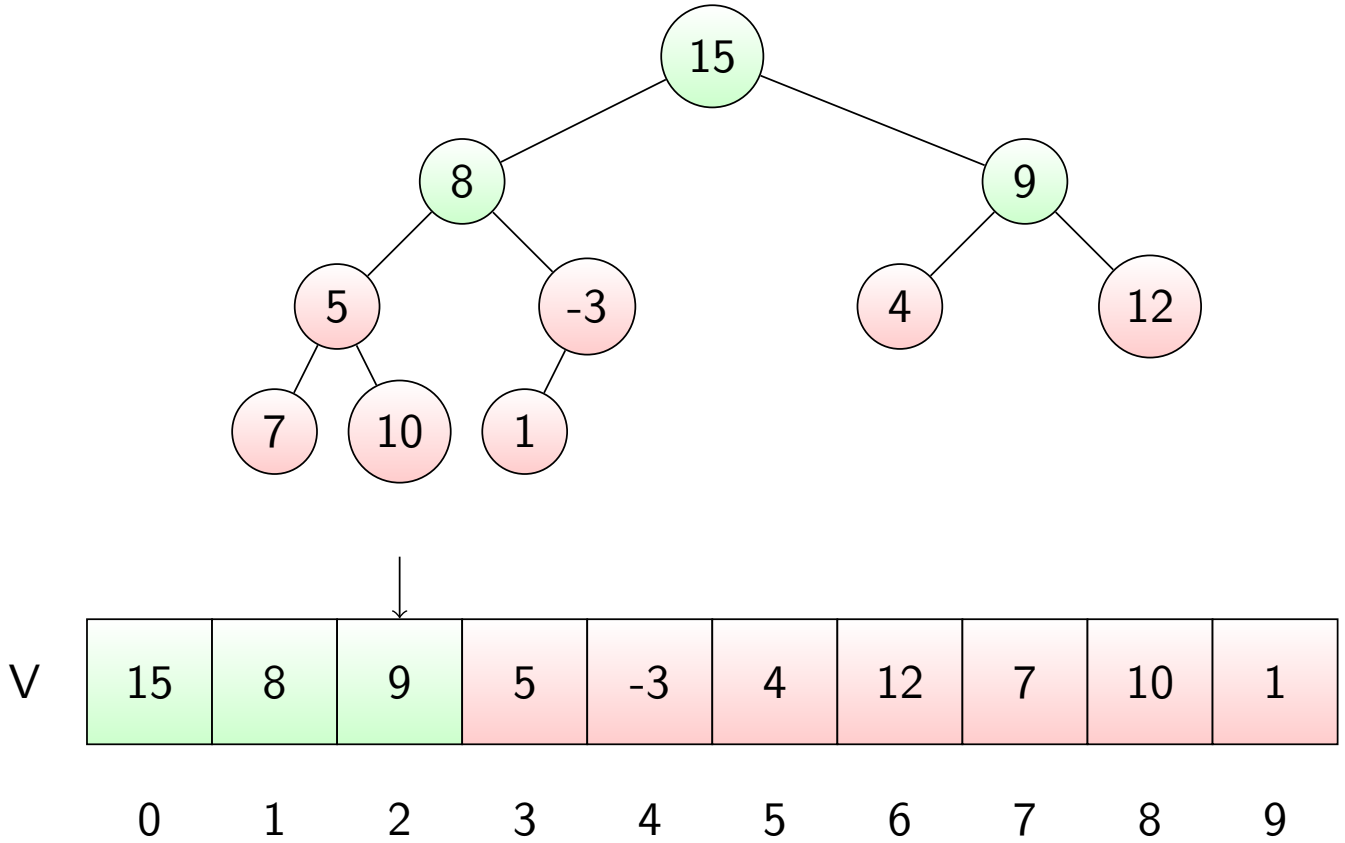




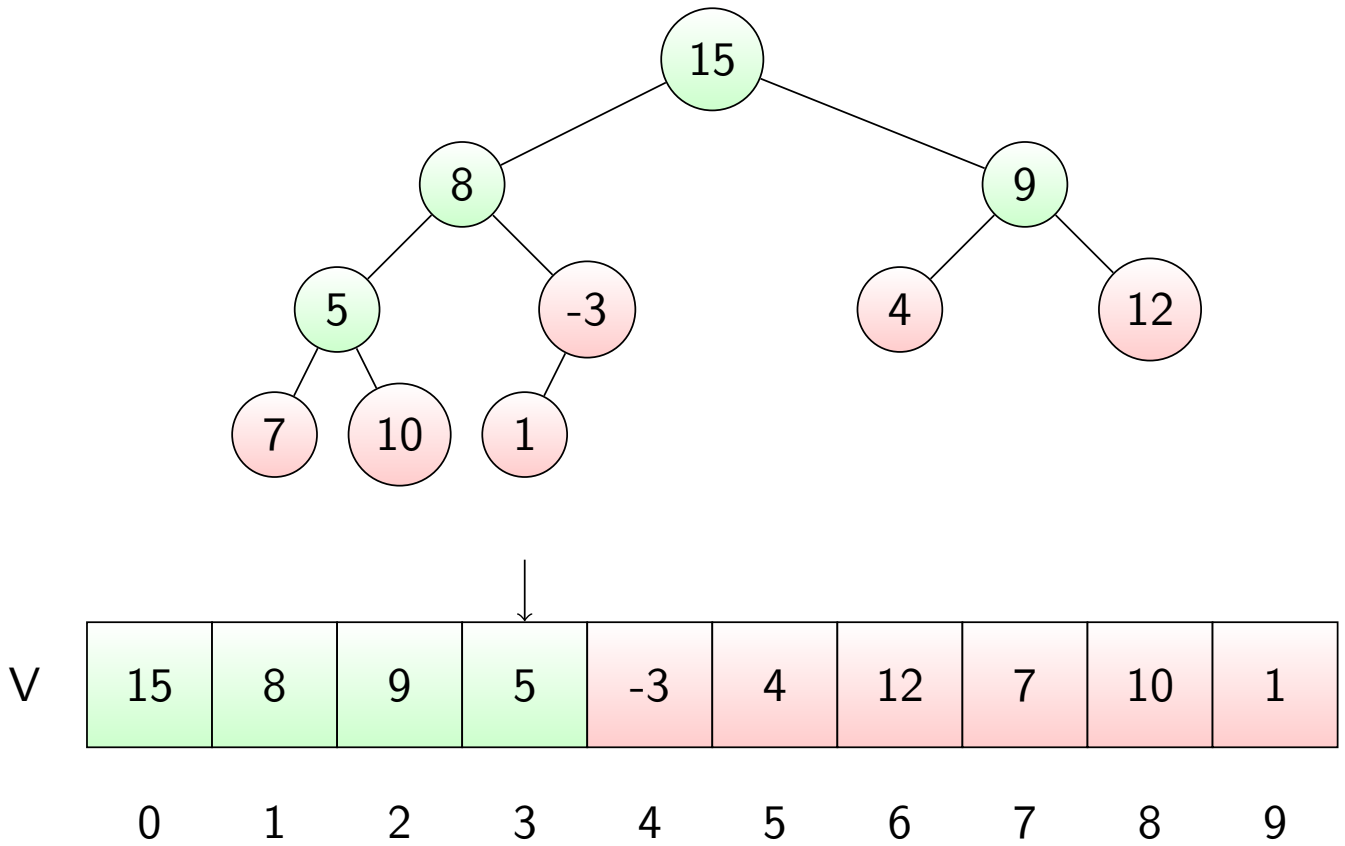
Heapsort *(Fase 1: construindo a heap)*



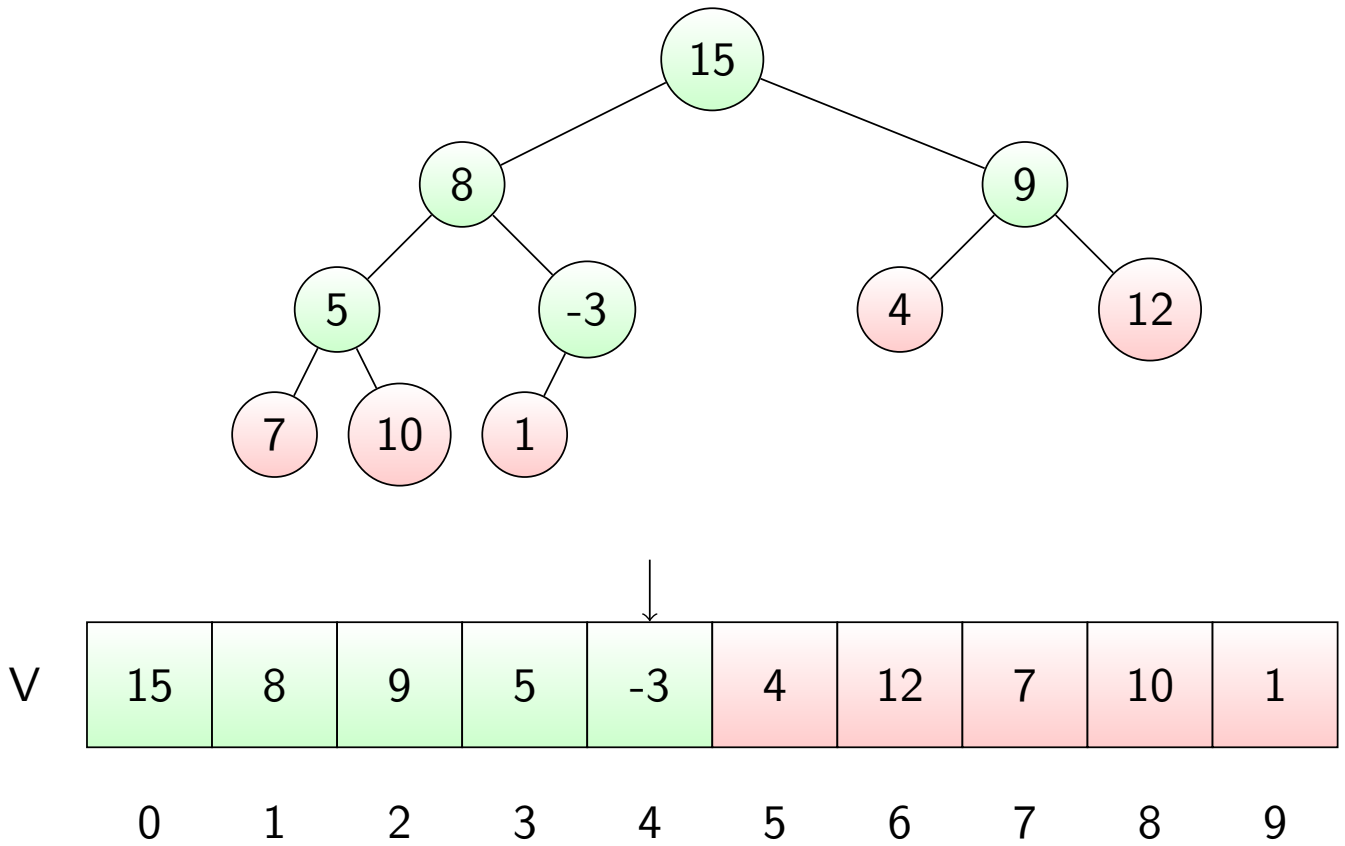
Heapsort *(Fase 1: construindo a heap)*



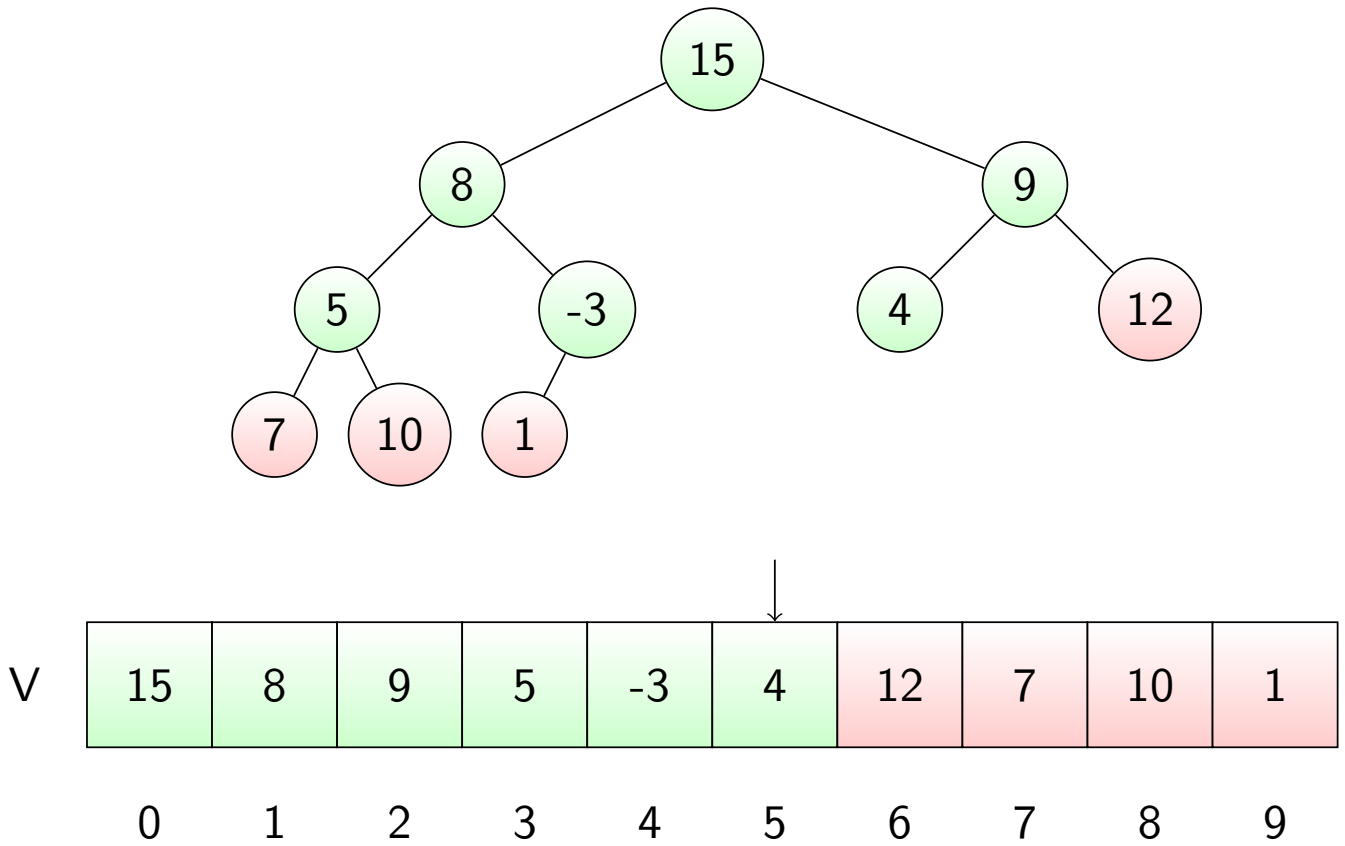
Heapsort *(Fase 1: construindo a heap)*



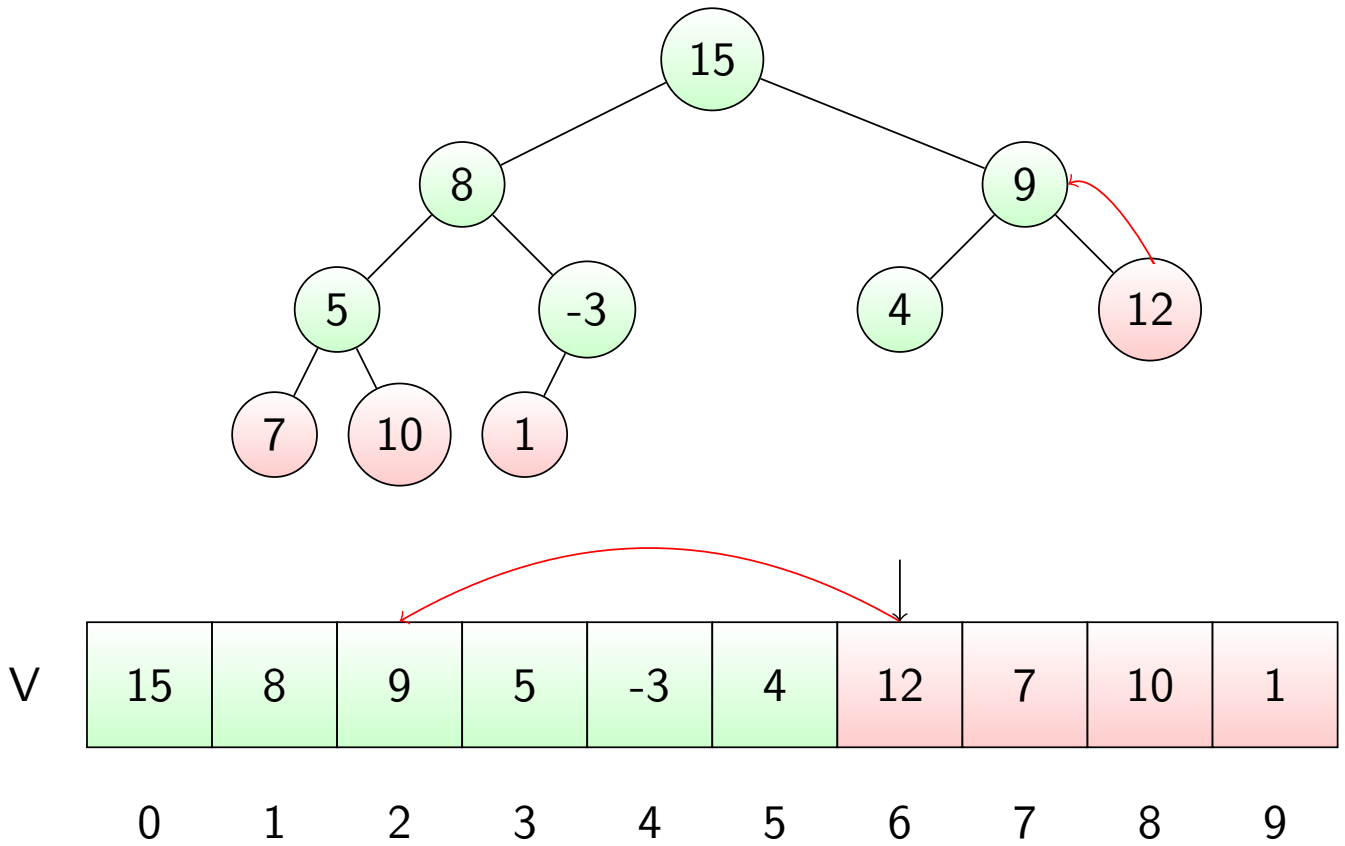
Heapsort *(Fase 1: construindo a heap)*



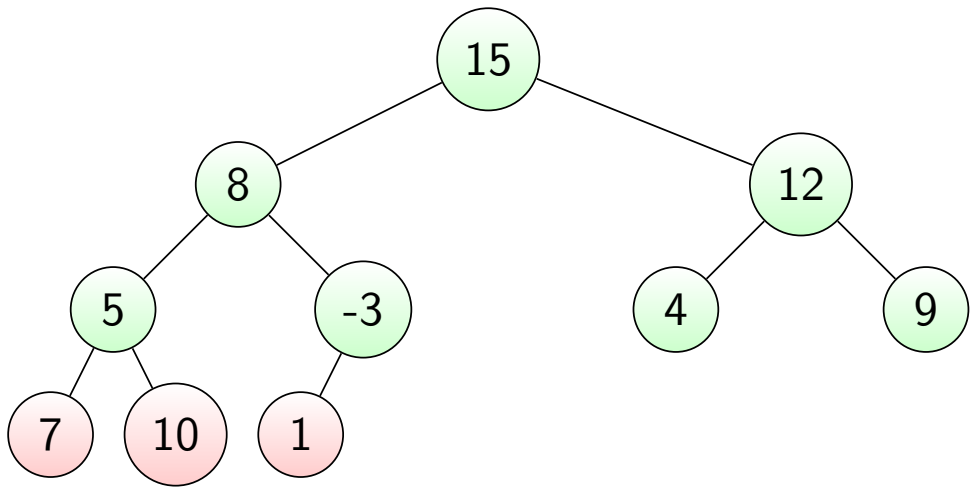
Heapsort *(Fase 1: construindo a heap)*

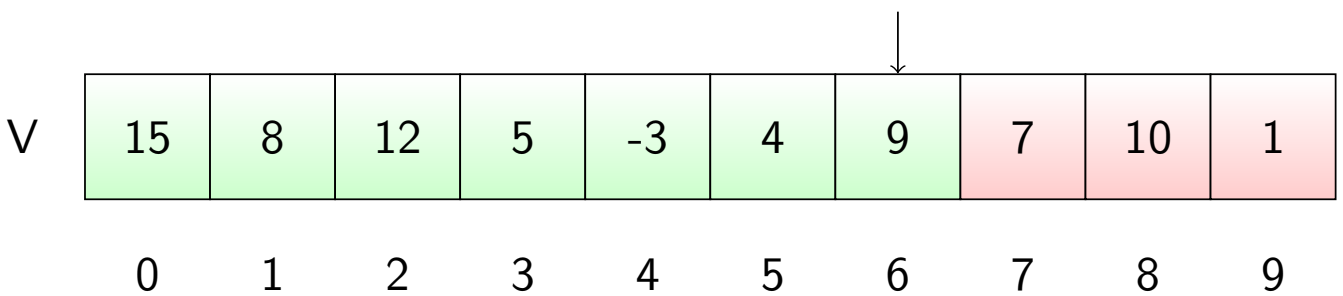


Heapsort *(Fase 1: construindo a heap)*

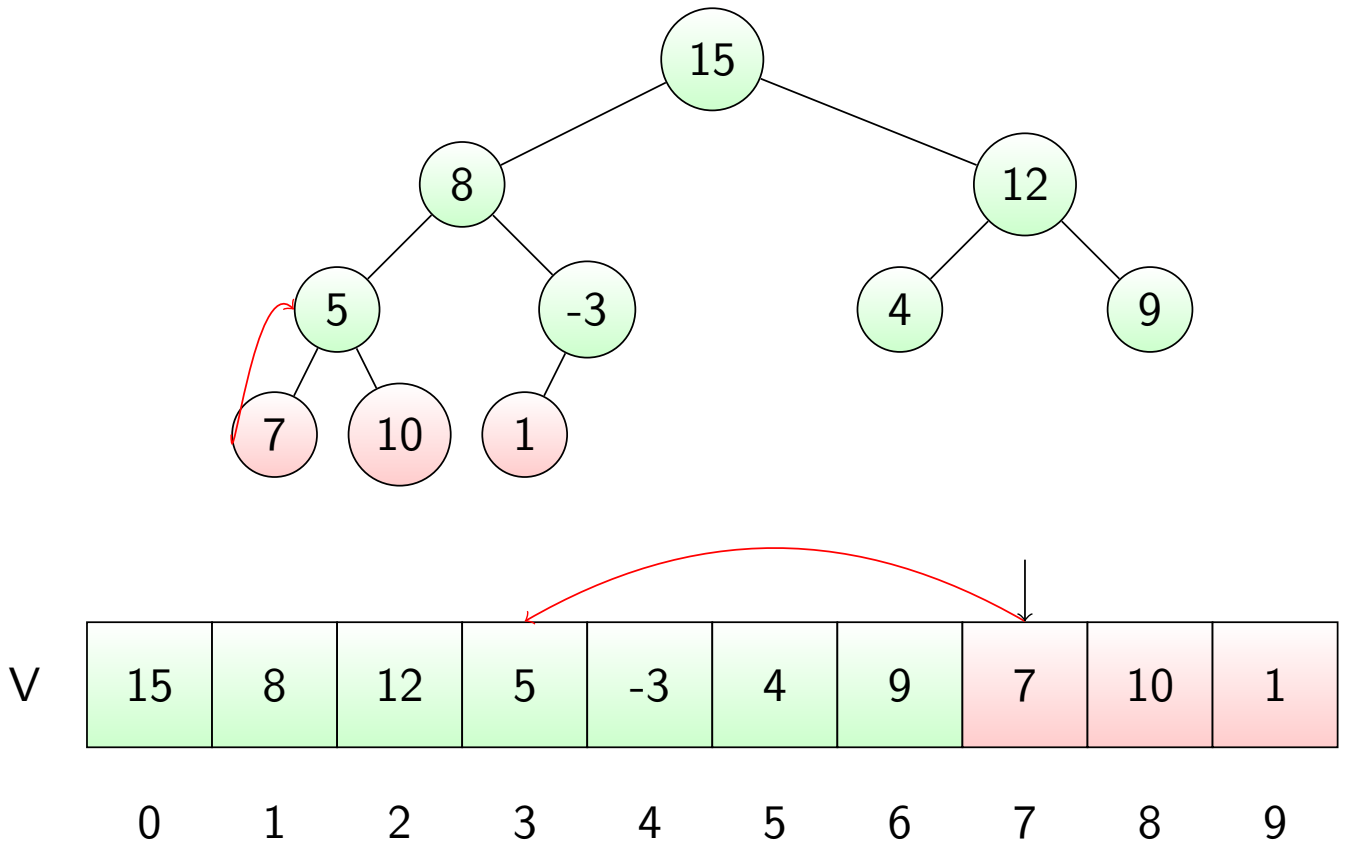


Heapsort *(Fase 1: construindo a heap)*

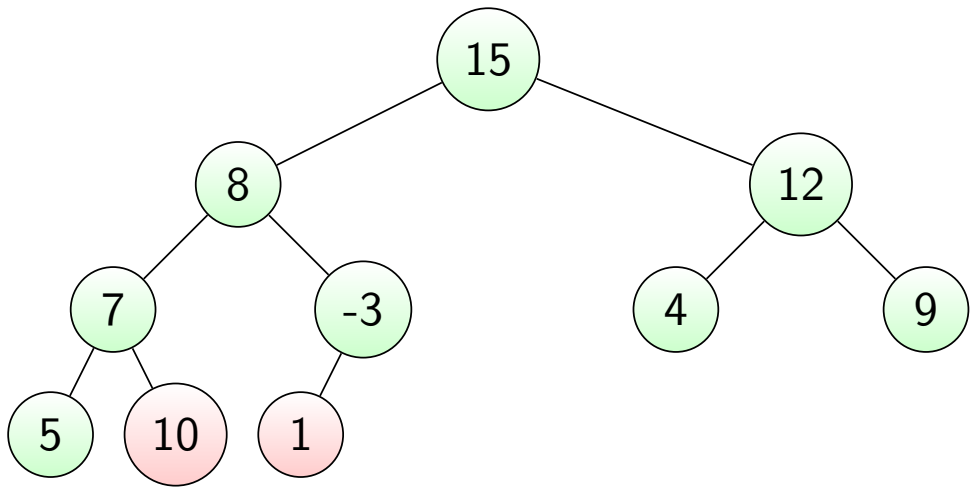


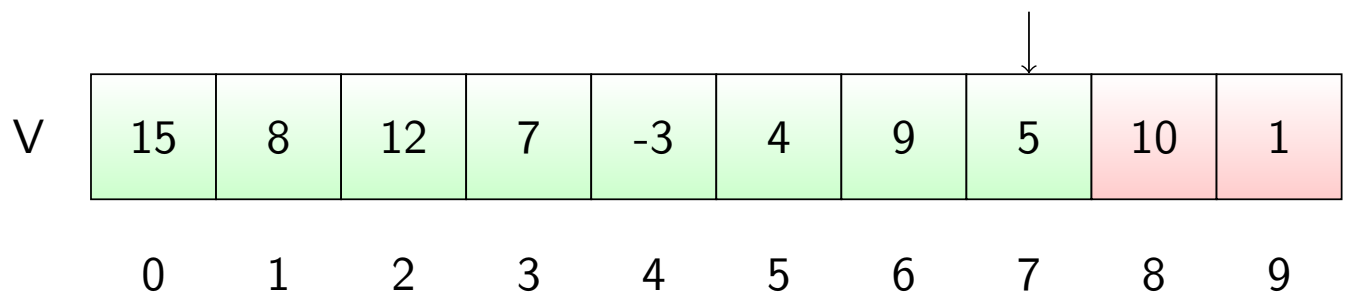


Heapsort *(Fase 1: construindo a heap)*

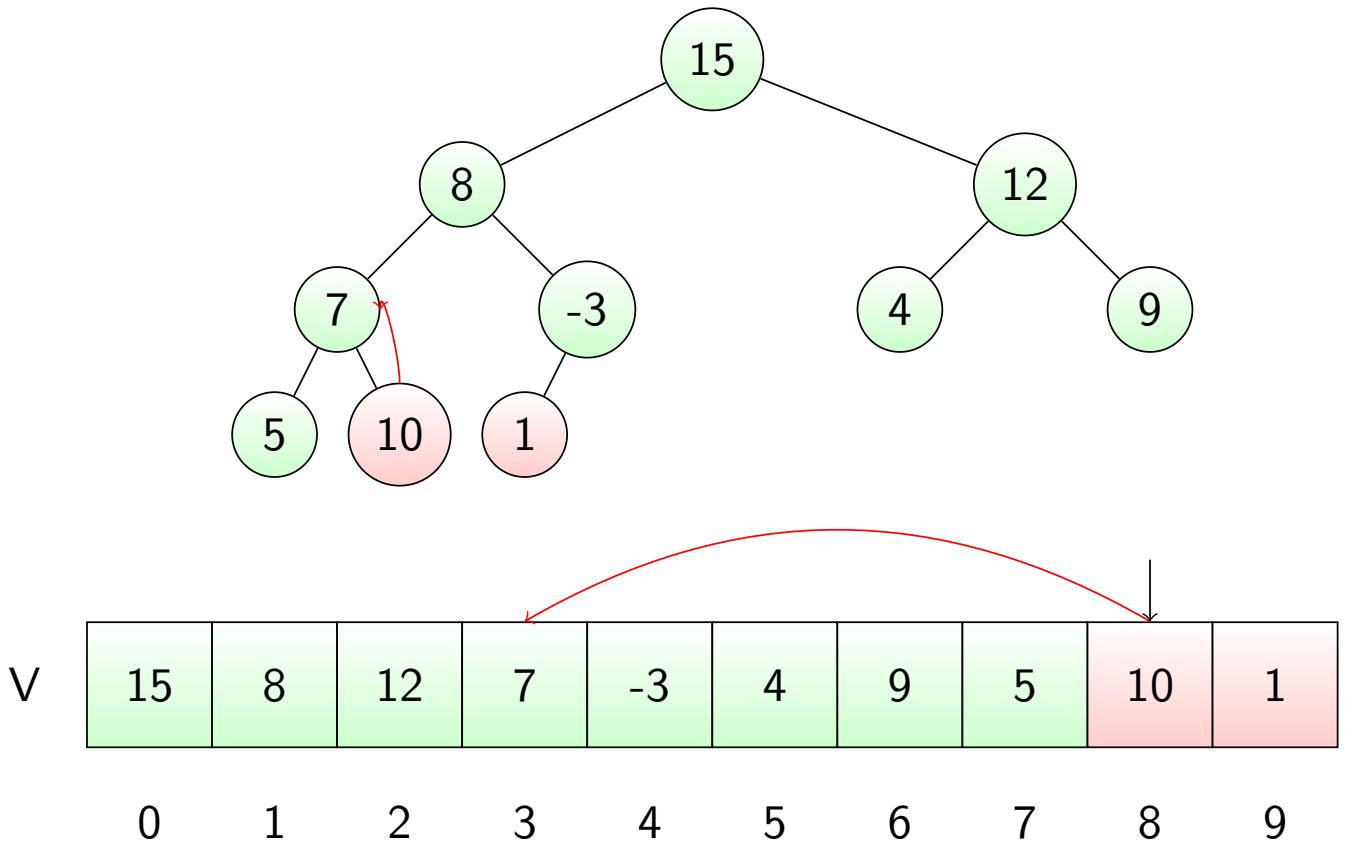


Heapsort *(Fase 1: construindo a heap)*

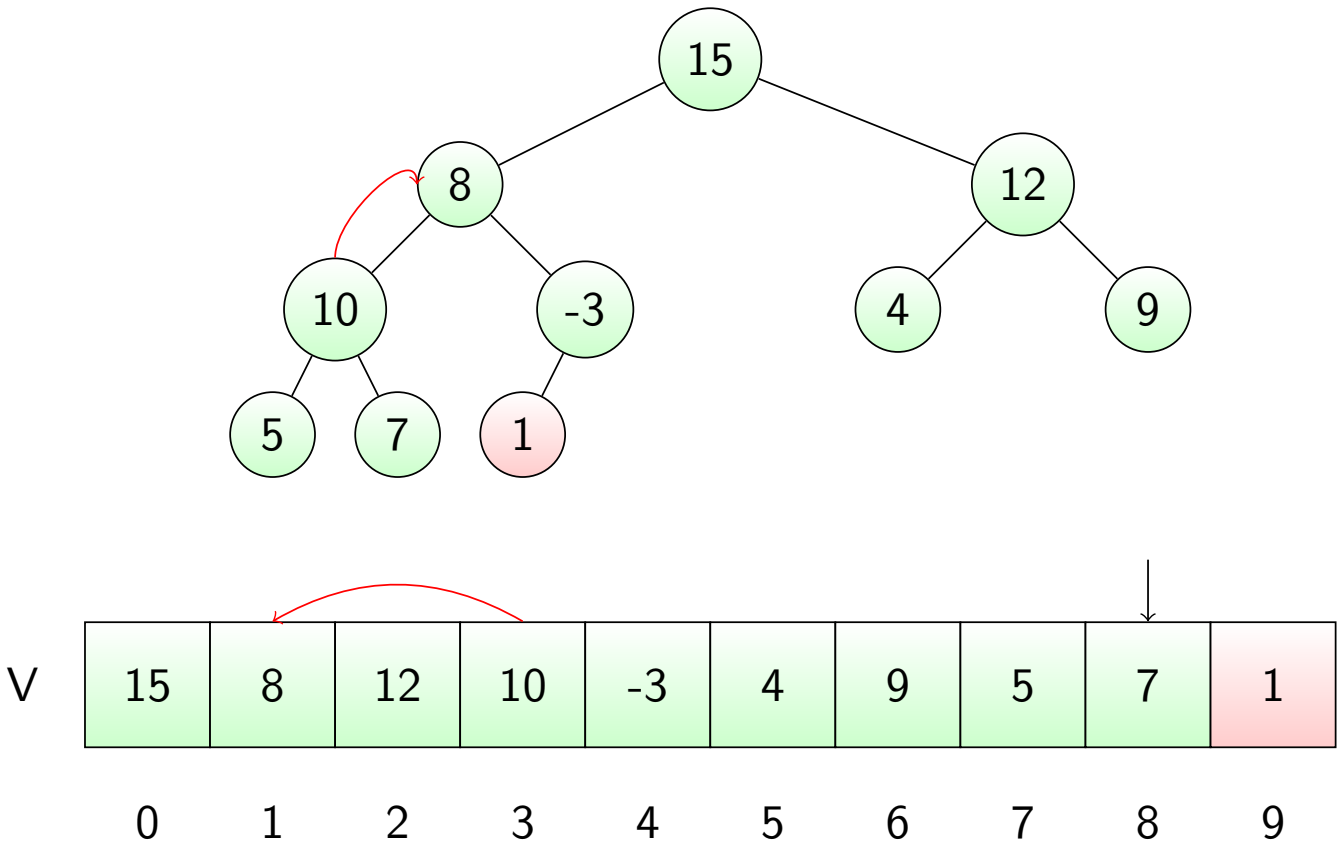




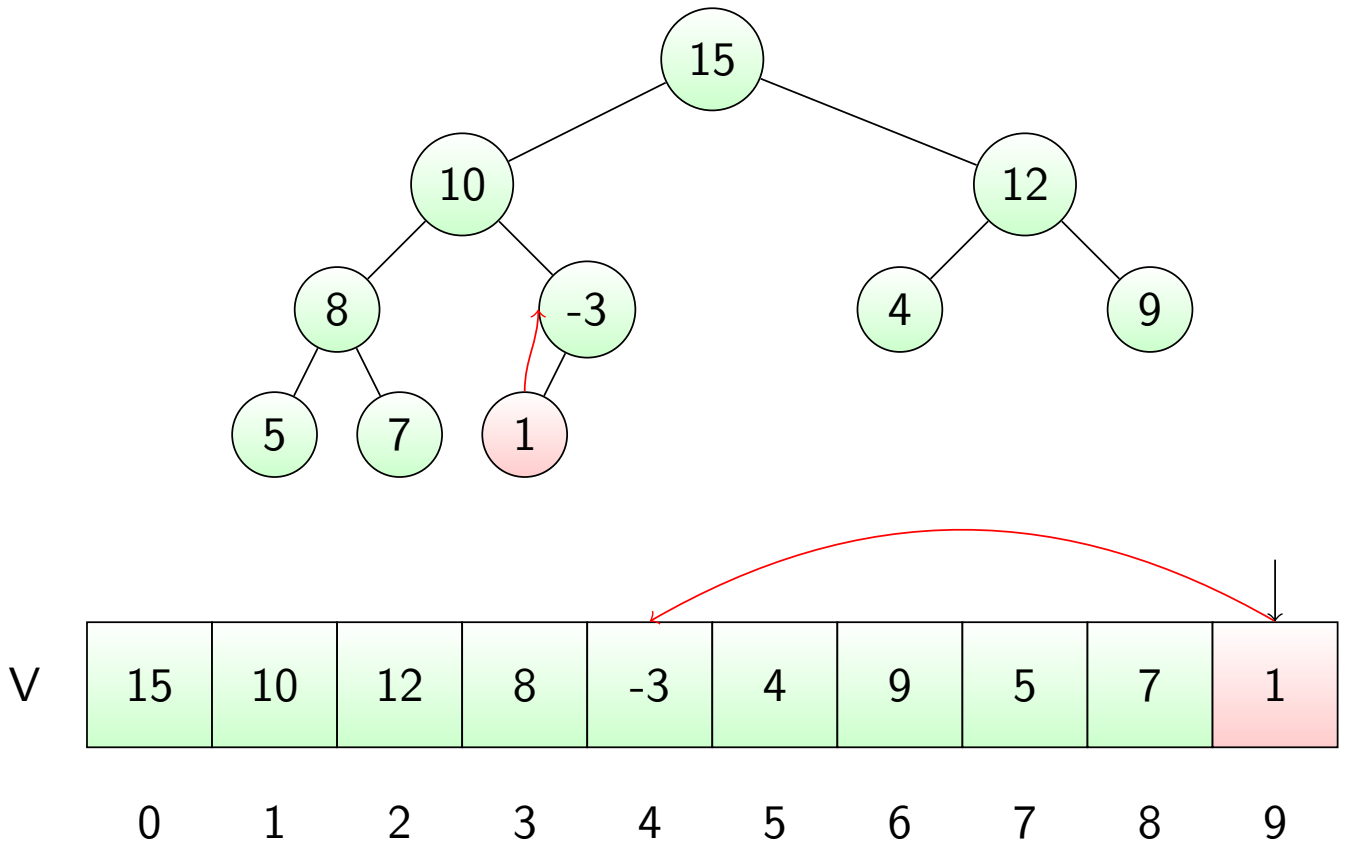
Heapsort *(Fase 1: construindo a heap)*



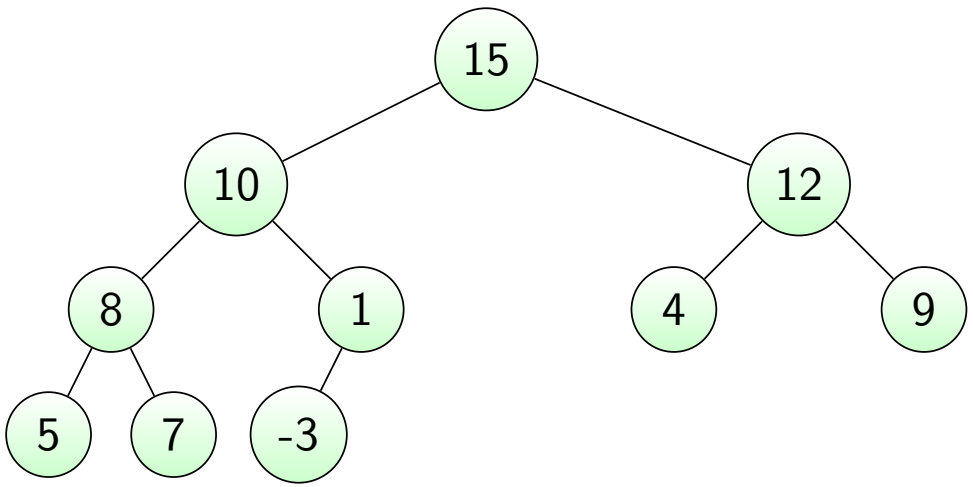
Heapsort *(Fase 1: construindo a heap)*

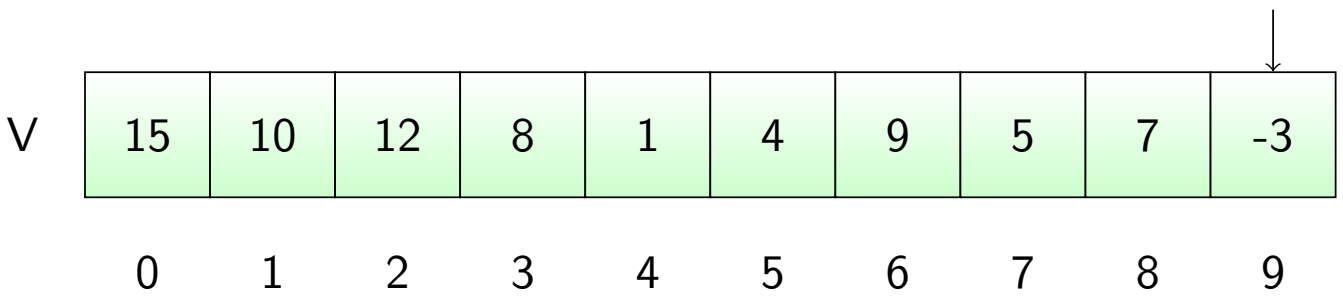


Heapsort *(Fase 1: construindo a heap)*



Heapsort *(Fase 1: construindo a heap)*

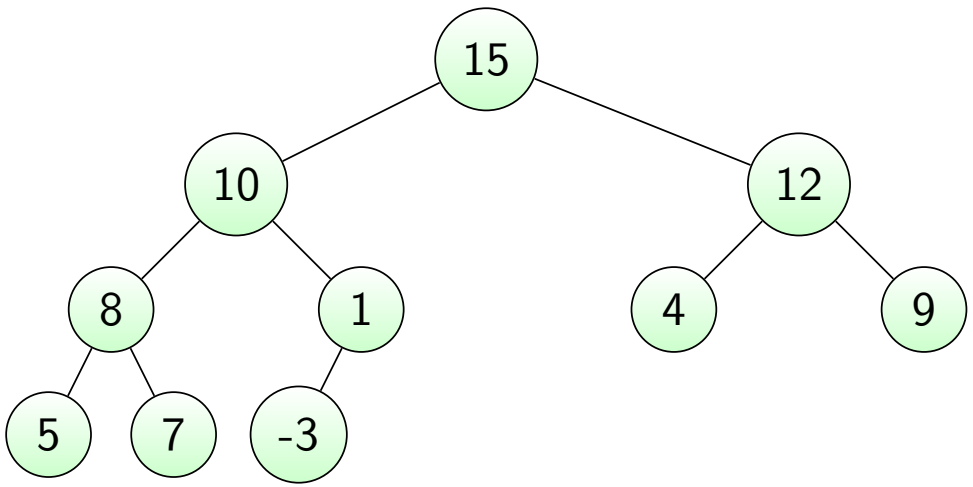


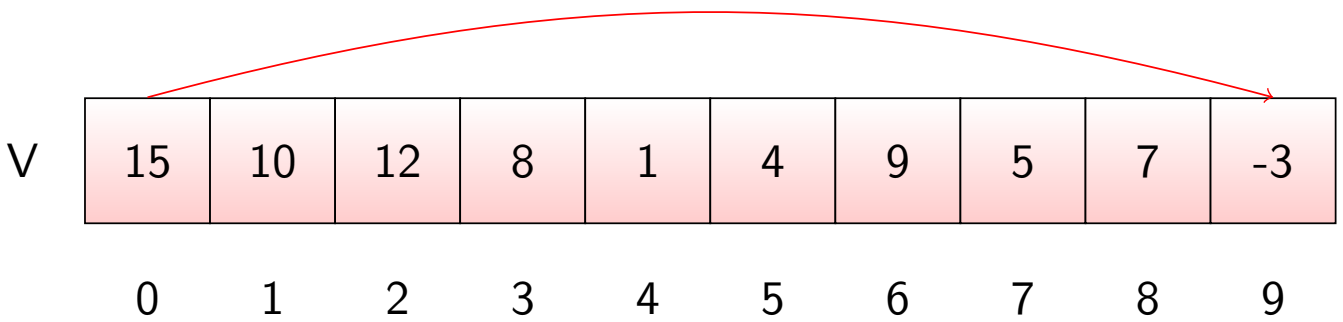


Construindo a Heap

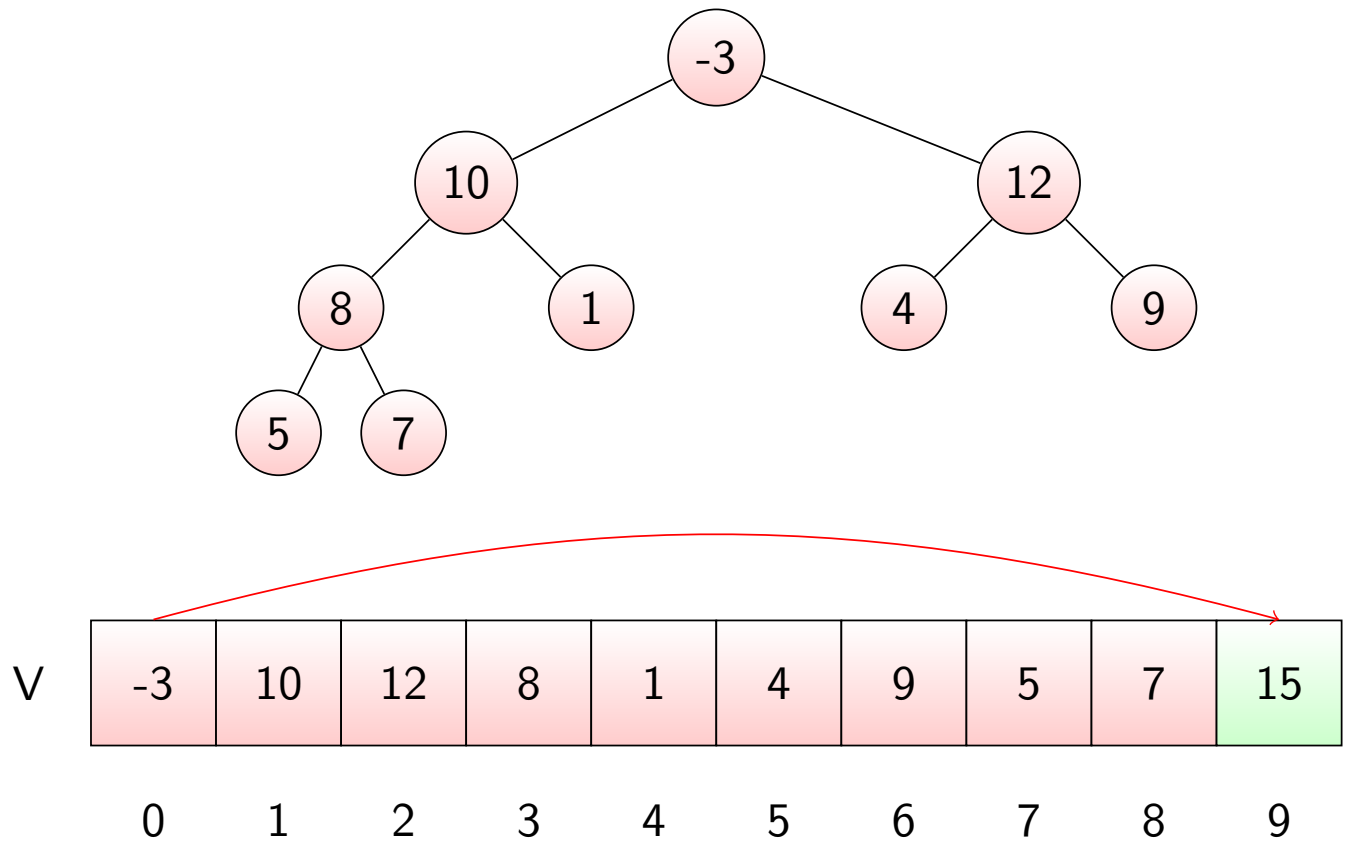
```
void constroiHeap (int* v, int tam){
    int f;
    for (int m=1; m < tam-1; m++){
        f=m+1;
        while (f>0 && v[(f-1)/2] < v[f]){
            troca (v, f, (f-1)/2);
            f = (f-1)/2;
        }
    }
}
```

Heapsort *(Fase 2: ordenação)*

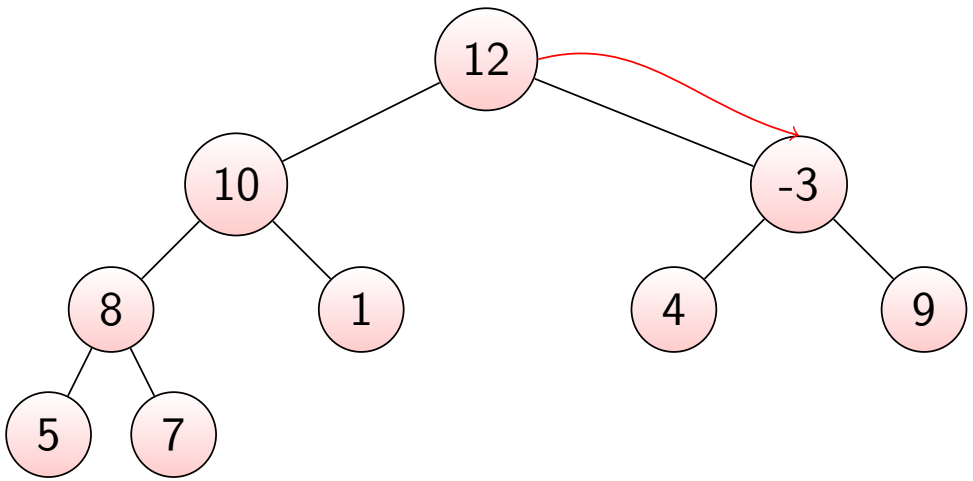


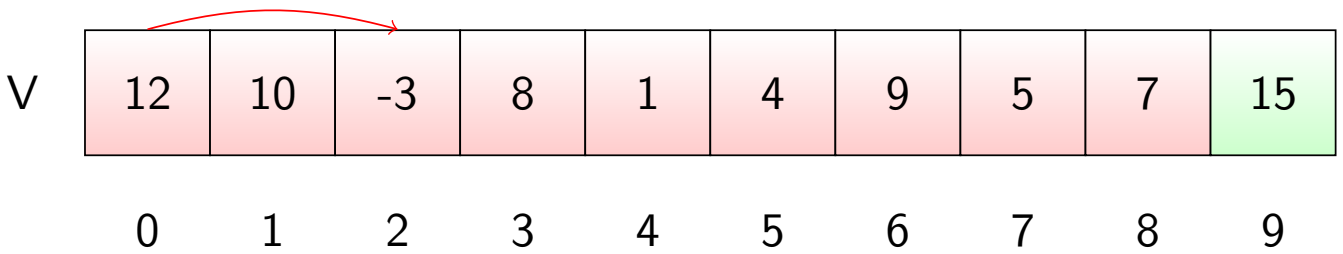


Heapsort *(Fase 2: ordenação)*

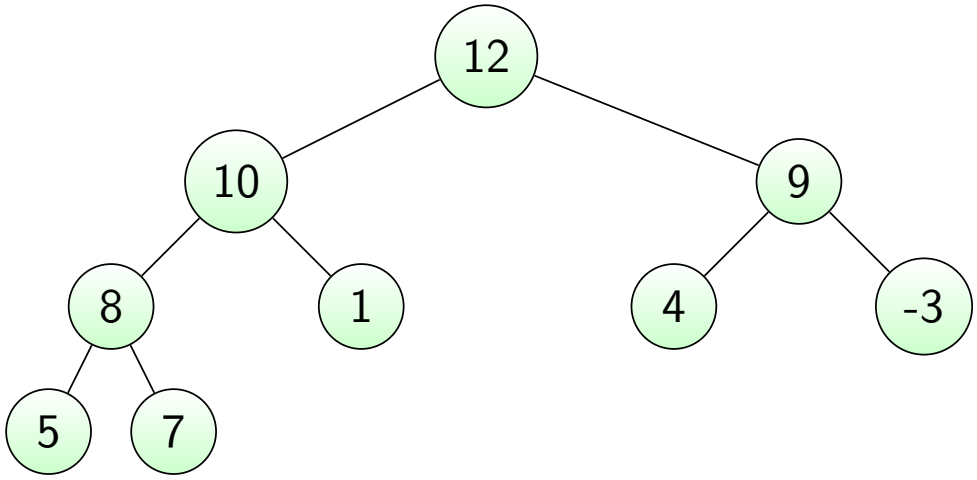


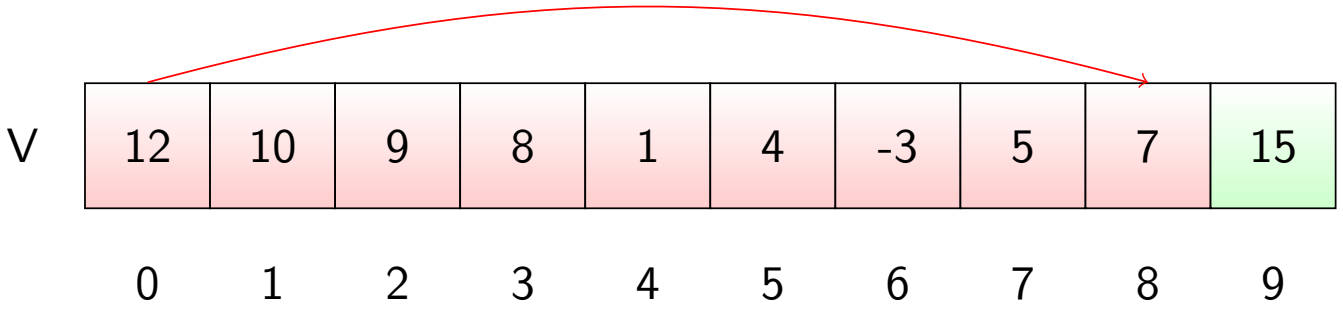
Heapsort *(Fase 2: ordenação)*



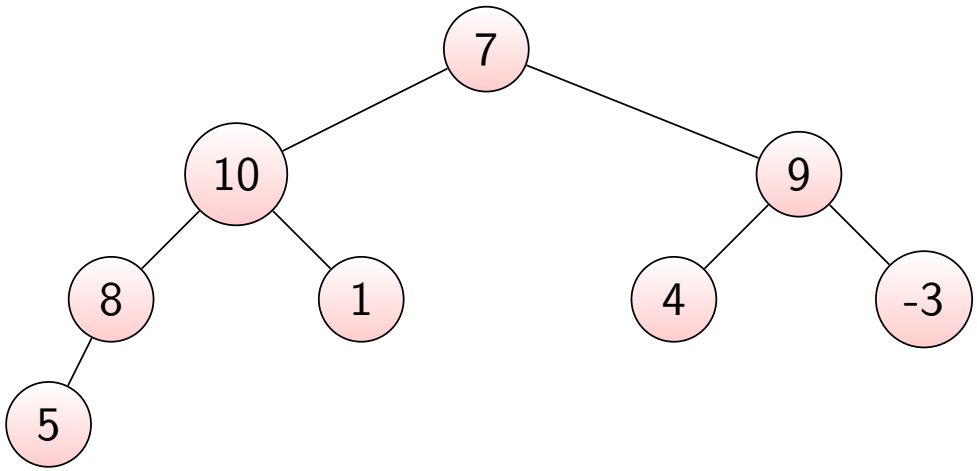


Heapsort *(Fase 2: ordenação)*



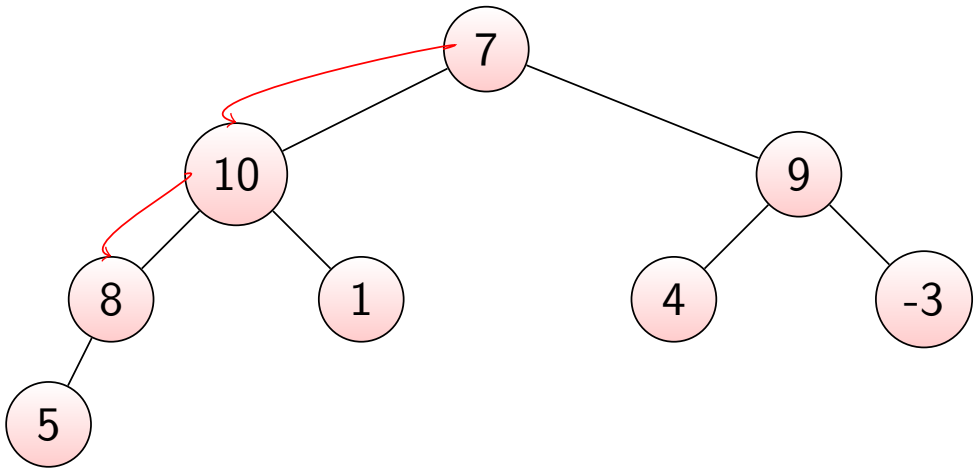


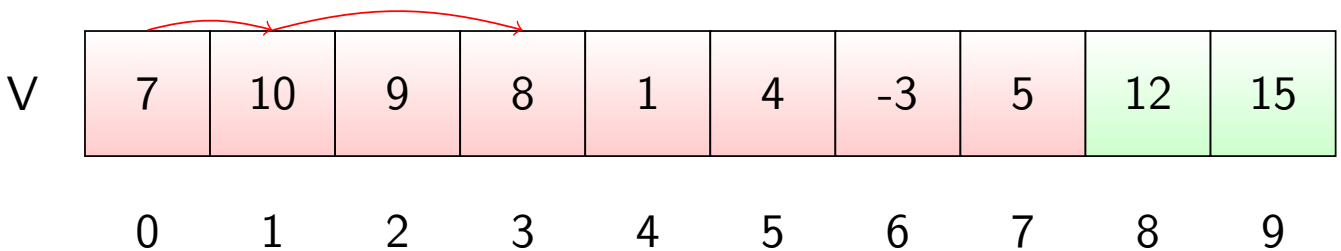
Heapsort *(Fase 2: ordenação)*



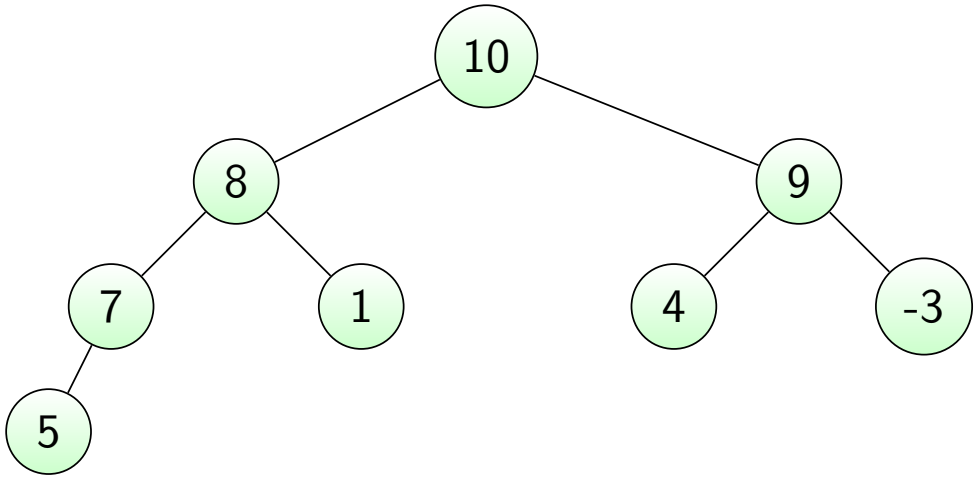
V	7	10	9	8	1	4	-3	5	12	15
	0	1	2	3	4	5	6	7	8	9

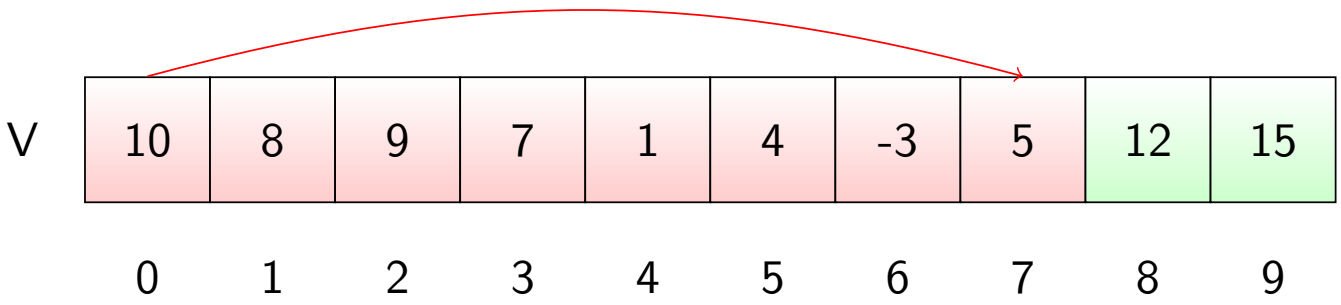
Heapsort *(Fase 2: ordenação)*



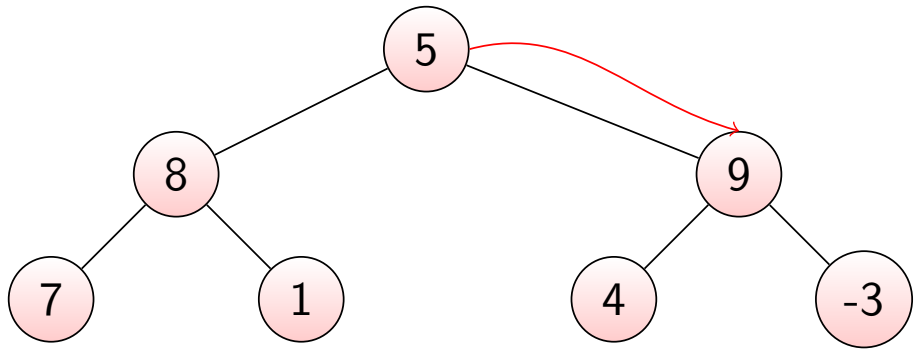


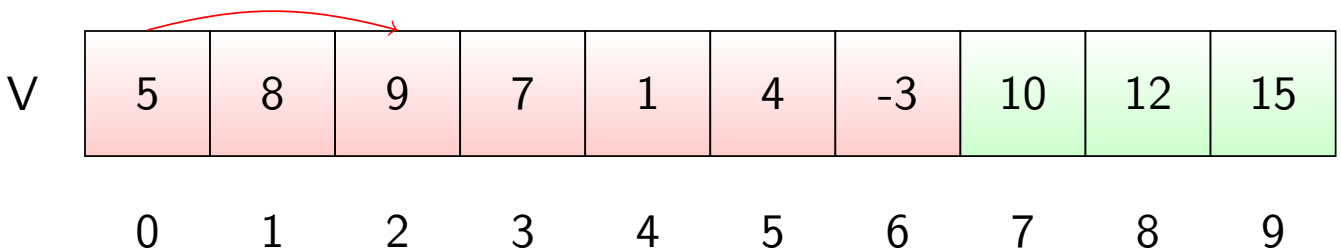
Heapsort *(Fase 2: ordenação)*



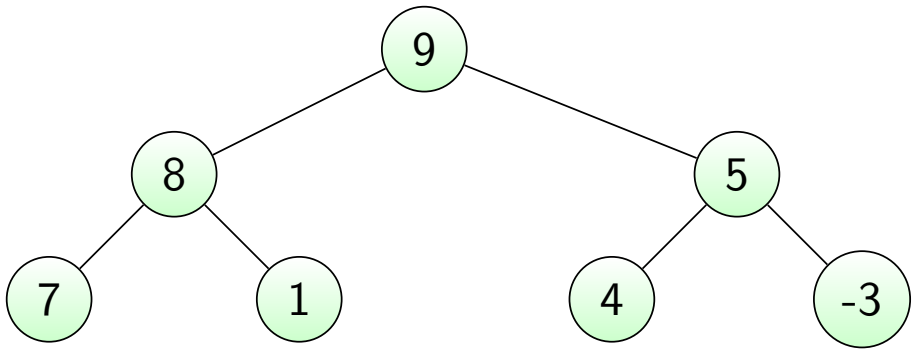


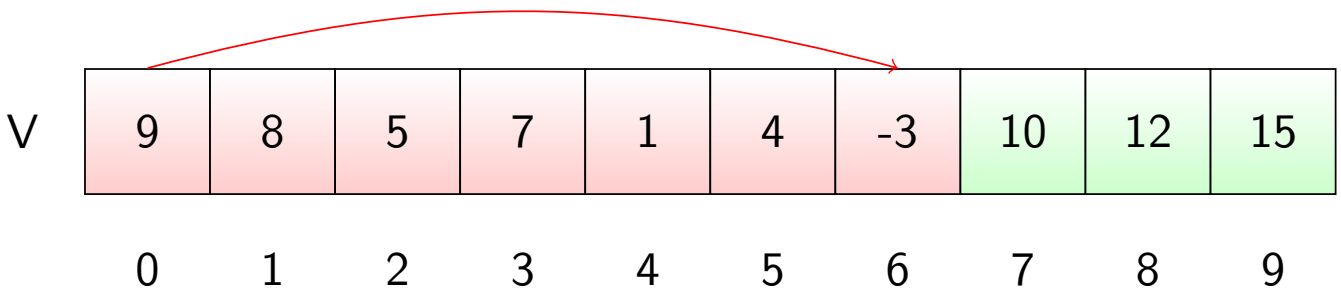
Heapsort *(Fase 2: ordenação)*



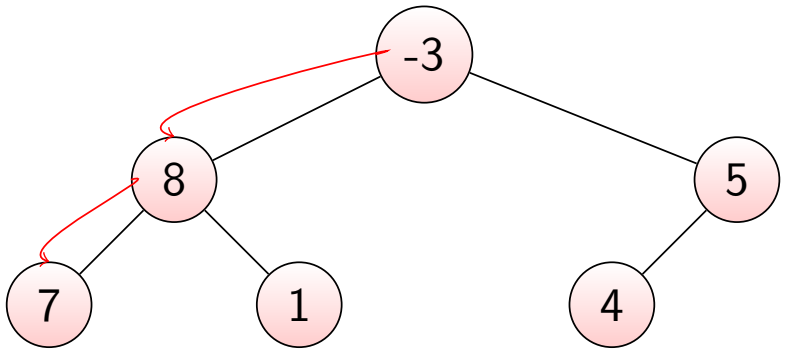


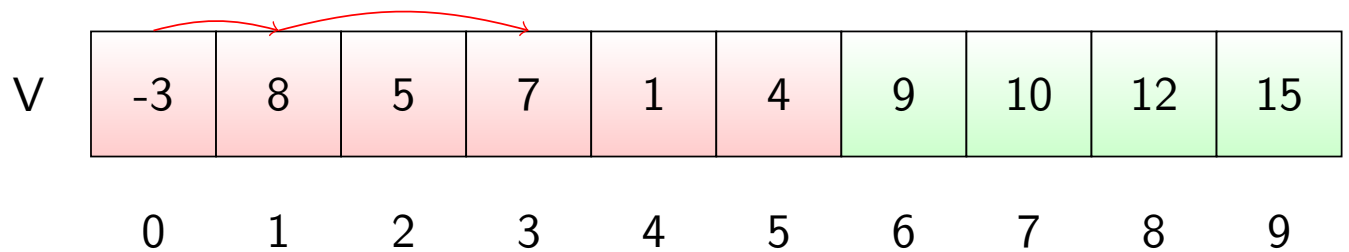
Heapsort *(Fase 2: ordenação)*



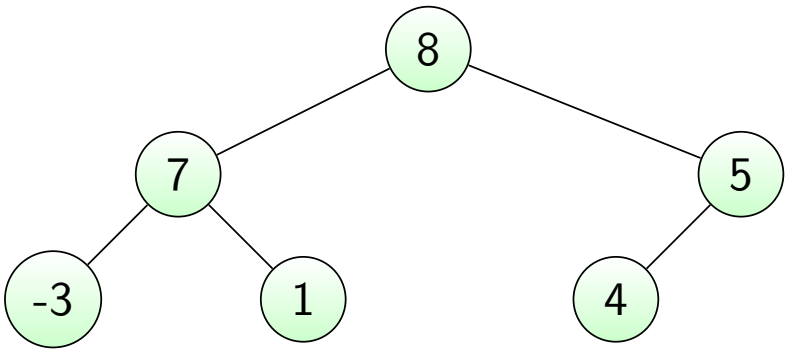


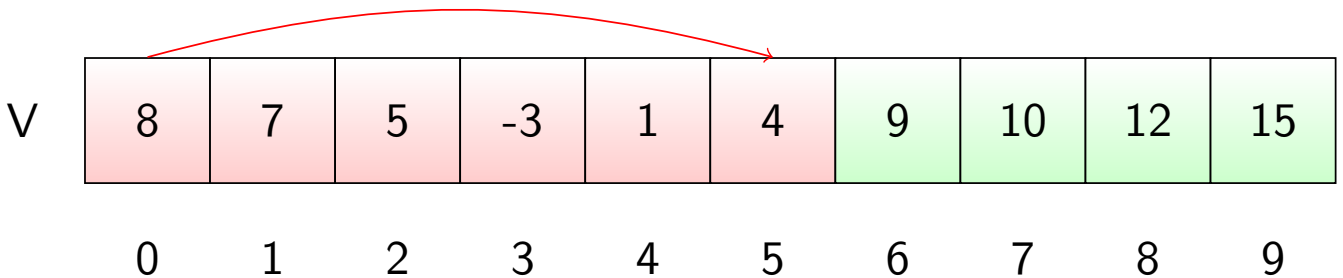
Heapsort *(Fase 2: ordenação)*



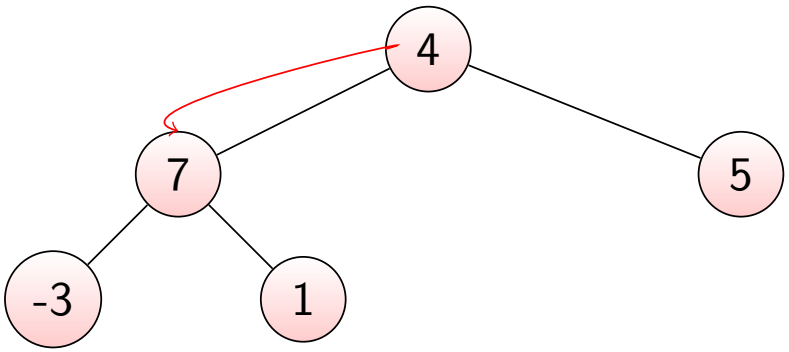


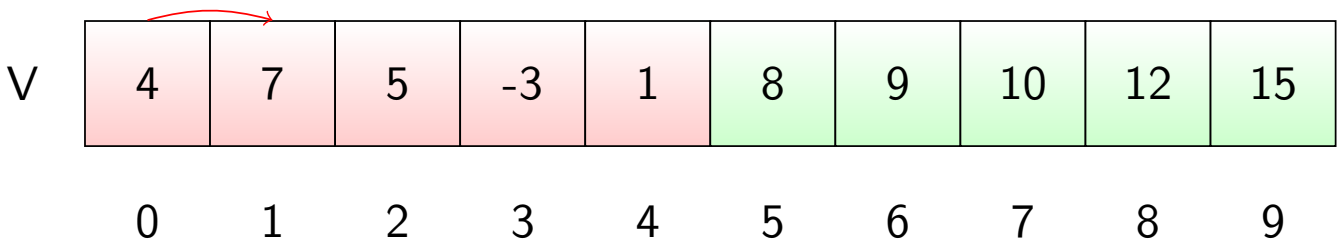
Heapsort *(Fase 2: ordenação)*



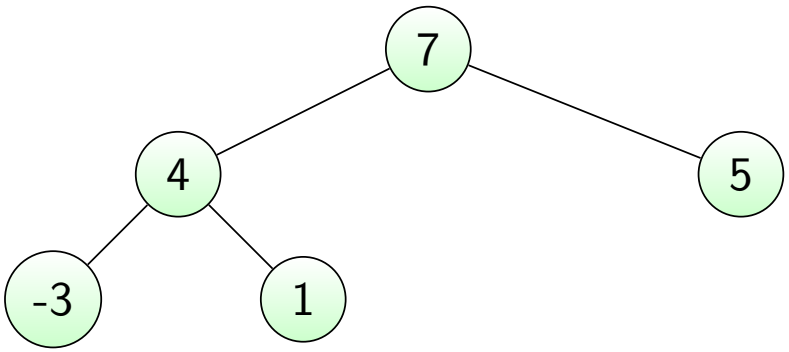


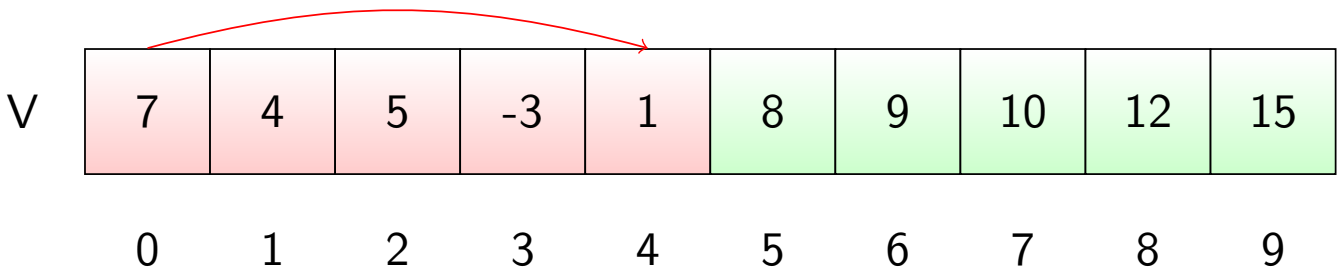
Heapsort *(Fase 2: ordenação)*



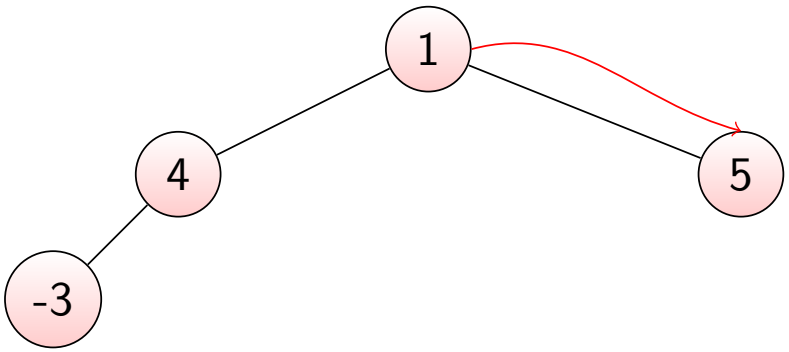


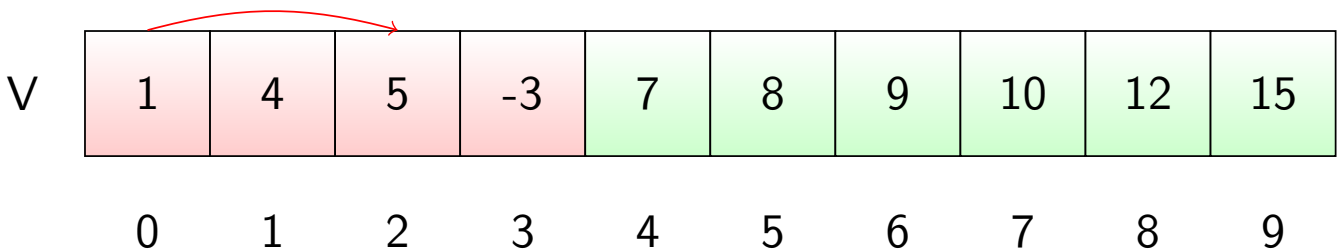
Heapsort *(Fase 2: ordenação)*



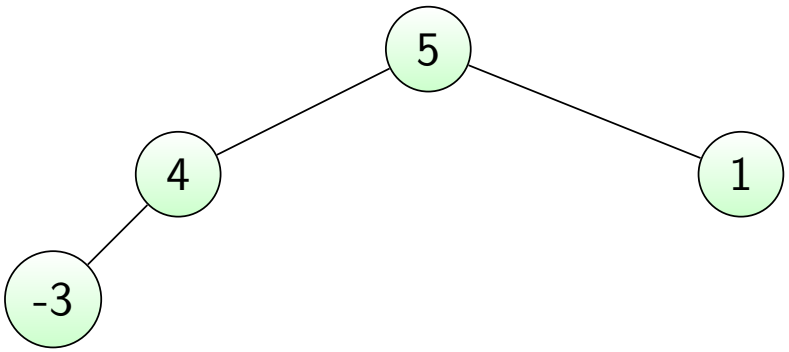


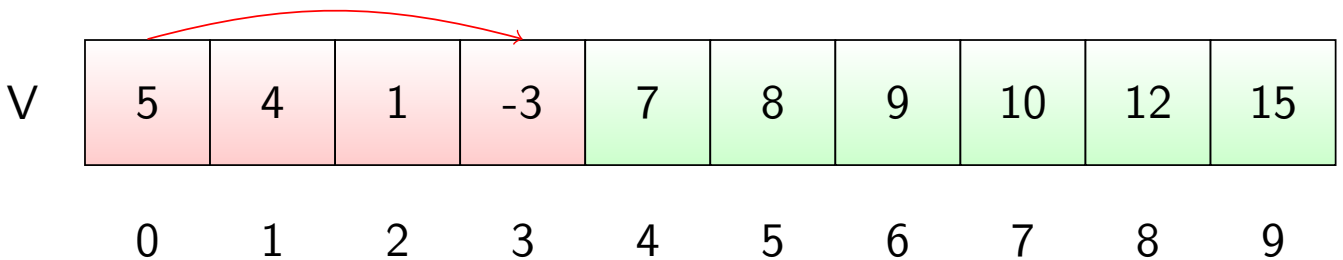
Heapsort *(Fase 2: ordenação)*



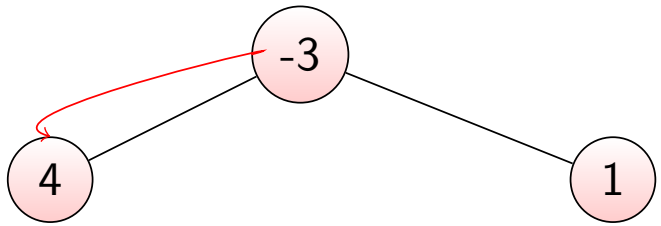


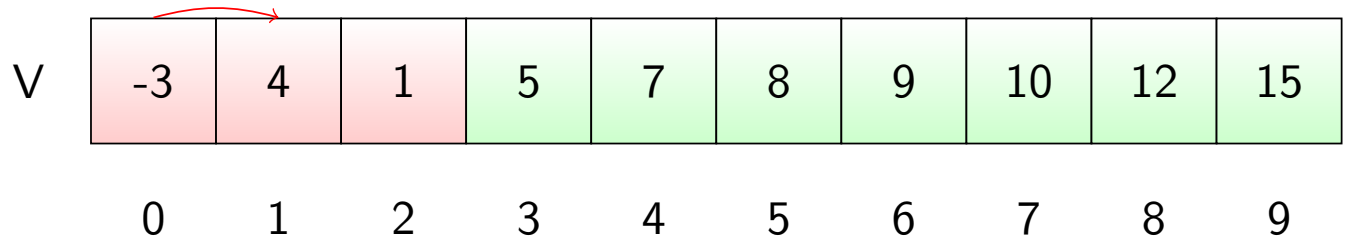
Heapsort *(Fase 2: ordenação)*



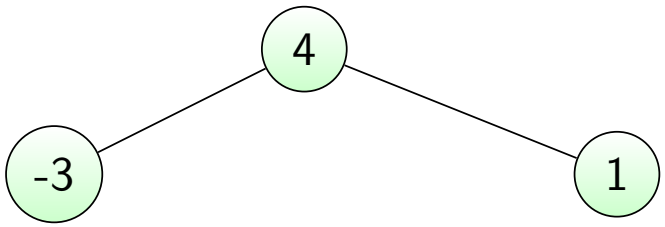


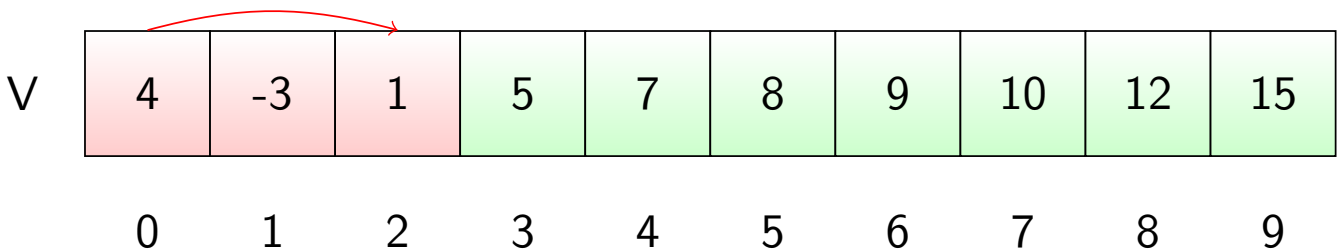
Heapsort *(Fase 2: ordenação)*



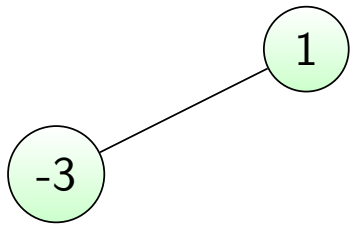


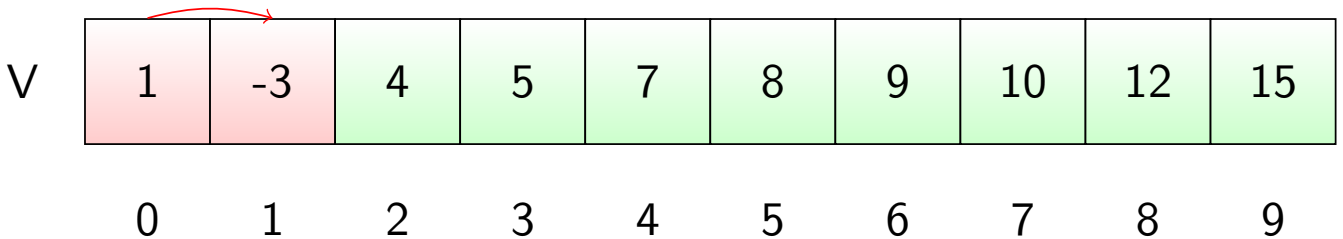
Heapsort *(Fase 2: ordenação)*



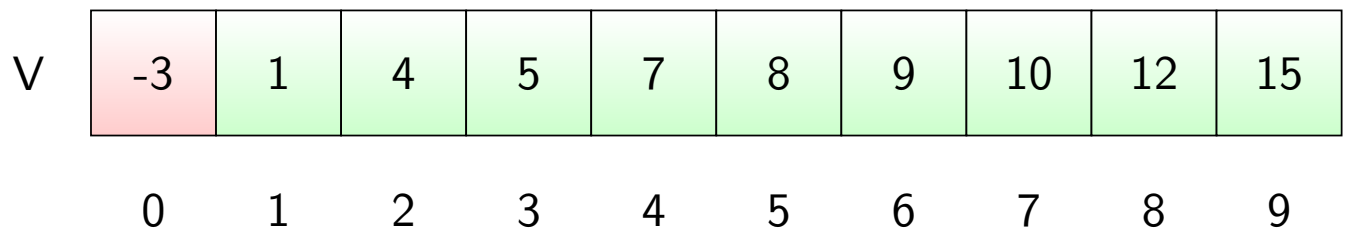


Heapsort *(Fase 2: ordenação)*





Heapsort *(Fase 2: ordenação)*



Heapsort *(Fase 2: ordenação)*

V	-3	1	4	5	7	8	9	10	12	15
	0	1	2	3	4	5	6	7	8	9

Função que ajeita a Heap

```
void ajeitaHeap (int *v, int tam){
    int t, f=1;
    while (f <= tam){
        if (f < tam && v[f] < v[f+1]) f++;
        if (v[(f-1)/2] >= v[f]) break;
           troca (v, (f-1)/2, f);
           f = 2*f + 1;
    }
    return;
}
```

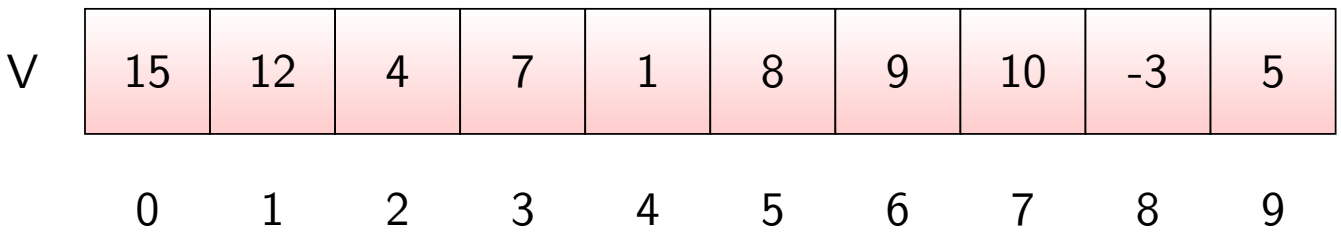
Função que ajeita a Heap

```
void heapsort (int* v, int tam){
    int m;
    constroiHeap(v, tam);
    for (m=tam-1; m>0; m--){
        troca (v, 0, m);
        ajeitaHeap (v, m-1);
    }
    return;
```

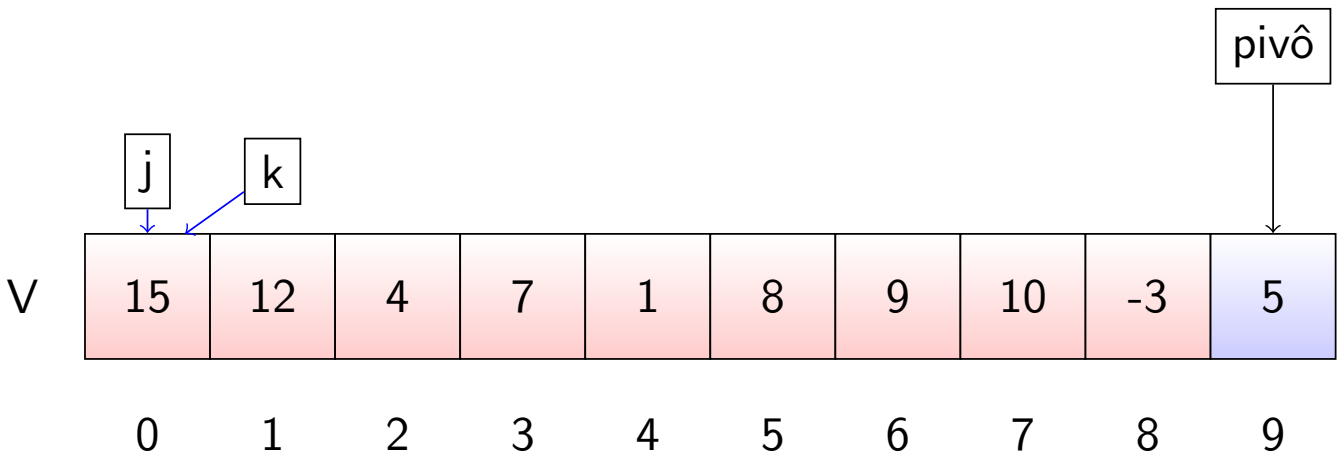
Quicksort

- Algoritmo eficiente muito famoso.
- Utiliza chamadas recursivas para particionar o vetor.
- Utiliza uma função auxiliar para separar elementos menores de elementos maiores.

Quicksort *(Particiona)*

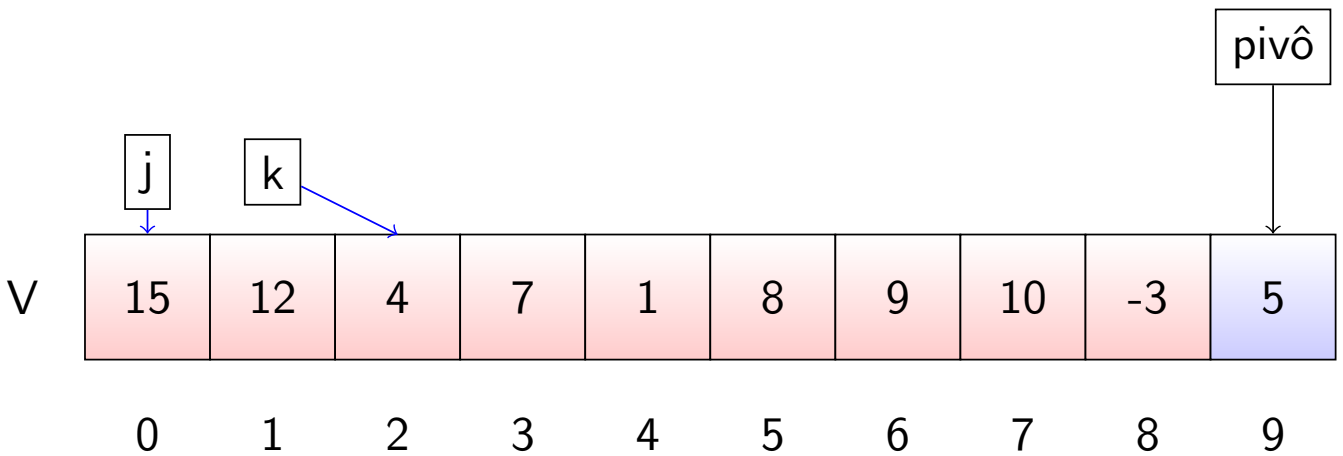


Quicksort *(Particiona)*

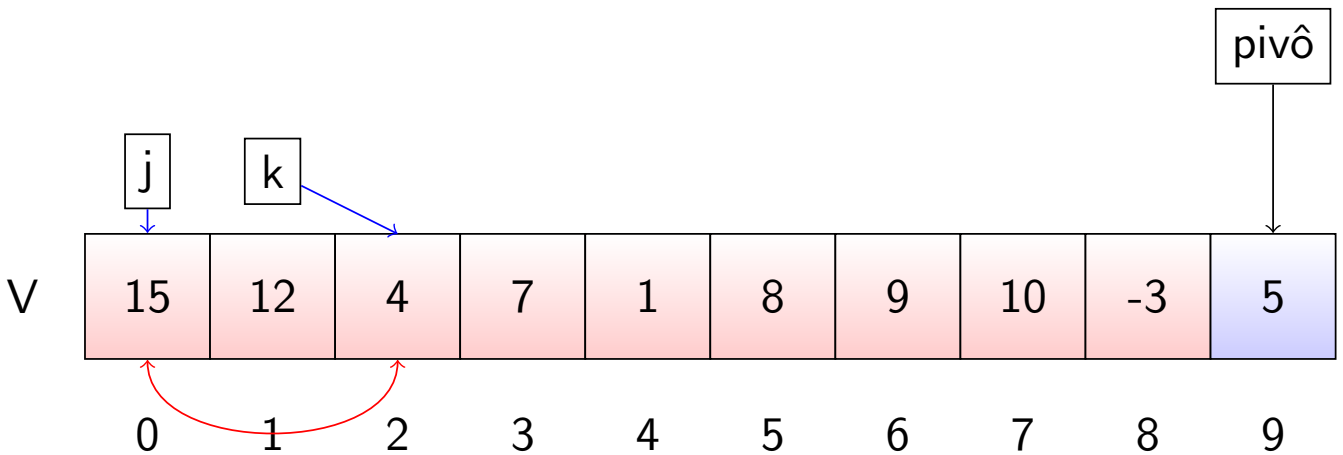


Quicksort *(Particiona)*

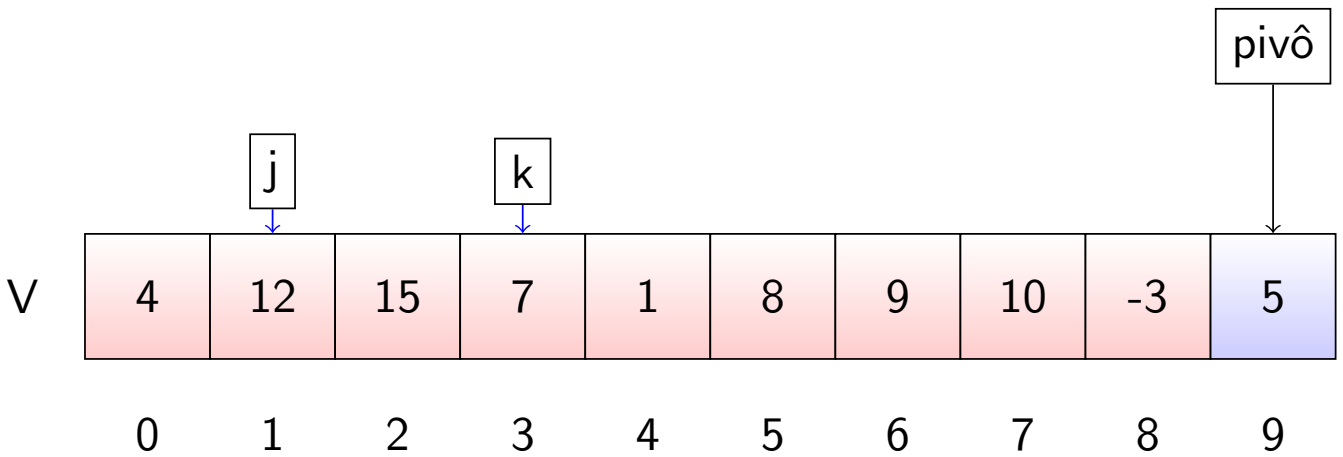
```
while (v[k] > pivô) k++;
```



Quicksort *(Particiona)*

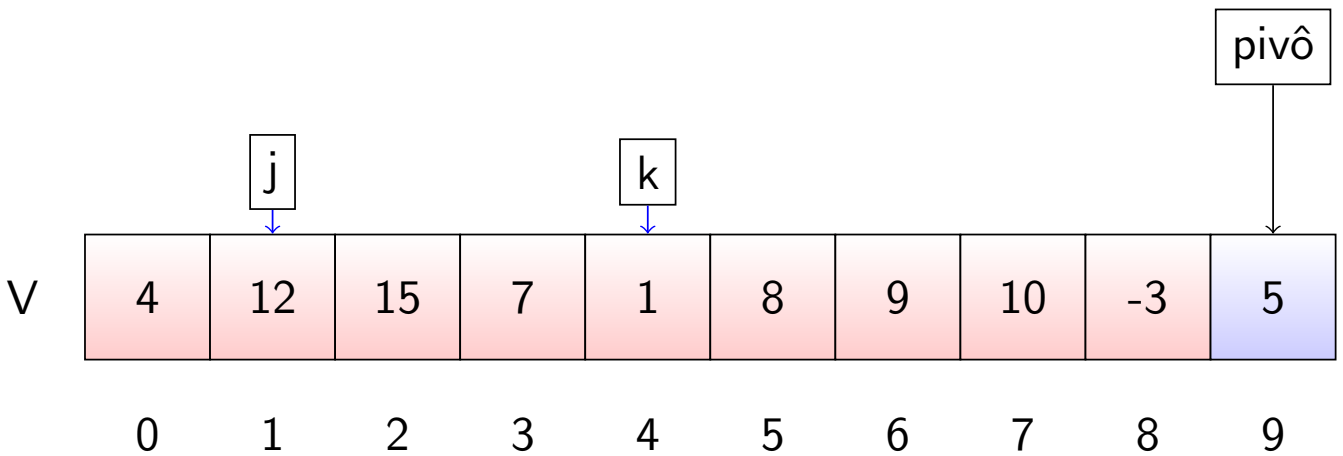


Quicksort *(Particiona)*

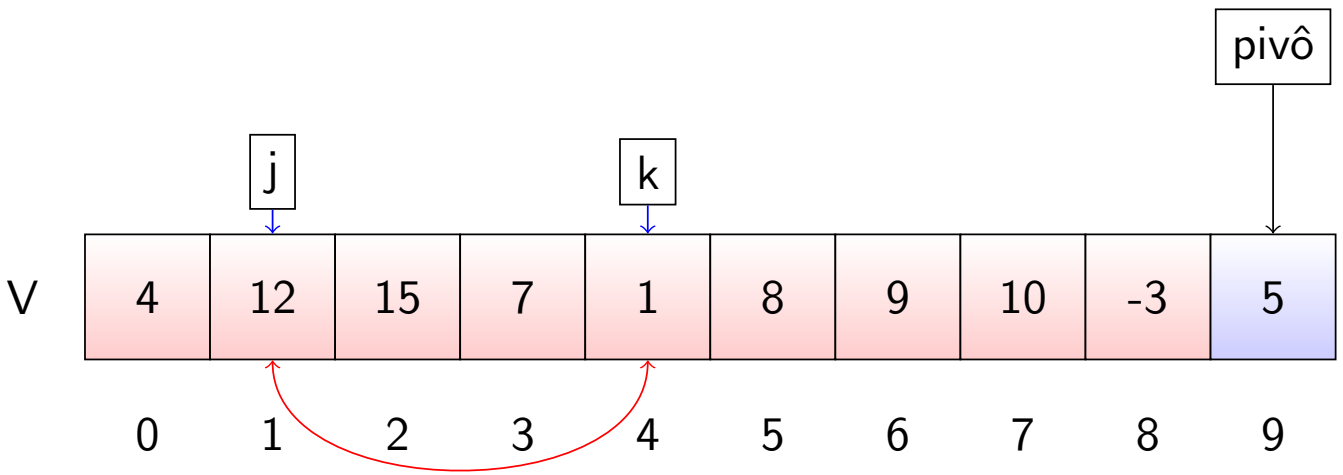


Quicksort *(Particiona)*

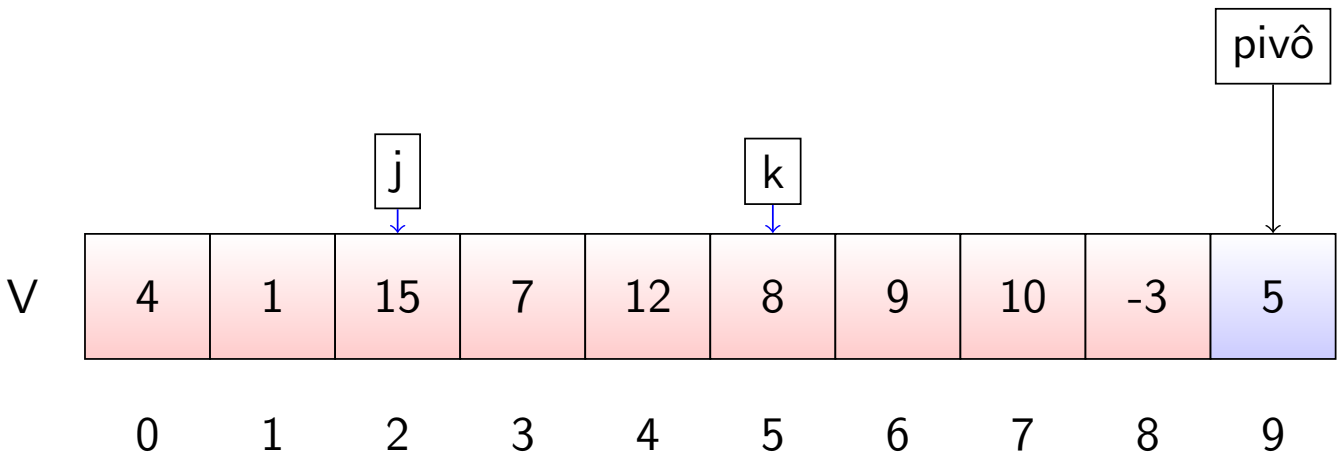
while (v[k] > pivô) k++;



Quicksort *(Particiona)*

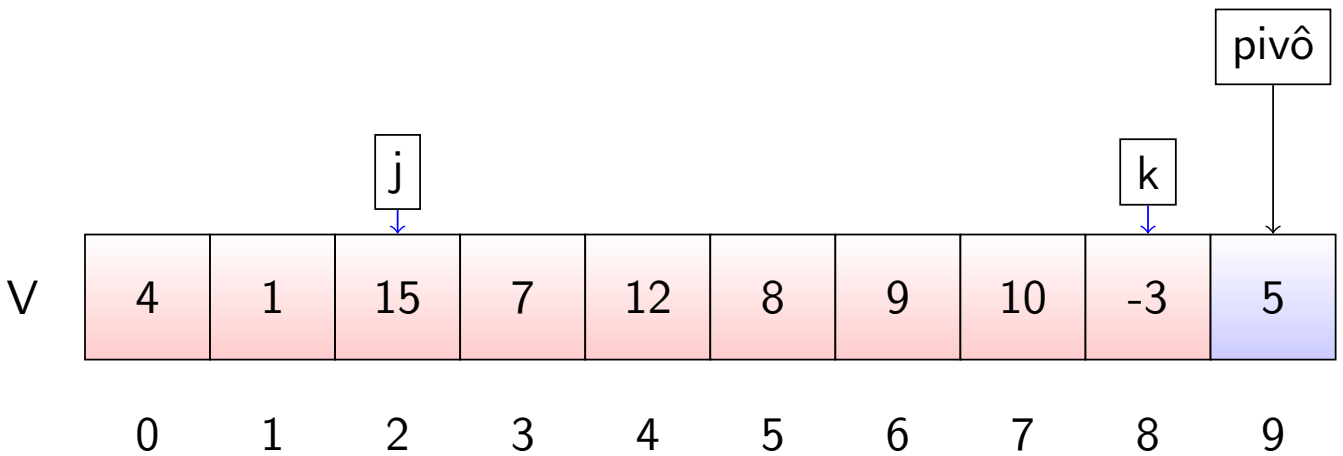


Quicksort *(Particiona)*

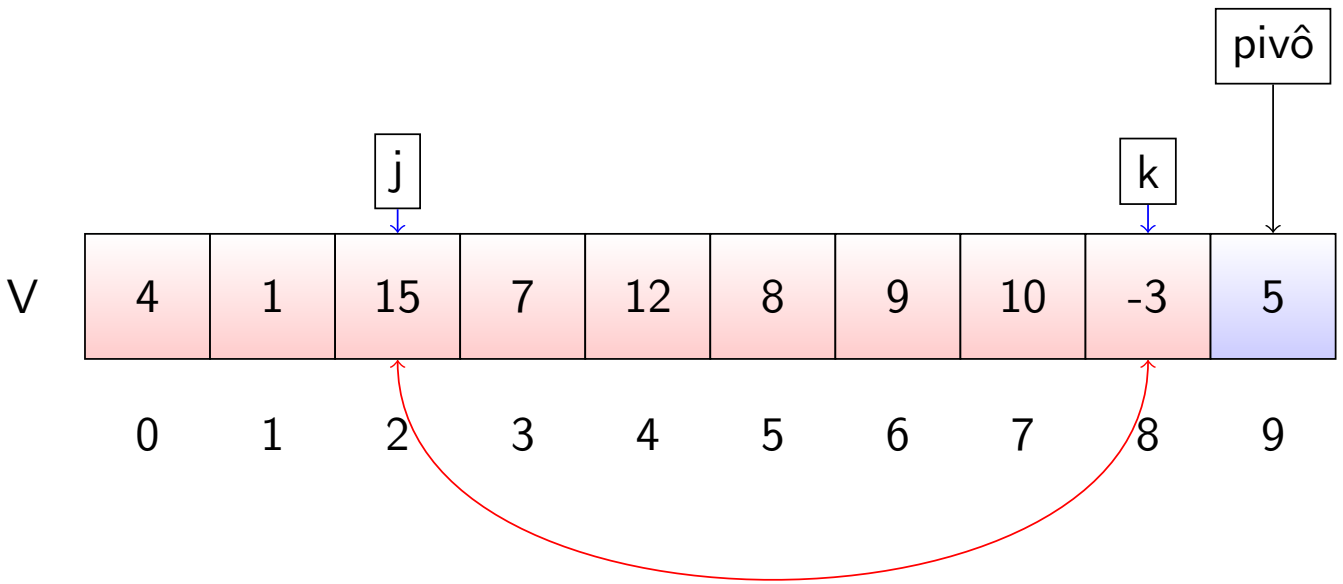


Quicksort *(Particiona)*

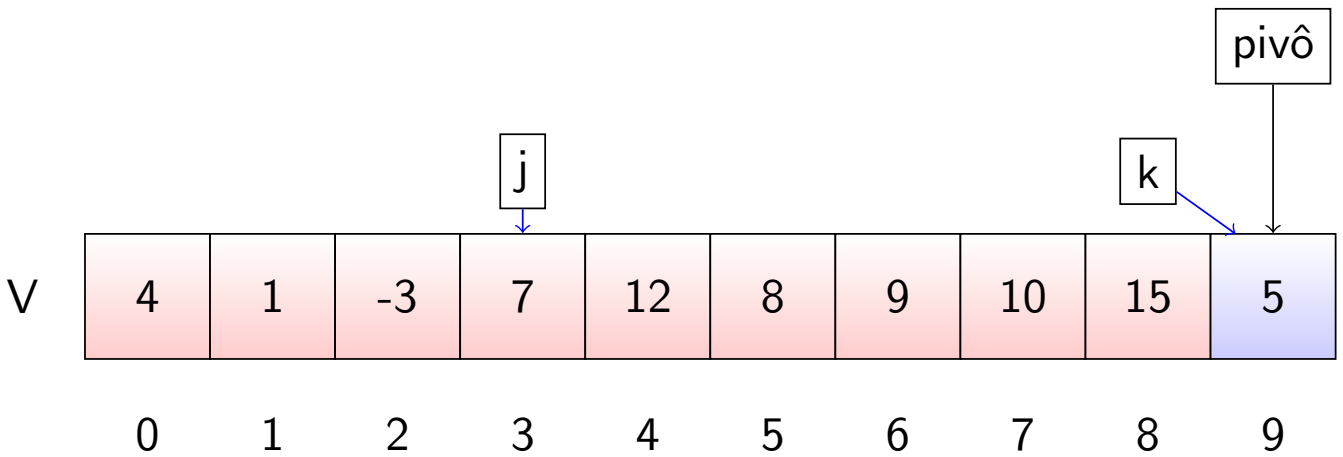
while (v[k] > pivô) k++;



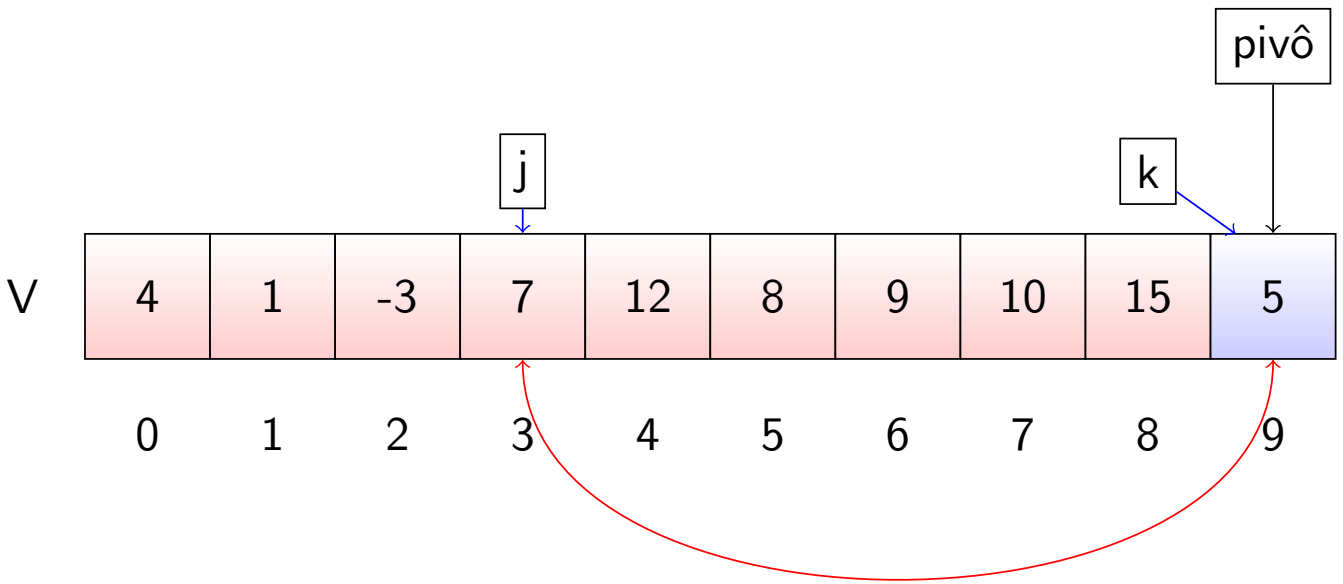
Quicksort *(Particiona)*



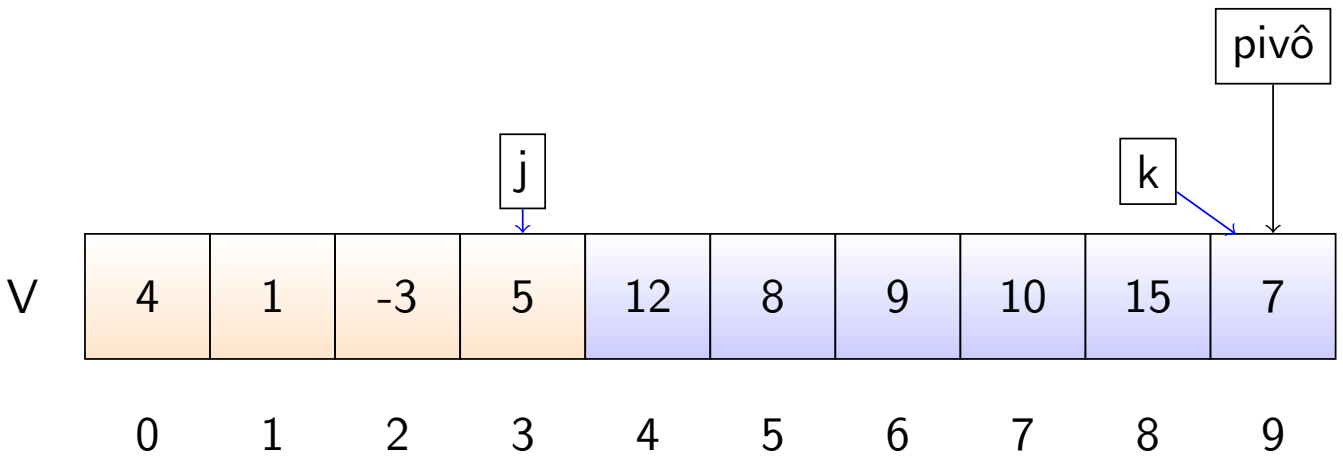
Quicksort *(Particiona)*



Quicksort *(Particiona)*



Quicksort *(Particiona)*



Quicksort

```
void quicksort (int* v, int p, int r){
    int j;
    if (p < r) {
            j = particiona (v, p, r);
            quicksort (v, p, j-1);
            quicksort (v, j+1, r);
    }
```

Quicksort

```
int particiona (int* v, int p, int r){
   int j, pivo, k;
   pivo = v[r];
   j = p;
   for (k=p; k < r; k++){
      if (v[k] <= pivo){
          troca (v[k], v[j]);
          j++;
      }
   }
   troca (v[j],pivo);
   return j;
}
```

Quicksort

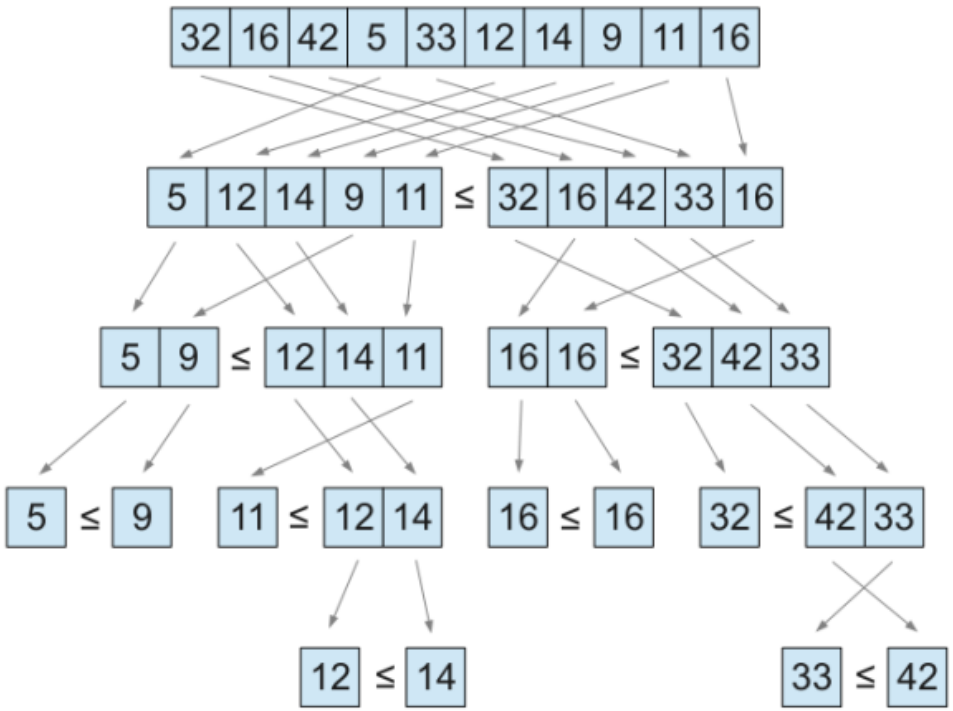


Figura: Execução do quicksort. Retirado de `https://simpledevcode.wordpress.com/2014/06/13/quicksort-in-c/`