Estruturas de Dados 1

481440

Abril/2018

Mario Liziér
lizier@ufscar.br

Priority Queue

- TAD Fila com prioridade (*mínima*)
  - insert(T)
  - T removeMin()
  - T min()
- TAD Fila com prioridade (*máxima*)
  - insert(T)
  - T removeMax()
  - T max()
- "Fila" com prioridade (conjunto de elementos)
  - Remoção do item com maior/menor prioridade

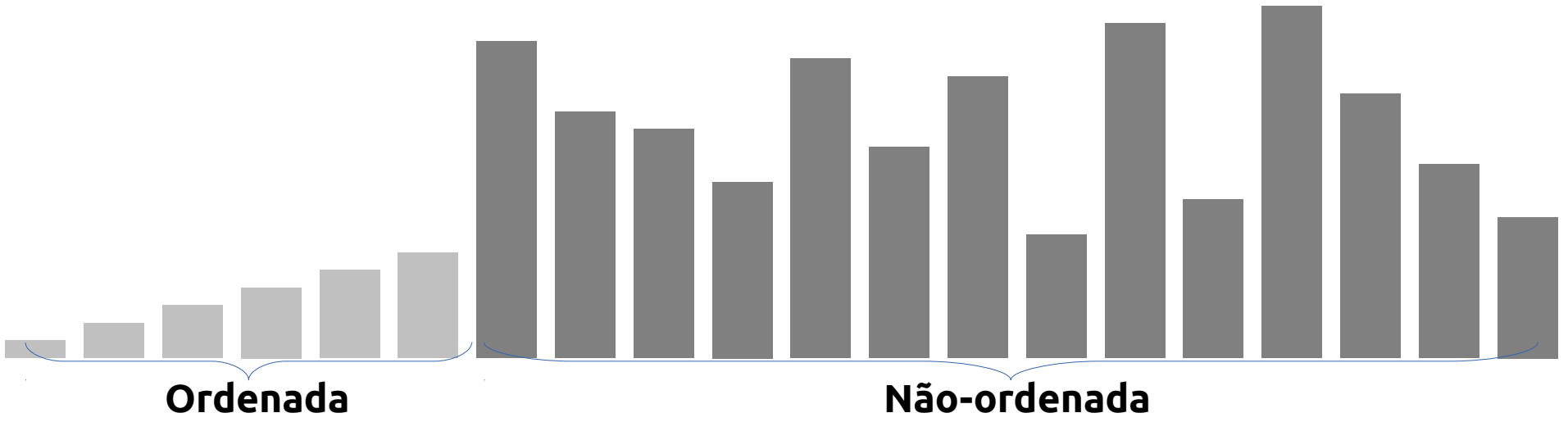
- Implementações?

***Priority** Queue*

- “Fila” com prioridade (conjunto de elementos)
  - Inserção de elementos no conjunto
  - Remoção do item com maior/menor prioridade
- Implementações:
  - *SelectionSort-based*
  - *InsertionSort-based*
  - *Heap-based*

SelectionSort

- Divisão dos dados em duas sequências: ordenada e não-ordenada
- Iteração: procurar pelo **menor** elemento da sequência não-ordenada e concatená-lo na sequência ordenada



SelectionSort

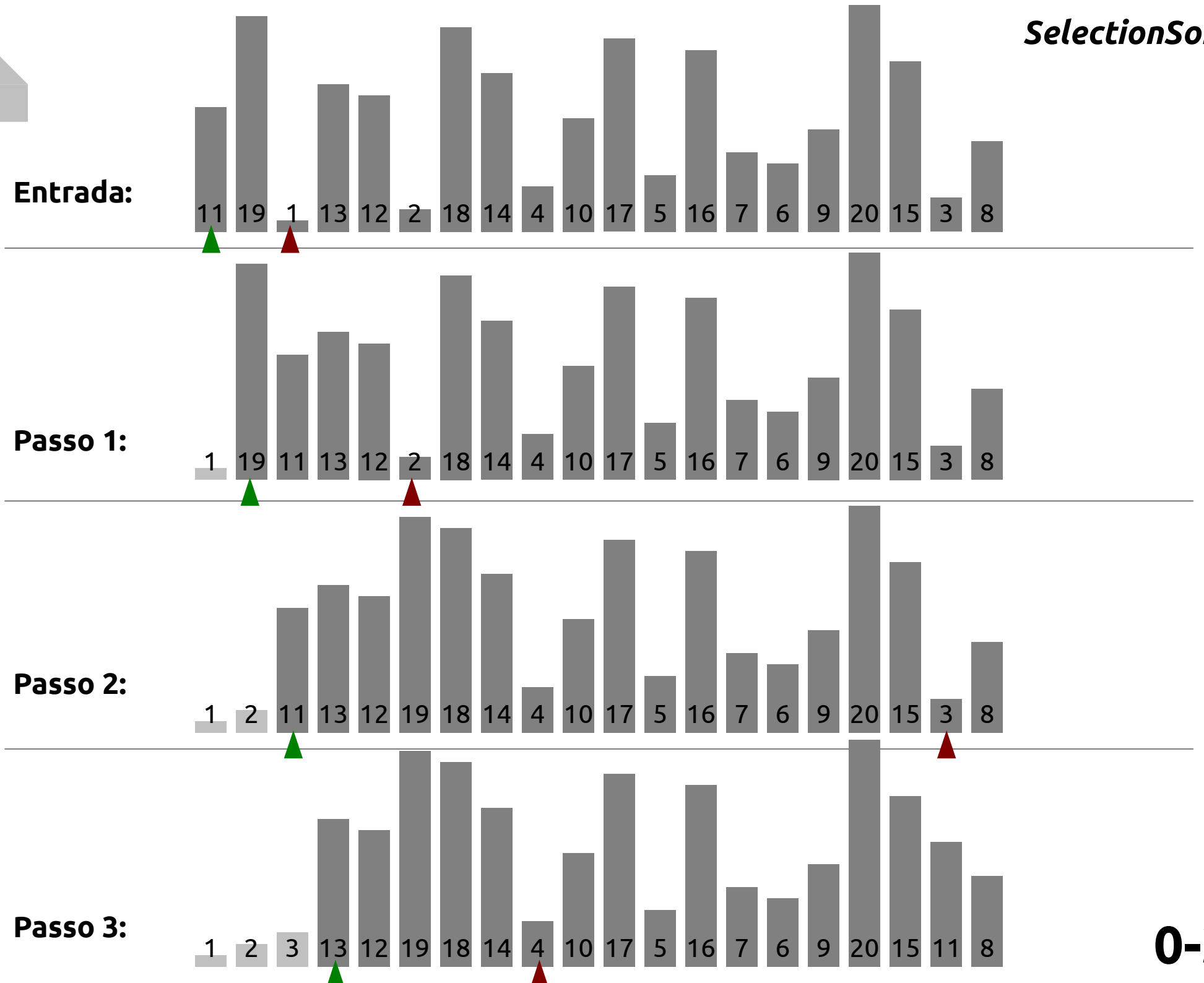
```
void selection(Item vetor[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int min = i;

        for (int j = i+1; j < n; j++)
            if (vetor[j] < vetor[min])
                min = j;

        swap(vetor[i], vetor[min]);

    }
}
```

i
controla a iteração,
índice da sequência ordenada

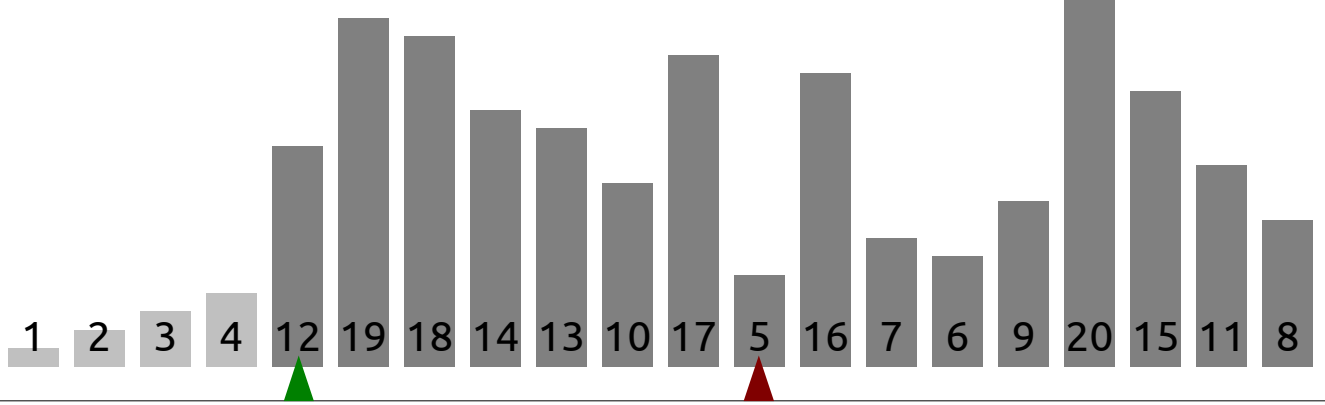
j
controla a busca pelo valor mínimo, índice da sequência não-ordenada

min
Índice do menor elemento da sequência não-rdenada

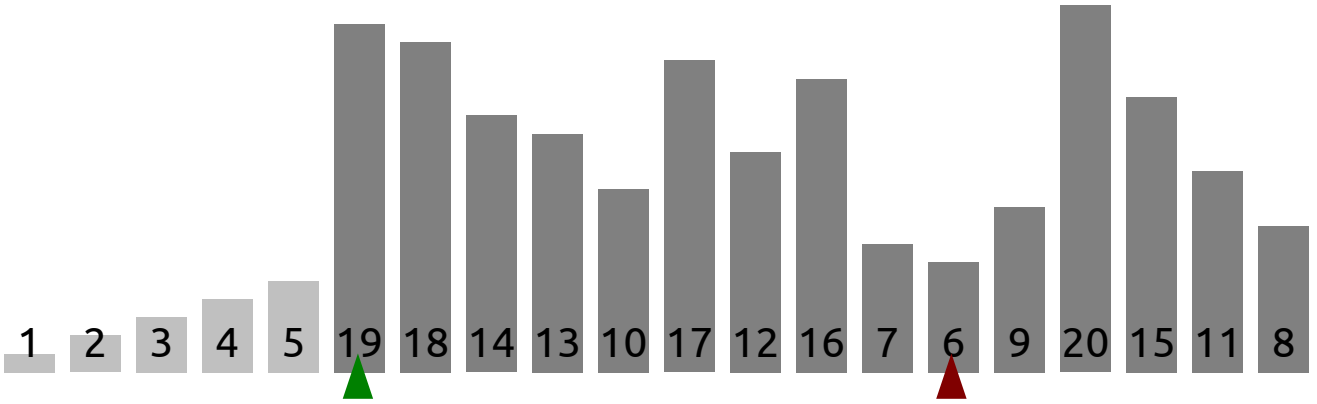
```
void swap(Item &A, Item &B)
  { Item t = A ; A = B; B = t; }
```


Entrada:
11 19 1 13 12 2 18 14 4 10 17 5 16 7 6 9 20 15 3 8
Passo 1:
1 19 11 13 12 2 18 14 4 10 17 5 16 7 6 9 20 15 3 8
Passo 2:
1 2 11 13 12 19 18 14 4 10 17 5 16 7 6 9 20 15 3 8
Passo 3:
1 2 3 13 12 19 18 14 4 10 17 5 16 7 6 9 20 15 11 8

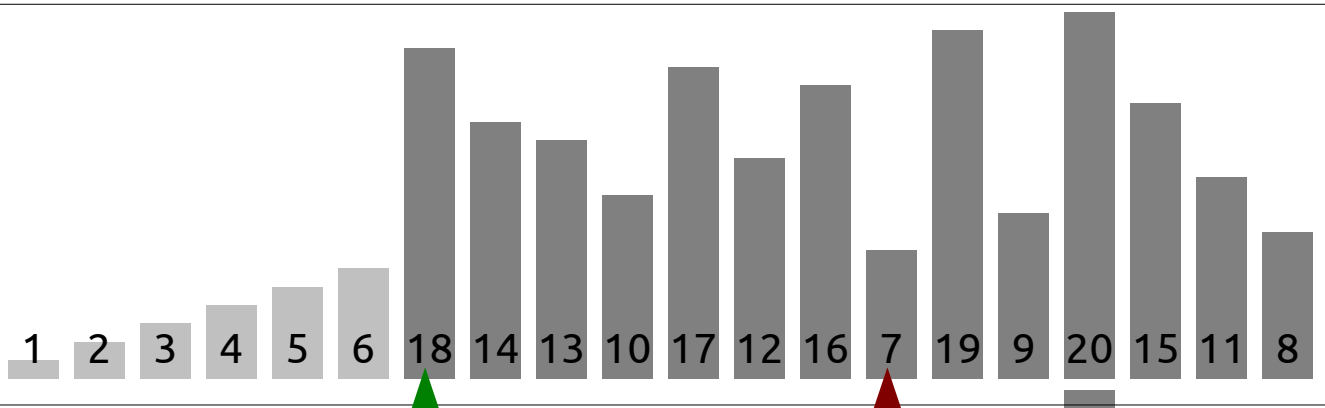
Passo 4:



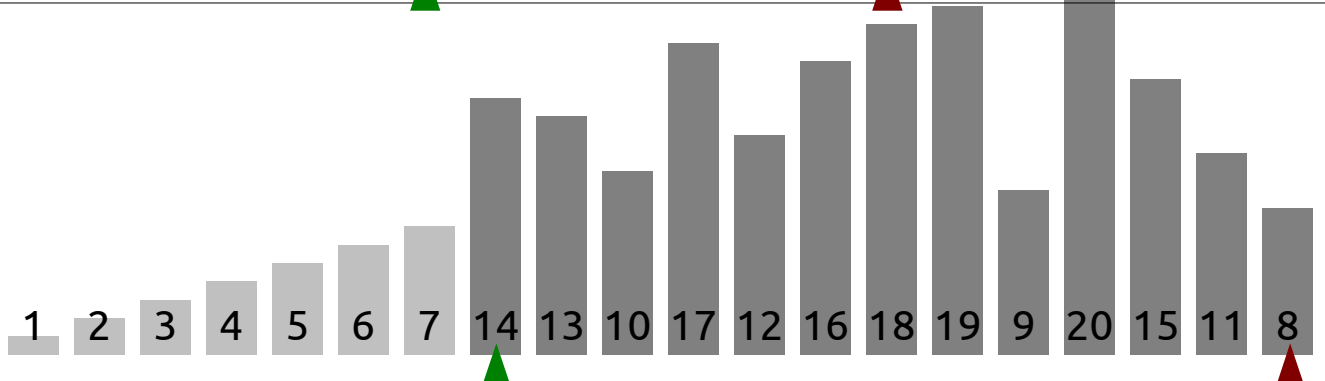
Passo 5:



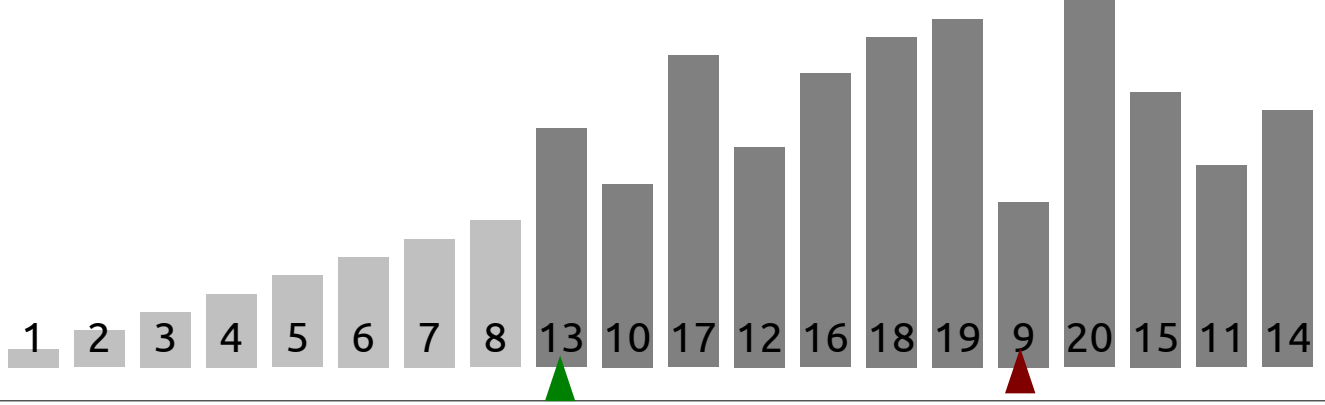
Passo 6:



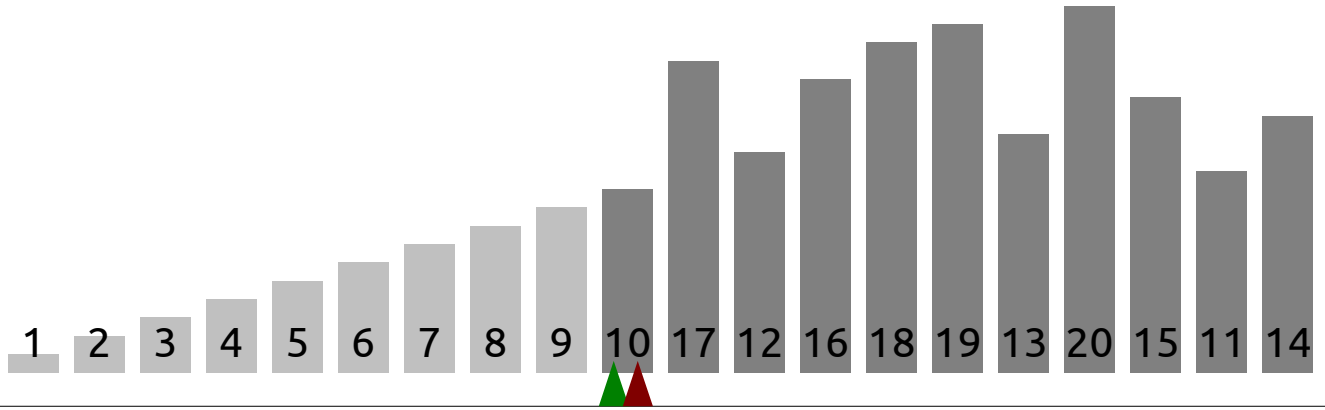
Passo 7:



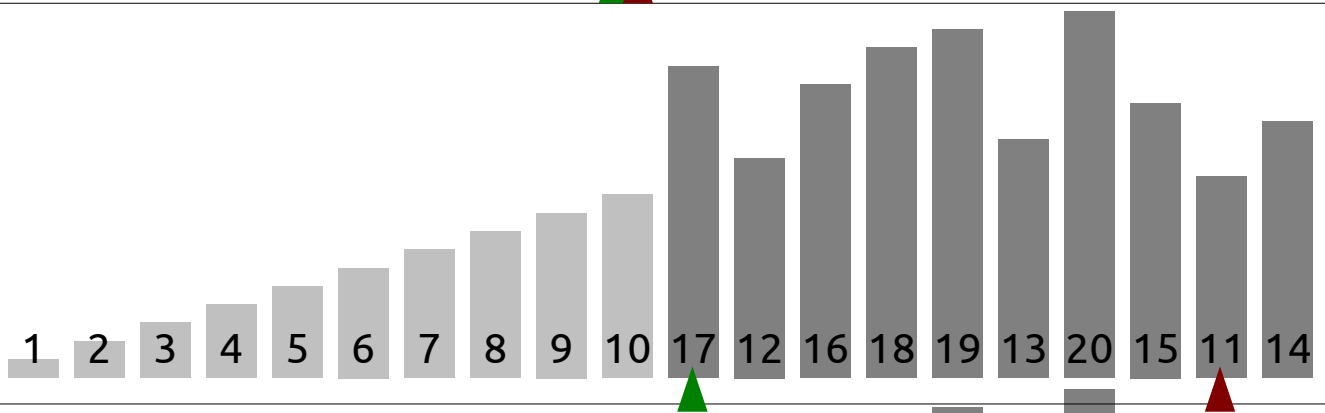
Passo 8:



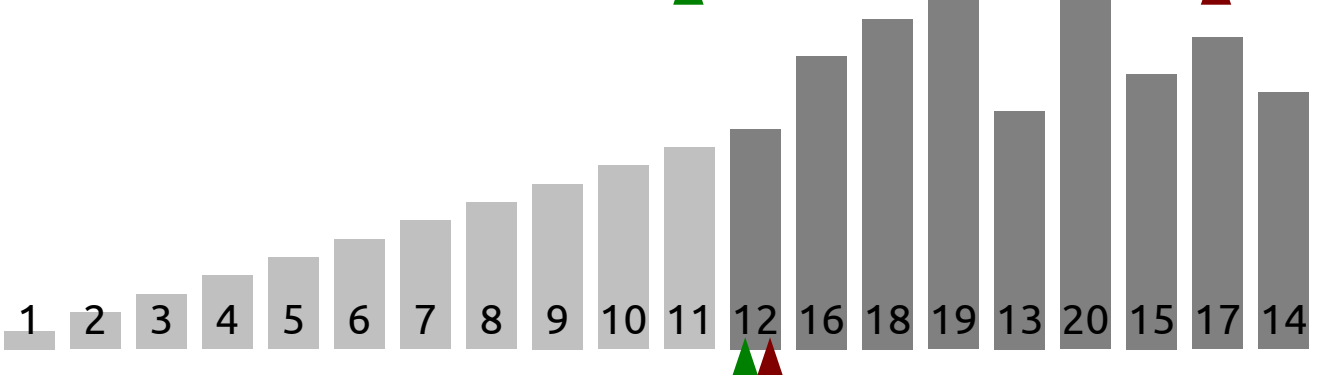
Passo 9:



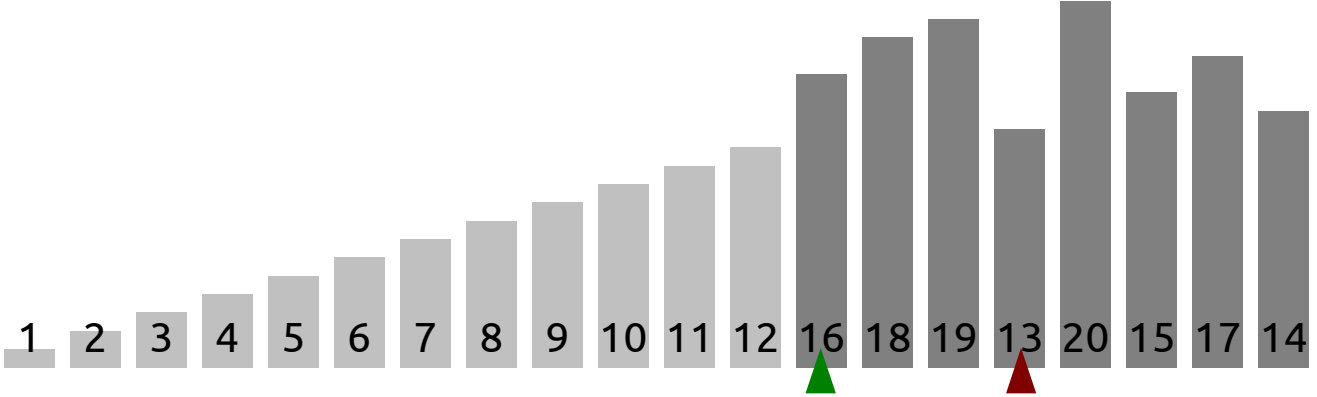
Passo 10:



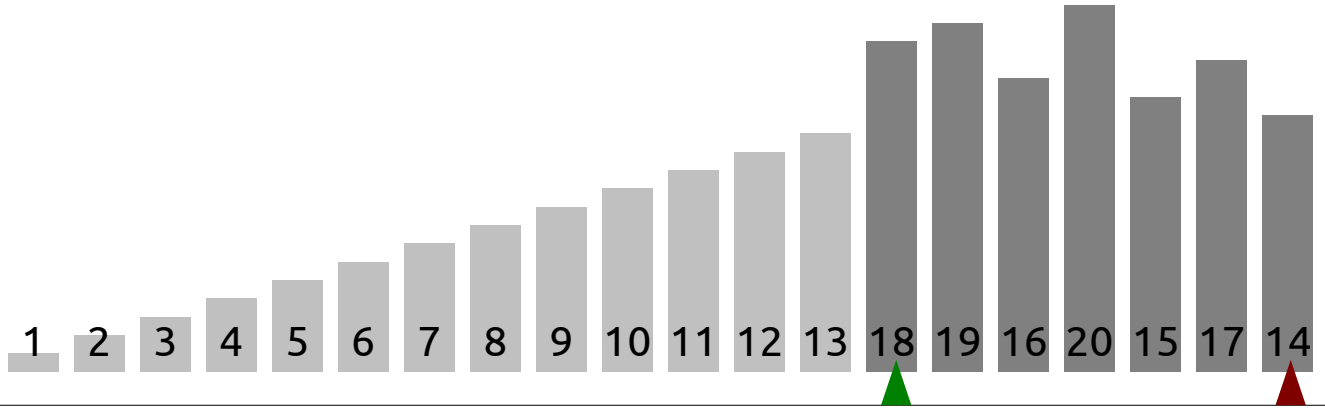
Passo 11:



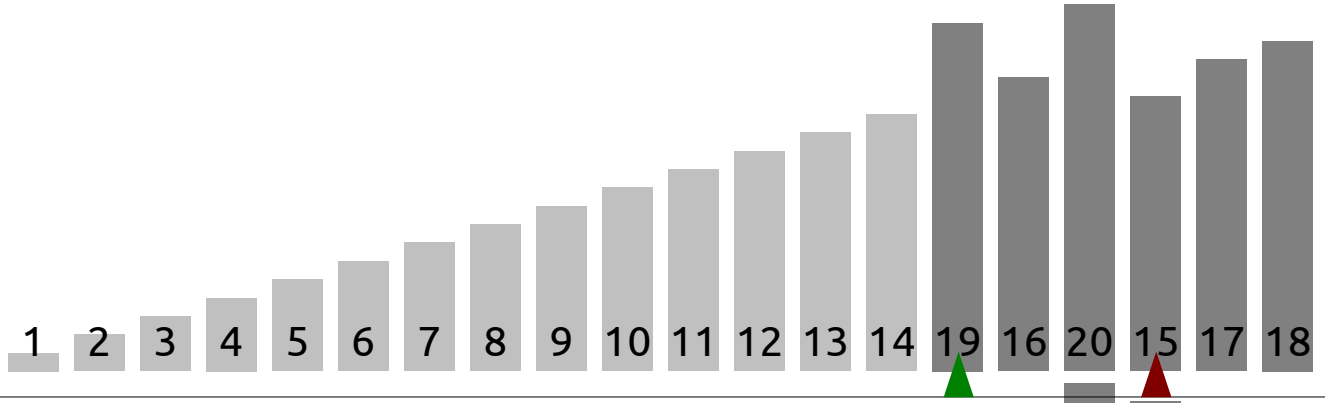
Passo 12:



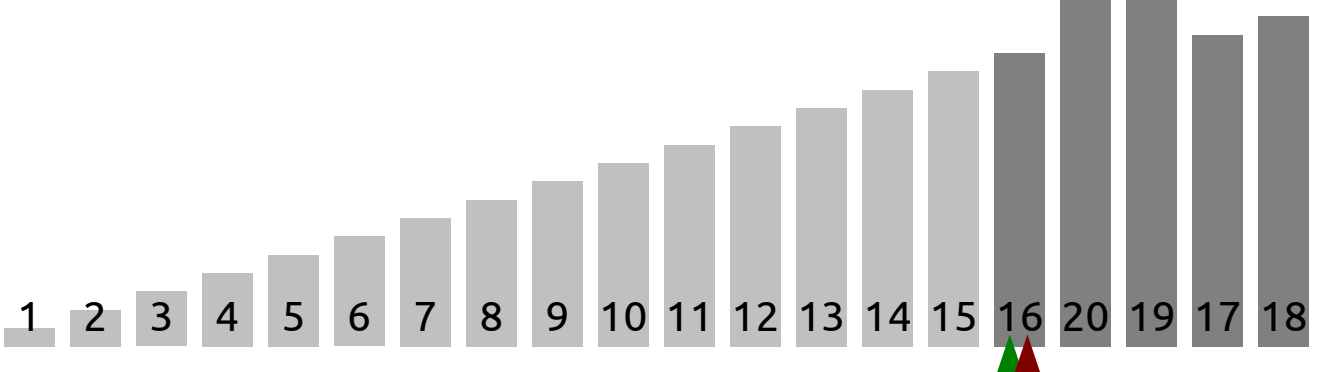
Passo 13:



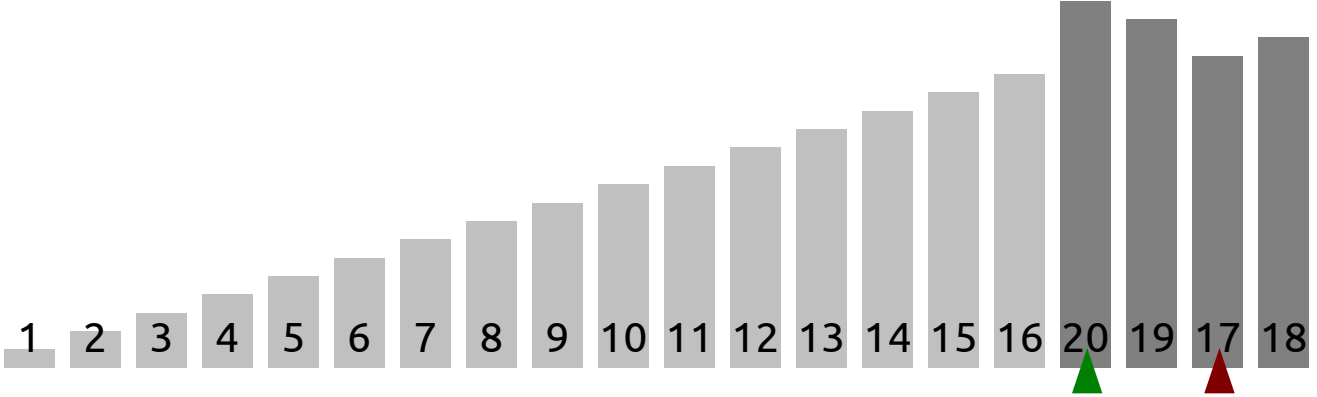
Passo 14:



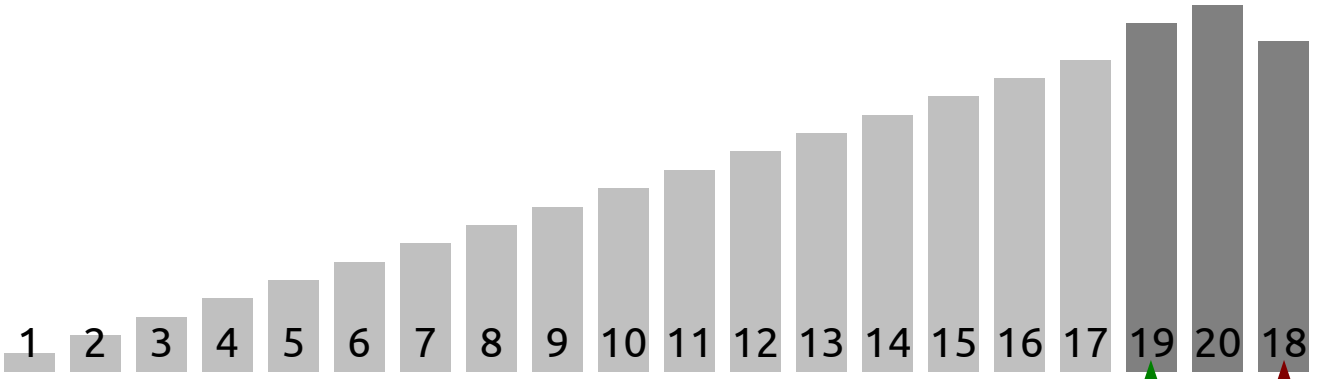
Passo 15:



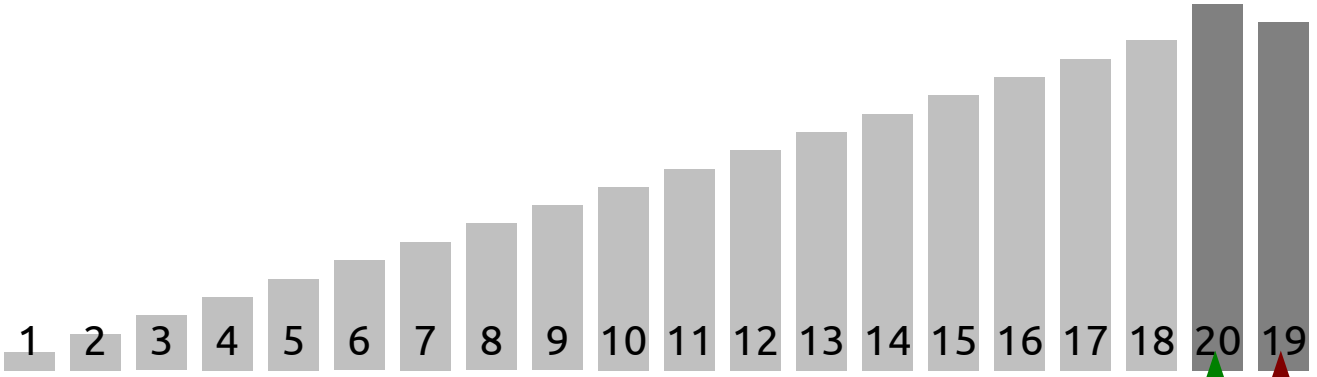
Passo 16:



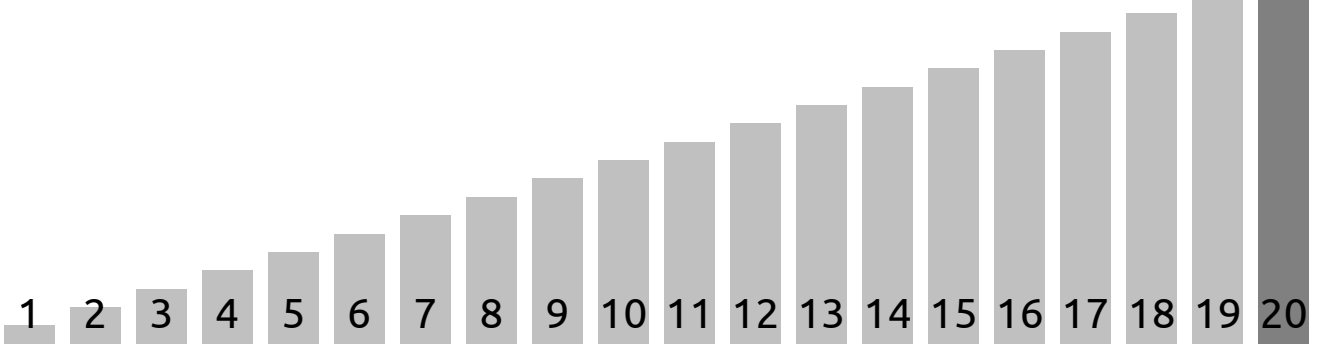
Passo 17:



Passo 18:



Passo 19:



SelectionSort

- Quantas comparações são executadas?
- Quantas trocas são executadas?

```
void selection(Item vetor[], int n)
{
    for (int i = 0; i < n; i++)
   {
        int min = i;

        for (int j = i+1; j < n; j++)
            if (vetor[j] < vetor[min])
                min = j;

        swap(vetor[i], vetor[min]);

    }
}
```

SelectionSort

- Quantas comparações são executadas?
- Quantas trocas são executadas?

Sempre!
O número de comparações ou trocas não dependem dos valores dos dados.

n-1, n-2, n-3, ..., 1

n x 1

```
void selection(Item vetor[], int n)
{
    for (int i = 0; i < n; i++)
   {
        int min = i;

        for (int j = i+1; j < n; j++)
            if (vetor[j] < vetor[min])
                min = j;

        swap(vetor[i], vetor[min]);

    }
}
```

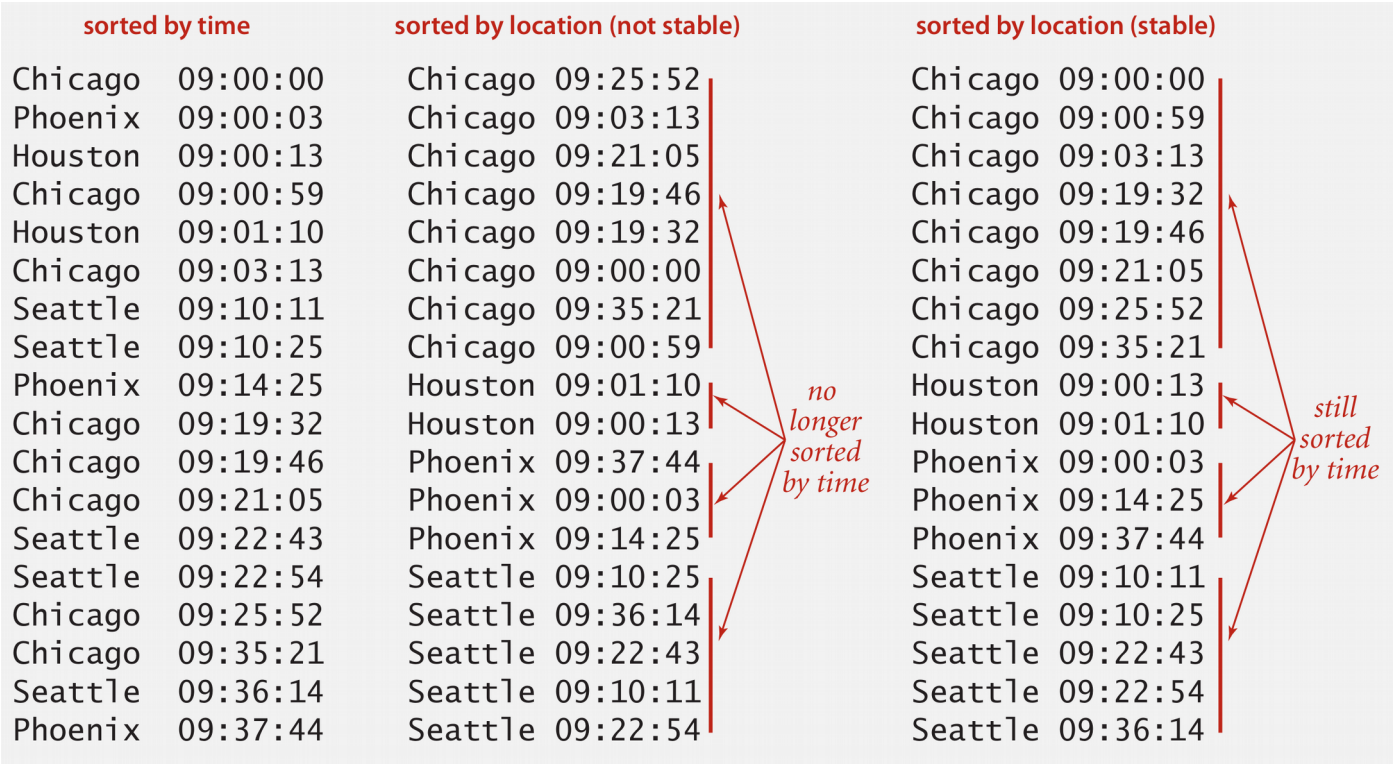
SelectionSort

- Os valores dos dados não interferem na execução do algoritmo
- Crescimento do número de trocas em relação ao tamanho de entrada: **linear**
- Crescimento do número de comparações em relação ao tamanho de entrada: **quadrático**
- Crescimento do uso de memória em relação ao tamanho da entrada: **constante**

- Algoritmo não é **estável**

Estabilidade

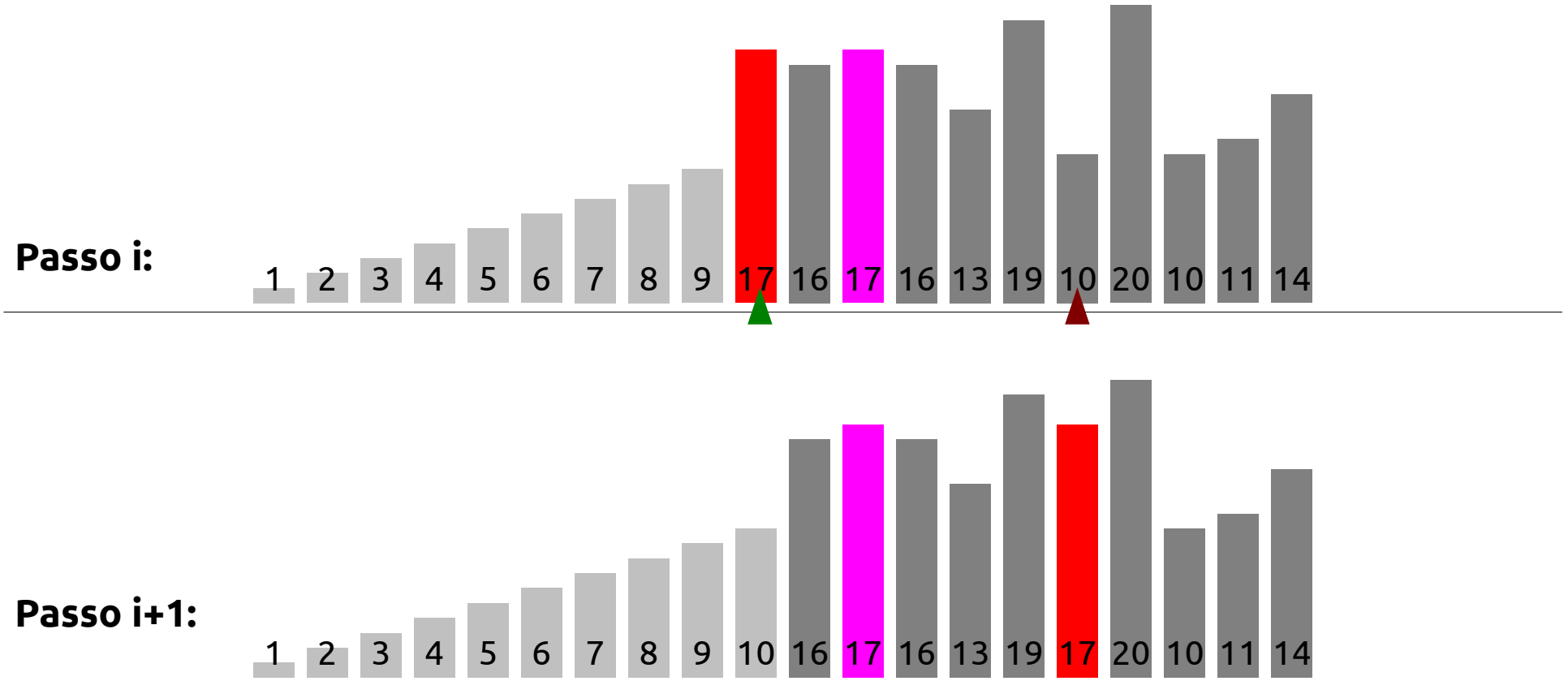
- Preservação da ordem relativa de elementos com chaves iguais



Porque o *SelectionSort* não é estável?

SelectionSort

- Não é estável:
  - No momento da troca pelo menor elemento da sequência não-ordenada, a ordem relativa entre chaves iguais pode ser quebrada



Priority *Queue – SelectionSort based*

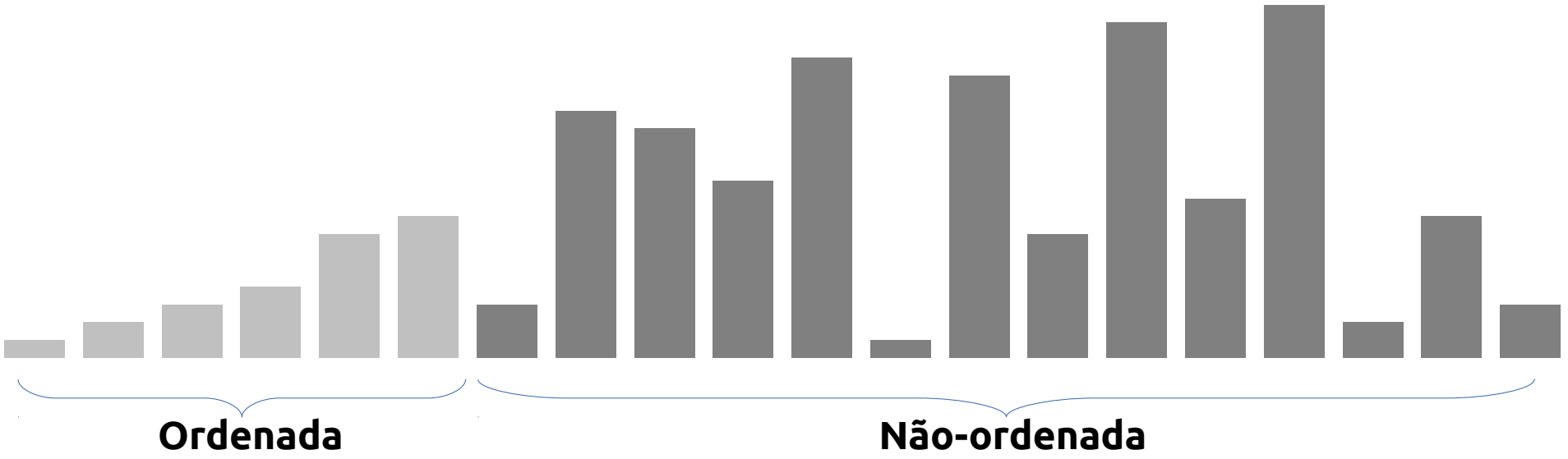
- Elementos ficam desordenados em relação a prioridade
  - "Rápida" inserção, "lenta" remoção
- Inserção:
  - Insere no início ou final do conjunto (qualquer lugar)
  - *O(1)*
- Remoção:
  - Percorre os elementos até encontrar o de menor/maior prioridade
  - *O(n)*

Priority Queue

- "Fila" com prioridade (conjunto de elementos)
  - Inserção de elementos no conjunto
  - Remoção do item com maior/menor prioridade
- Implementações:
  - *SelectionSort-based*
  - *InsertionSort-based*
  - *Heap-based*

InsertionSort

- Divisão dos dados em duas sequências: ordenada e não-ordenada
- Iteração: inserir o primeiro elemento da sequência não-ordenada na sequência ordenada



InsertionSort

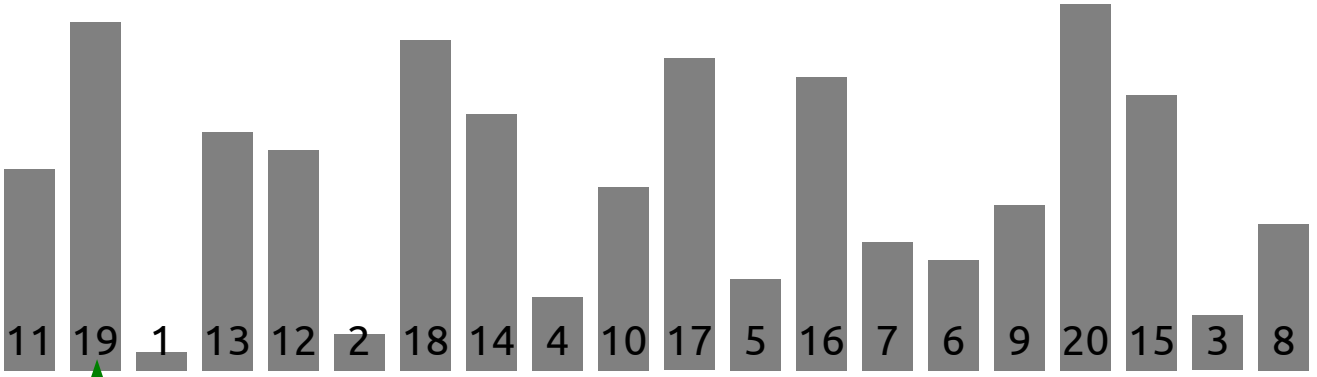
```
void insertion(Item vetor[], int n)
{
    for (int i = 1; i < n; i++)
    {
        int j = i;
        while ( j > 0 && vetor[j-1] > vetor[j])
        {
            swap(vetor[j-1], vetor[j]);
            j--;
        }
    }
}
```

i
controla a iteração, índice da sequência ordenada

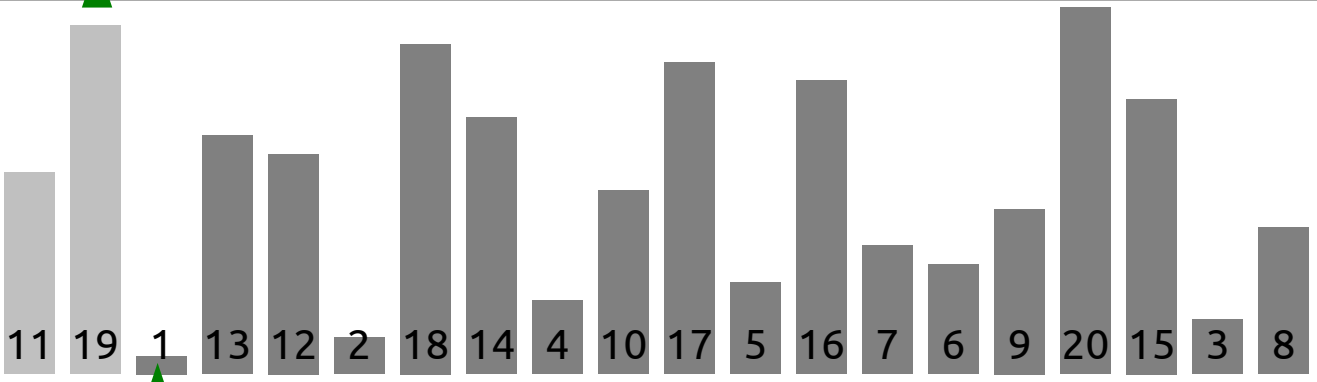
j
controla a inserção do elemento da sequência não-ordenada na sequência ordenada

```
void swap(Item &A, Item &B)
  { Item t = A ; A = B; B = t; }
```

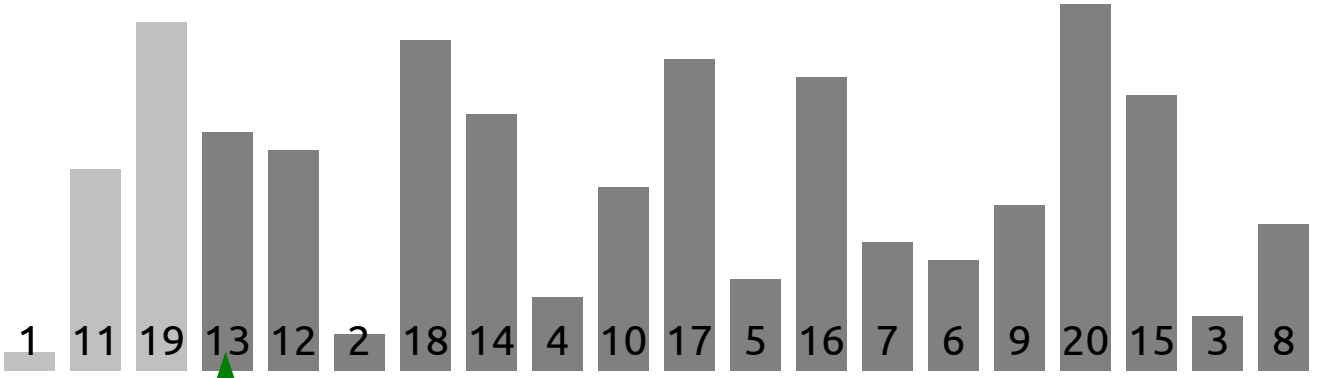
Entrada:



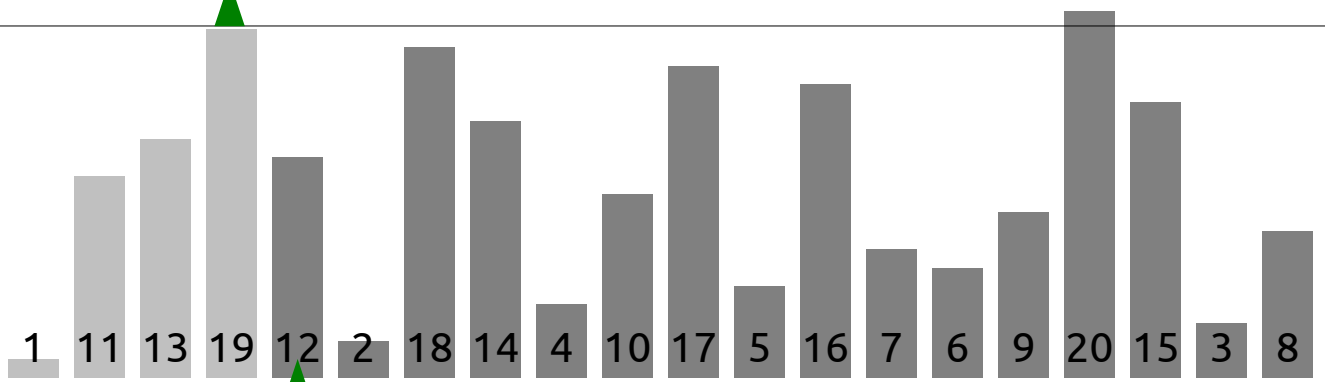
Passo 1:



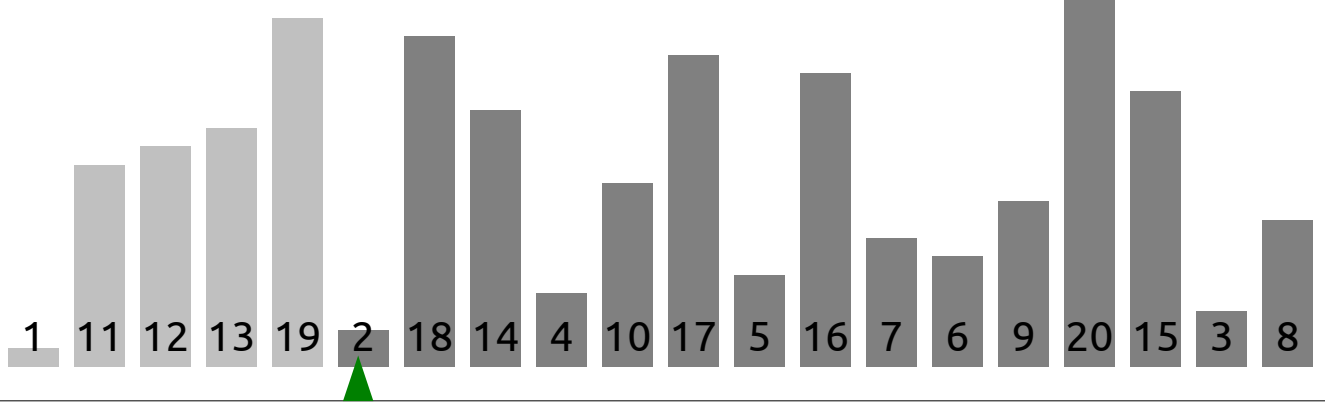
Passo 2:



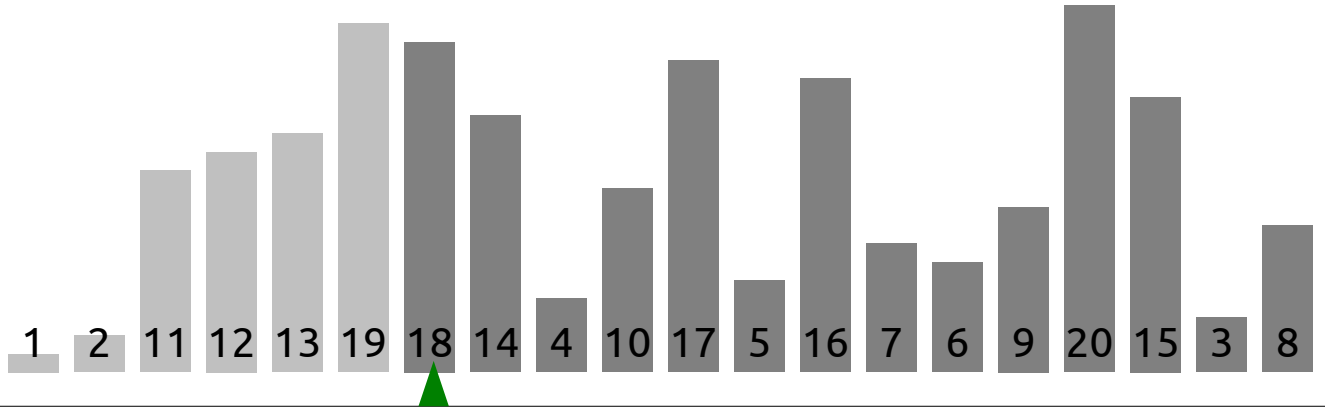
Passo 3:



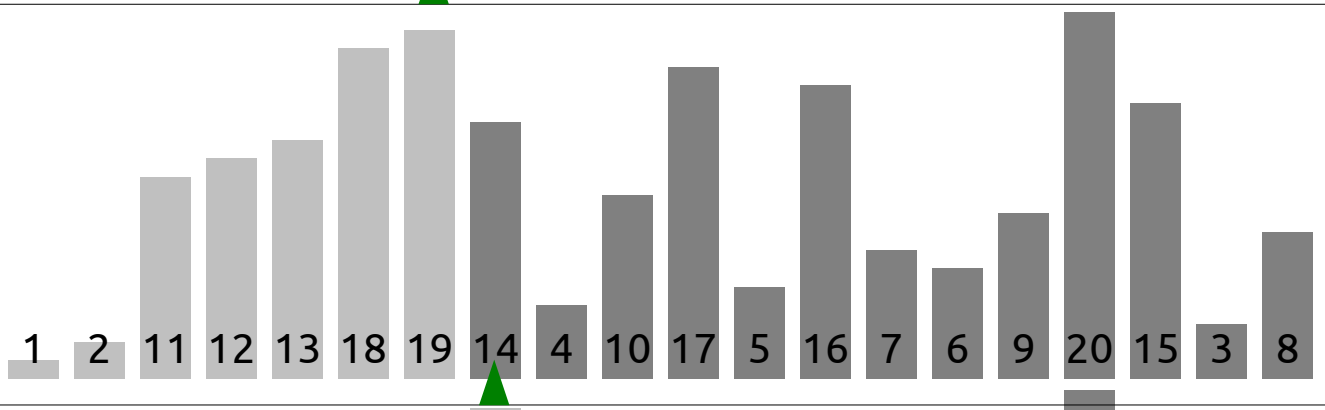
Passo 4:



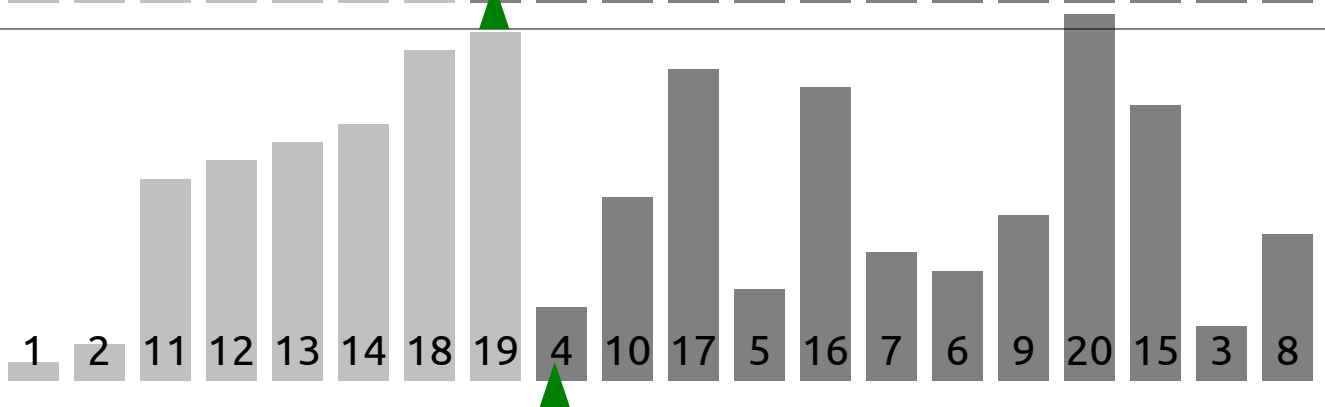
Passo 5:



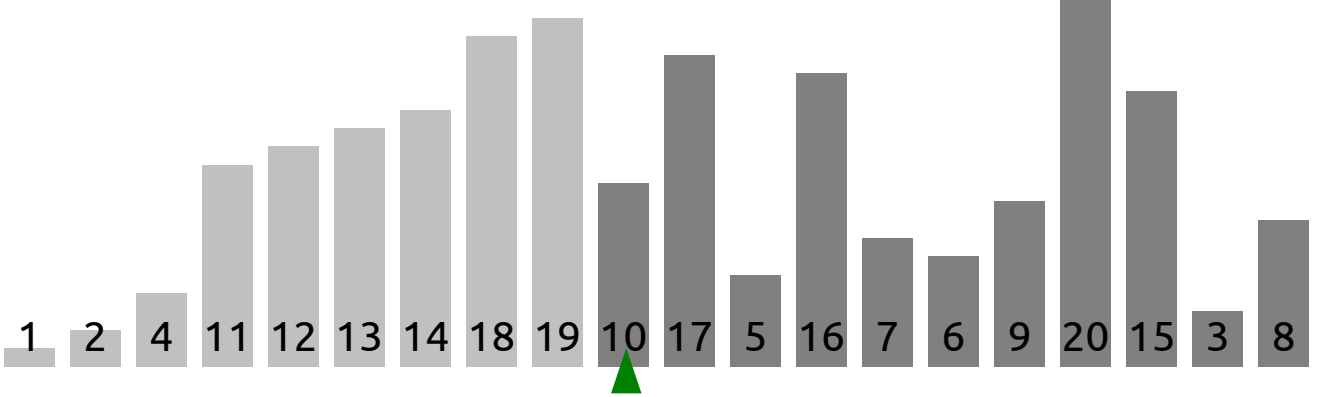
Passo 6:



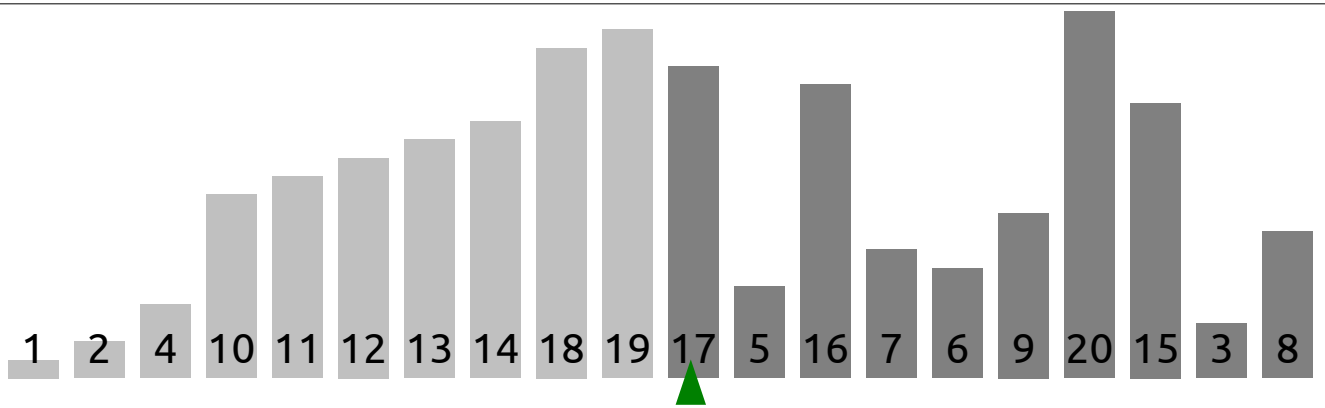
Passo 7:



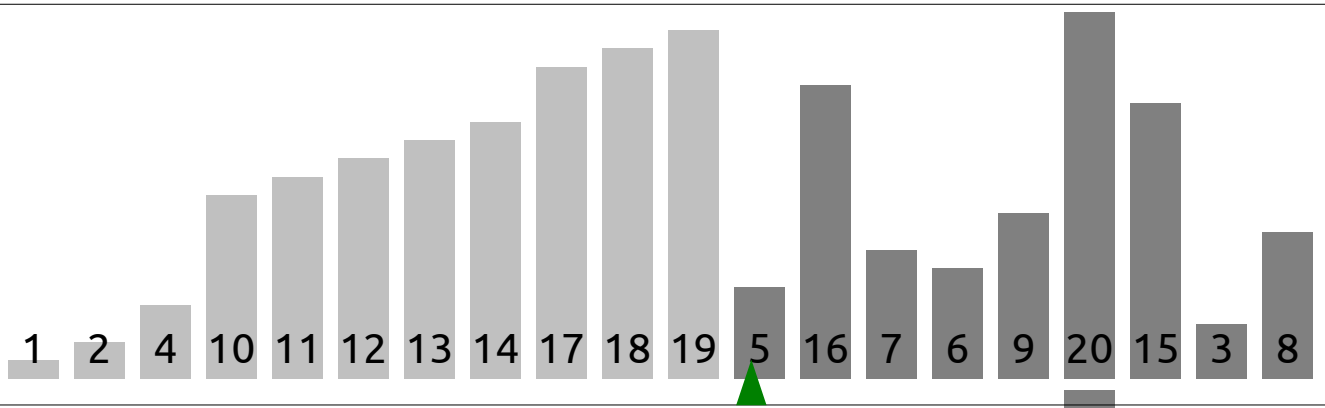
Passo 8:



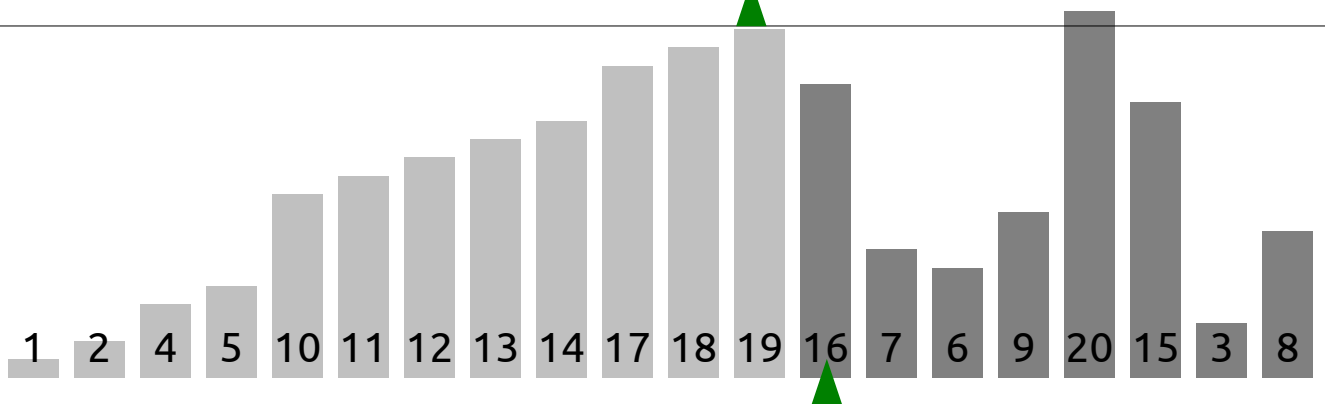
Passo 9:



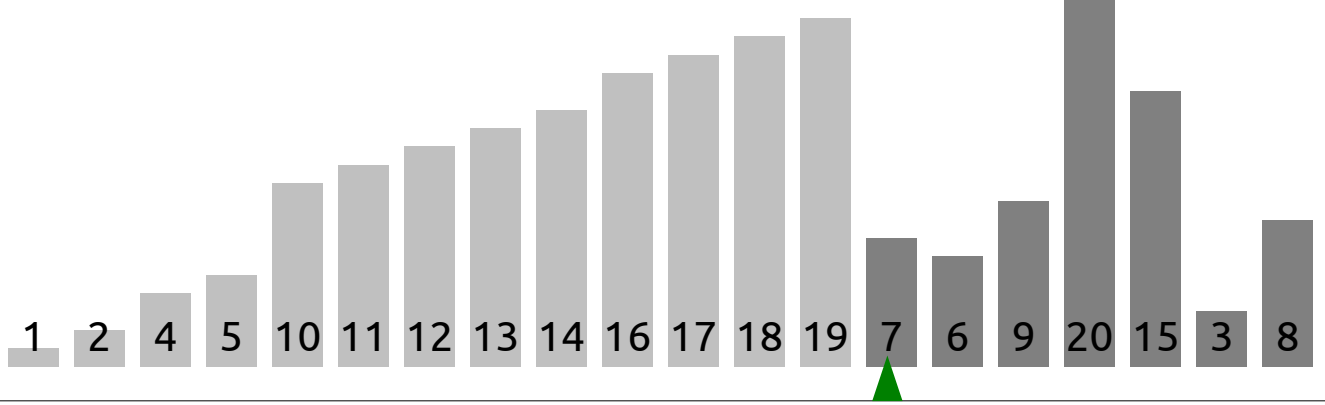
Passo 10:



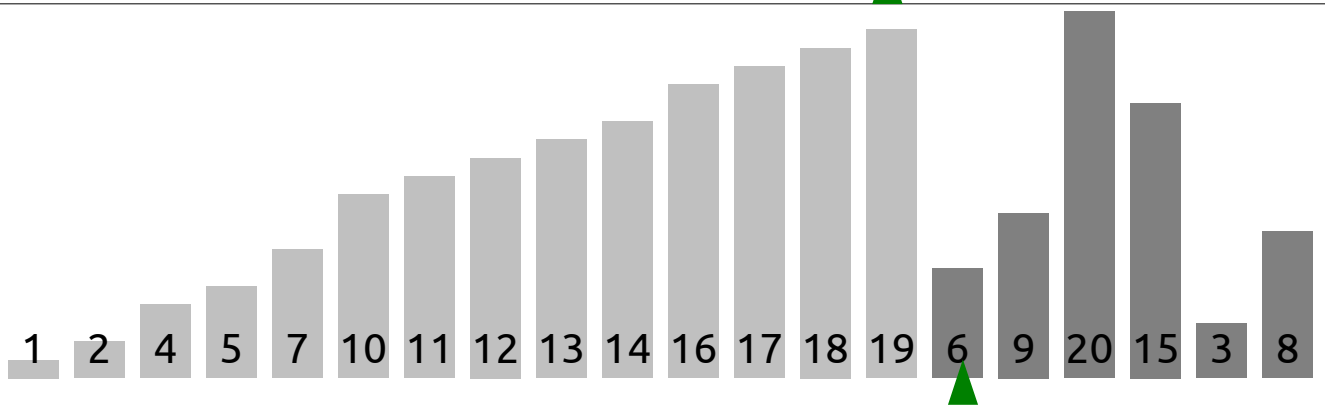
Passo 11:



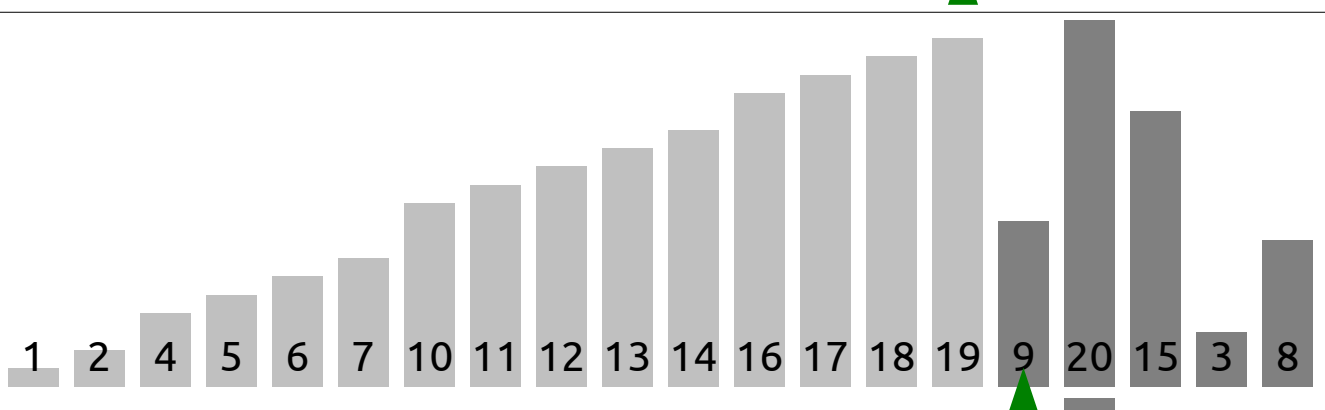
Passo 12:



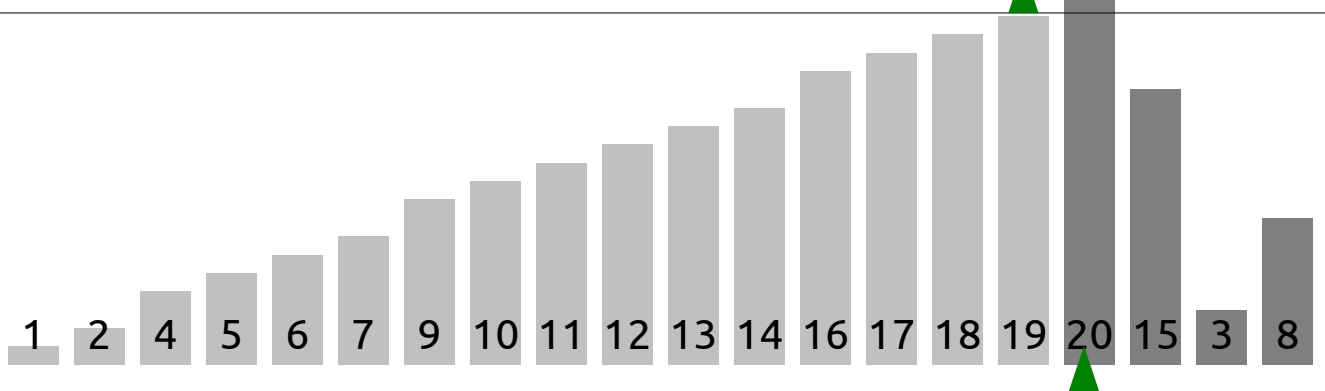
Passo 13:



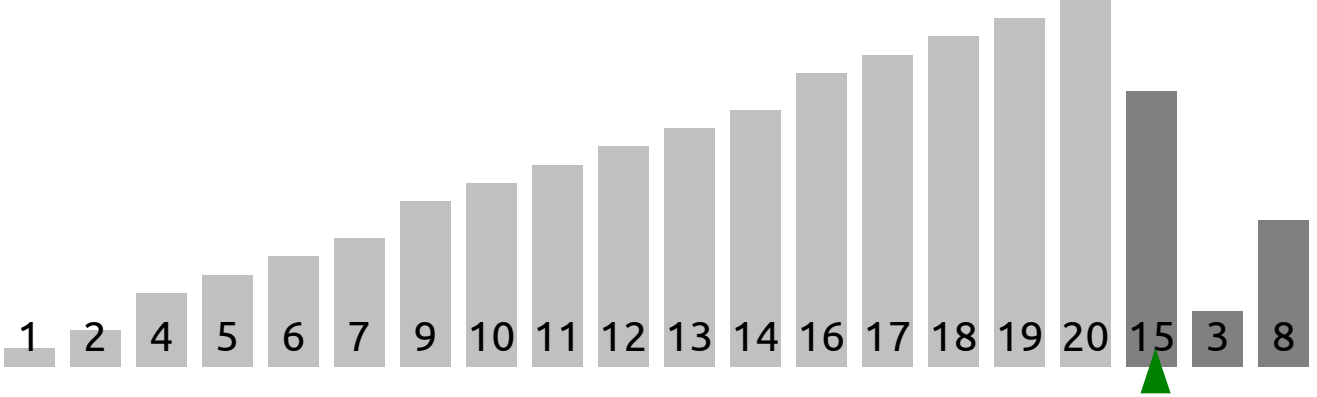
Passo 14:



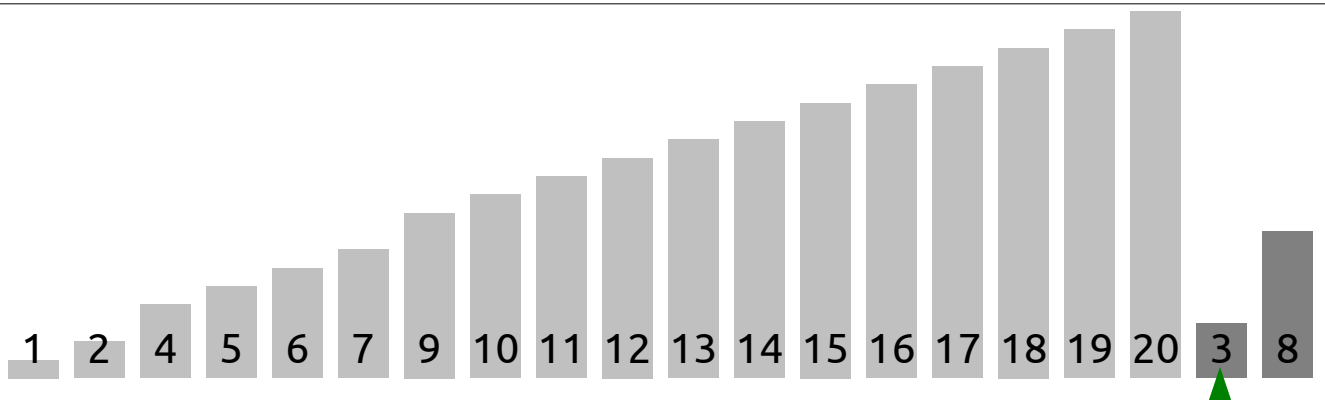
Passo 15:



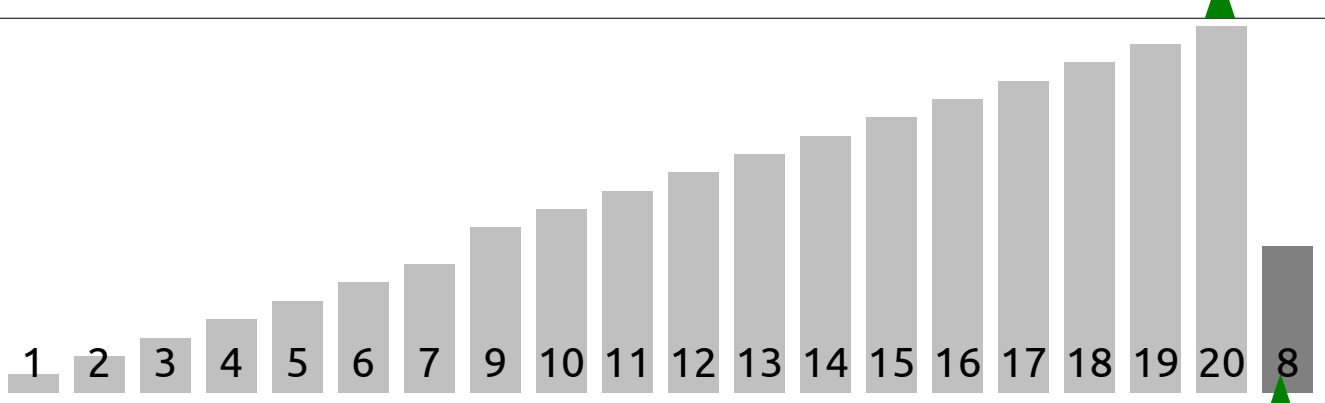
Passo 16:



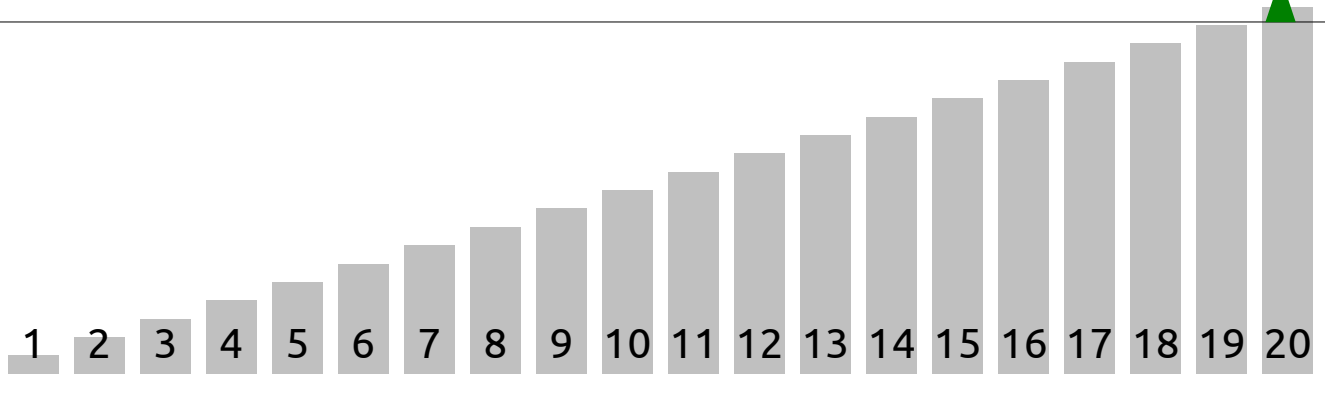
Passo 17:



Passo 18:



Passo 19:



InsertionSort

- Quantas comparações são executadas?
- Quantas trocas são executadas?
- É estável?
- Quantidade de memória?

```
void insertion(Item vetor[], int n)
{
     for (int i = 1; i < n; i++)
     {
          int j = i;
          while ( j > 0 && vetor[j-1] > vetor[j])
          {
               swap(vetor[j-1], vetor[j]);
               j--;
          }
     }
}
```

InsertionSort

- Melhor caso: vetor já ordenado
  - Comparações: **linear**
  - Trocas: **constante**

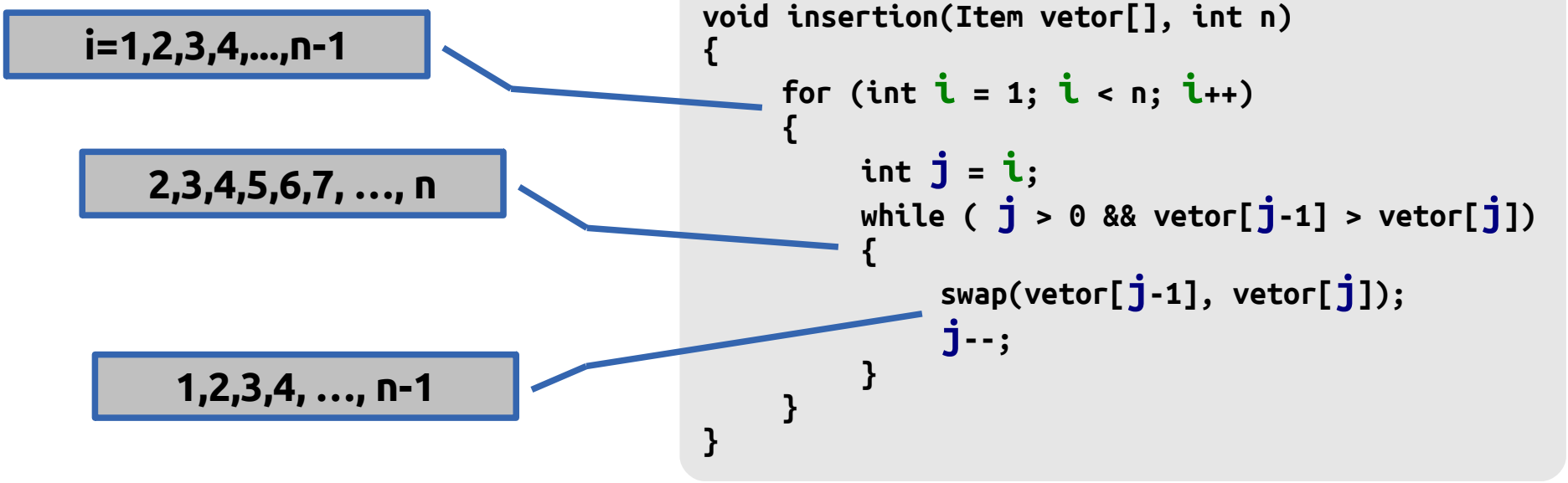
i=1,2,3,4,...,n-1

Nunca é executado!

```
void insertion(Item vetor[], int n)
{
     for (int i = 1; i < n; i++)
     {
          int j = i;
          while ( j > 0 && vetor[j-1] > vetor[j])
          {
               swap(vetor[j-1], vetor[j]);
               j--;
          }
     }
}
```

InsertionSort

- Pior caso: vetor inversamente ordenado
  - Comparações: **quadrático**
  - Trocas: **quadrático**



InsertionSort

- Os valores dos dados interferem na execução do algoritmo
- Crescimento do número de comparações em relação ao tamanho de entrada:
  - **linear** (no melhor caso)
  - **quadrático** (no pior caso)
- Crescimento do número de trocas em relação ao tamanho de entrada:
  - **constante** (no melhor caso)
  - **quadrático** (no pior caso)
- Crescimento do uso de memória em relação ao tamanho da entrada: **constante**

- O algoritmo é estável?

InsertionSort

- Se não conhecemos nada das possíveis entradas (valores aleatórios)
  - Assumimos a média, logo:
    - Comparações: **quadrático**
    - Trocas: **quadrático**
- Se conhecemos: depende!
  - Imagine entradas quase sempre ordenadas ...

- O algoritmo é estável? Sim! (Porque?)

Priority *Queue – InsertionSort based*

- Elementos ficam ordenamos pela prioridade
  - "Rápida" remoção, "lenta" inserção
- Inserção:
  - Percorre a fila até encontrar a posição correta e insere o elemento
  - *O(n)*
- Remoção:
  - Remove diretamente o primeiro da fila (menor/maior prioridade)
  - *O(1)*

Priority Queue

- "Fila" com prioridade (conjunto de elementos)
  - Inserção de elementos no conjunto (qualquer posição)
  - Remoção do item com maior/menor prioridade
- Implementações:
  - *InsertionSort-based*
  - *SelectionSort-based*
  - *Heap-based*

Priority *Queue –* ***Heap*** *based*

- Utilizando a estrutura ***heap***
- Inserção:
  - No final, atualizando a ***heap*** (*fixUp*)
  - *O(logN)*
- Remoção
  - Raiz, substituindo pelo último e atualizando a ***heap*** (*fixDown*)
  - *O(logN)*

Estrutura *heap*

- Árvore!
  - Critério de ordenação:
    - Max-Heap:
      - Elemento pai sempre maior ou igual aos filhos
    - Min-Heap:
      - Elemento pai sempre menor ou igual aos filhos
  - Chaves armazenadas nos nós
- Utilizaremos apenas:
  - Árvores **binárias** (até dois filhos)
  - **Completa**: elementos sem filhos apenas no último nível (e anterior, quando o último nível não está completo)
  - **Max-heap**

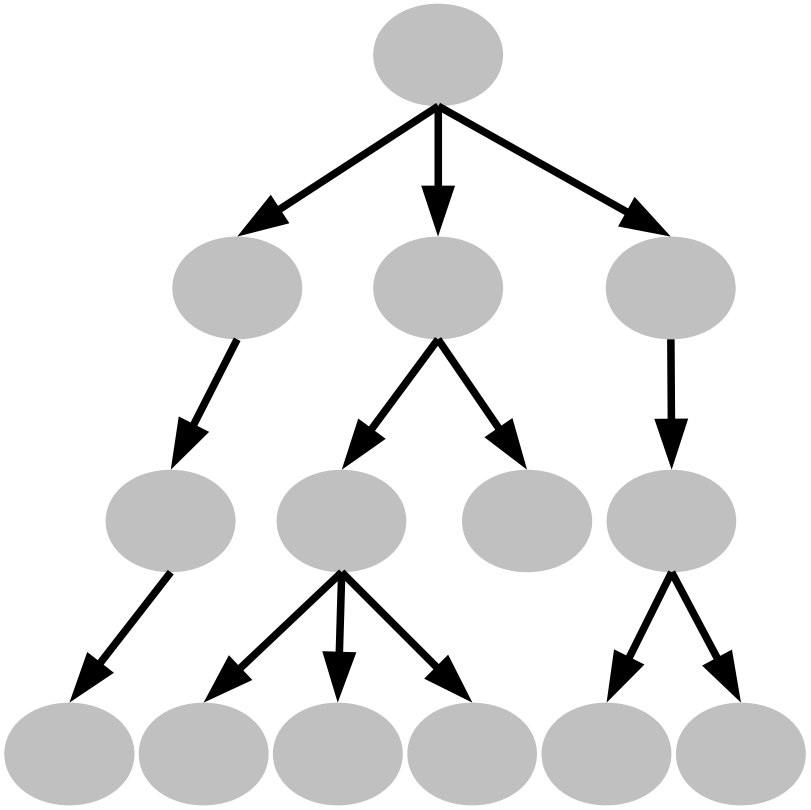
Aplicações diretas

- Fila com prioridade
  - Ordenação utilizando a estrutura ***heap***
  - Inserções e remoções diretamente na estrutura

- *Heap Sort*
  - Ordenação utilizando a estrutura ***heap***
  - Duas etapas:
    - Construção da estrutura max-heap
    - Ordenação pela concatenação dos valores máximo obtidos
      - Iteração: removendo um elemento (maior) por vez da heap

Árvores

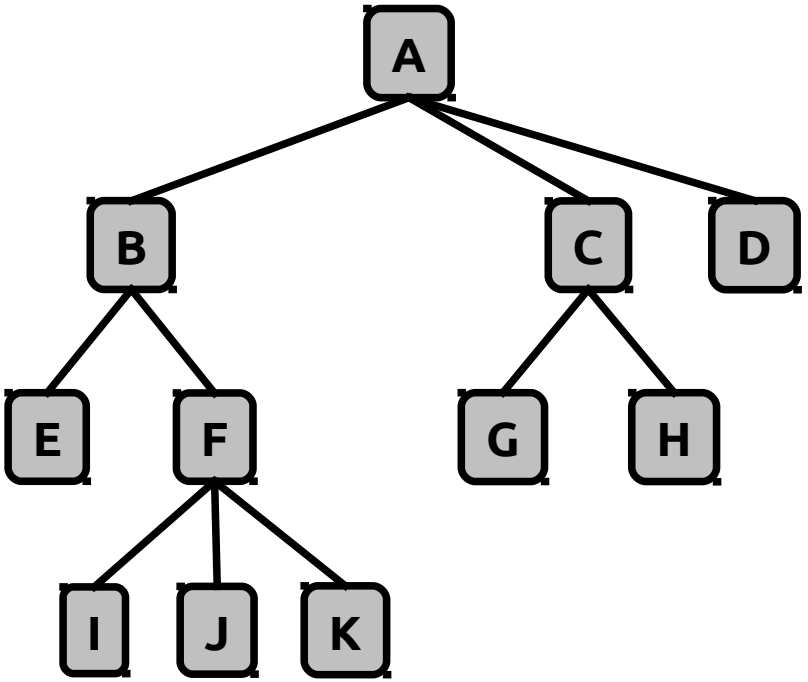
- Estrutura hierárquica e não-linear
- Definição recursiva:
  - Uma árvore pode ser vazia ou ser um elemento com no máximo duas (ou mais) árvores (chamadas de sub-árvores)
- Hierárquica:
  - Sub-árvores disjuntas
- Não-linear:
  - Um elemento pode possuir mais de um "próximo" imediato (diferente da lista)



Árvores

- Exemplos cotidianos:
  - Pastas ou diretórios
  - Herança em orientação à objetos
    - ops .. exceto herança múltipla!
  - Domínios da internet
  - ...

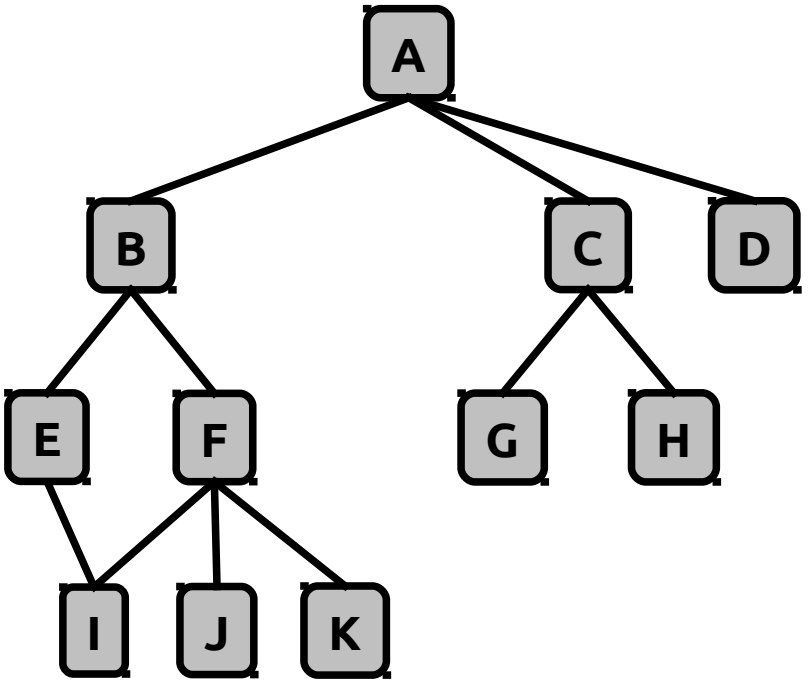
Árvores

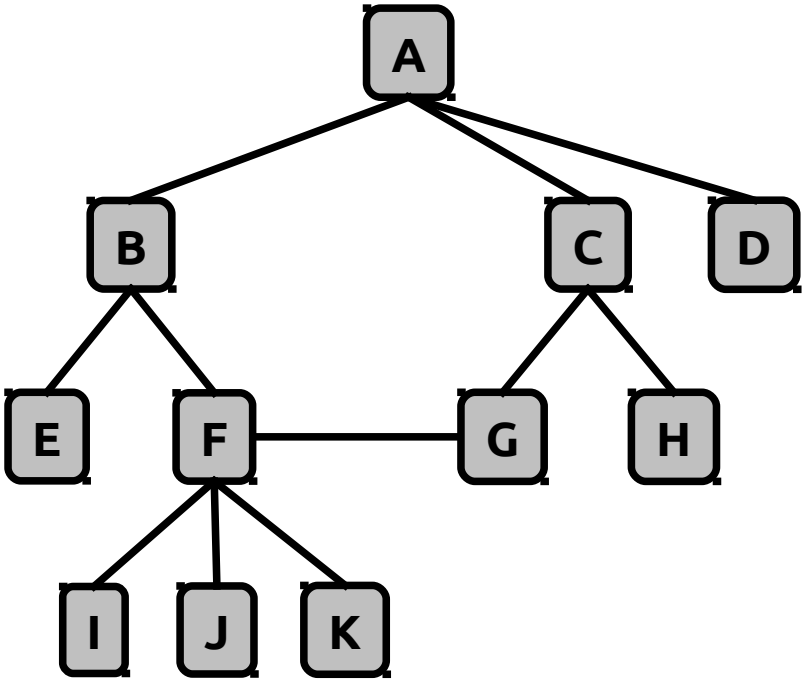


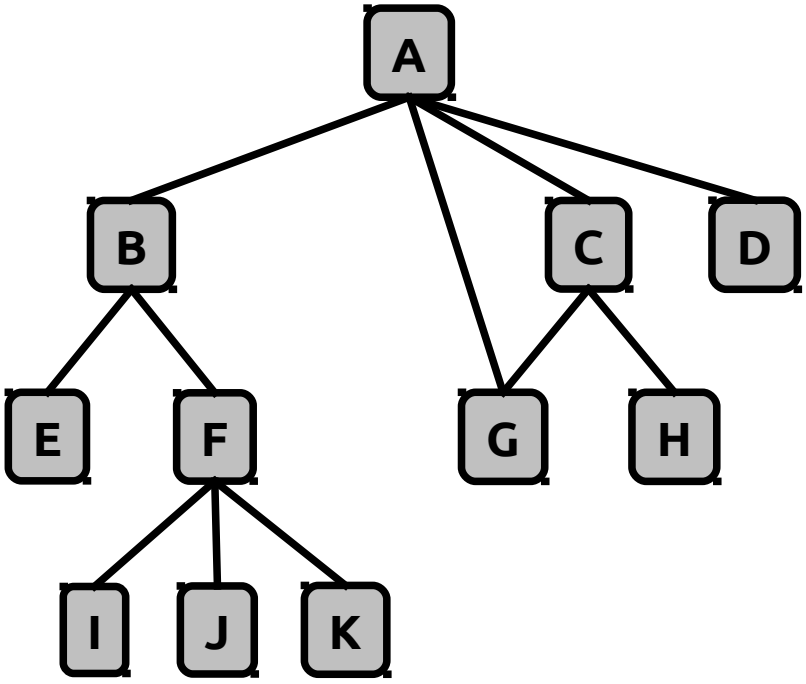
- Nomenclatura utilizada:
  - **A** é *pai* de **B**, **C** é *pai* de **G**
  - **J** é *filho* de **F**, **C** é *filho* de **A**
  - **A** é a *raiz* da árvore (nó sem pai)
  - **C** é *irmão* de **B**
  - **A**, **B**, **C** e **F** são nós *internos* (com pelo menos um filho)
  - **E**, **I**, **J**, **K**, **G**, **H** e **D** são nós *externos* ou *folhas* (sem filhos)
  - **B** é *ancestral* de **J**, **A** é *ancestral* de **H**
  - **I** é *descendente* de **A**, **H** é *descente* de **A**
  - *Sub-árvore*: nó e seus descendentes
  - **C** é *raíz* de uma sub-árvore
  - *Profundidade* de um nó: número de ascendentes
  - *Altura* de uma árvore (ou sub-árvore): maior profundidade
  - *Grau*: maior número de filhos

Árvores

- Não são árvores:





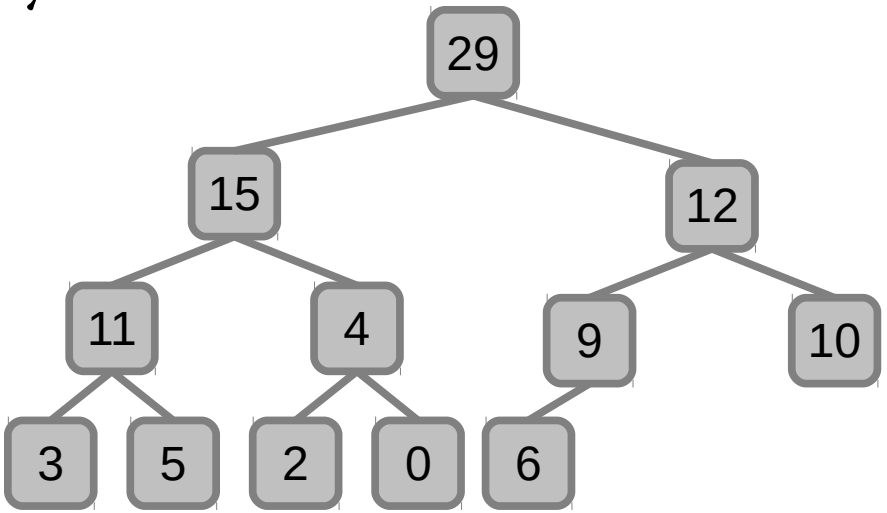


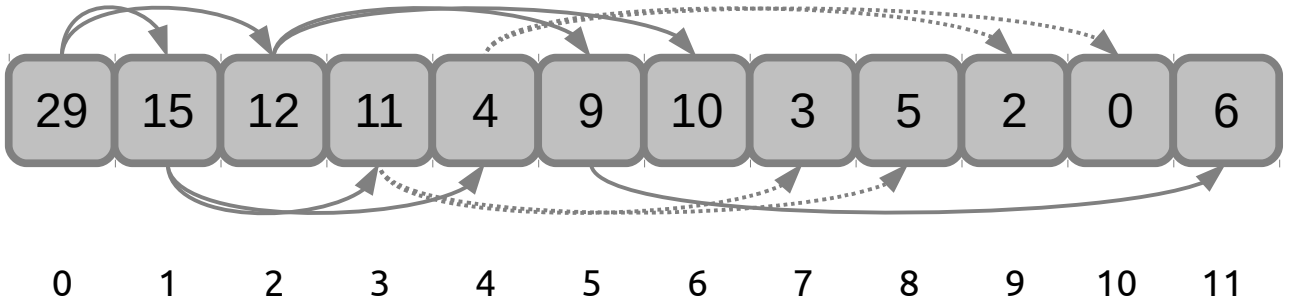
Estrutura *heap*

- Árvore:
  - Binária:
    - Grau 2, máximo de 2 filhos por nó
  - Completa:
    - Se a altura da árvore é *d*, cada nó folha está no nível *d* ou no nível *d-1*
      - Em outras palavras, a diferença de altura entre as sub-árvores de qualquer nó é no máximo 1
    - No caso da estrutura *heap*, os nós são alocados da esqueda para a direita
  - Critério de ordenação:
    - Max-Heap:
      - **Elemento pai sempre maior ou igual que os filhos (consequentemente, a todos os nós das suas sub-árvores)**
  - Chaves armazenadas diretamente nos nós

Estrutura *heap*

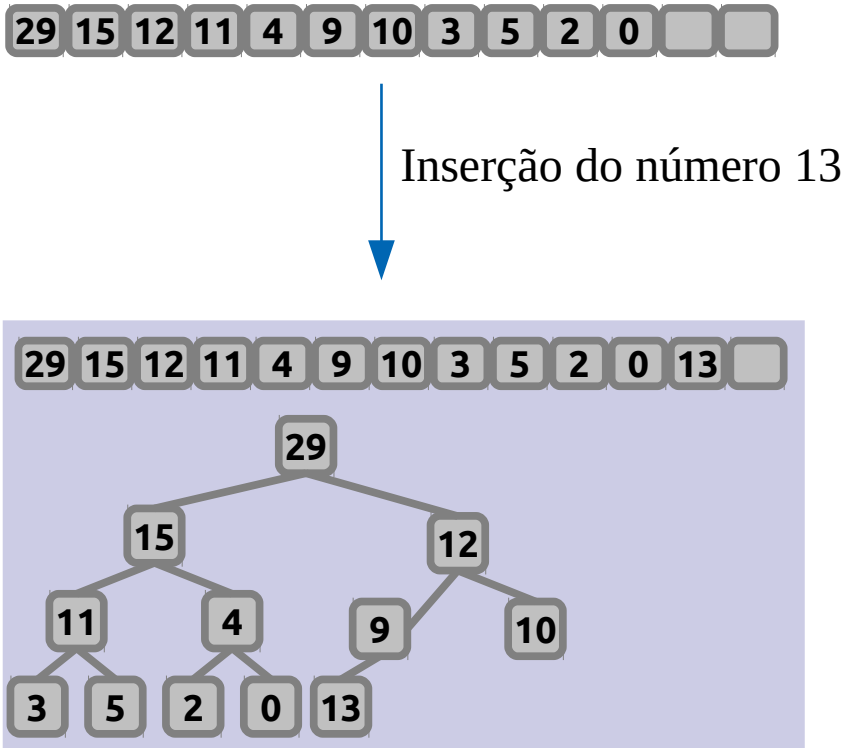
- Árvore binária completa:
  - Armazenamento direto em array!
  - Manipulação dos índices:
    - Pai: (**iFilho-1**)**/2**
    - Filho esquerda: **iPai*2 + 1**
    - Filho direita: **iPai*2 + 2**





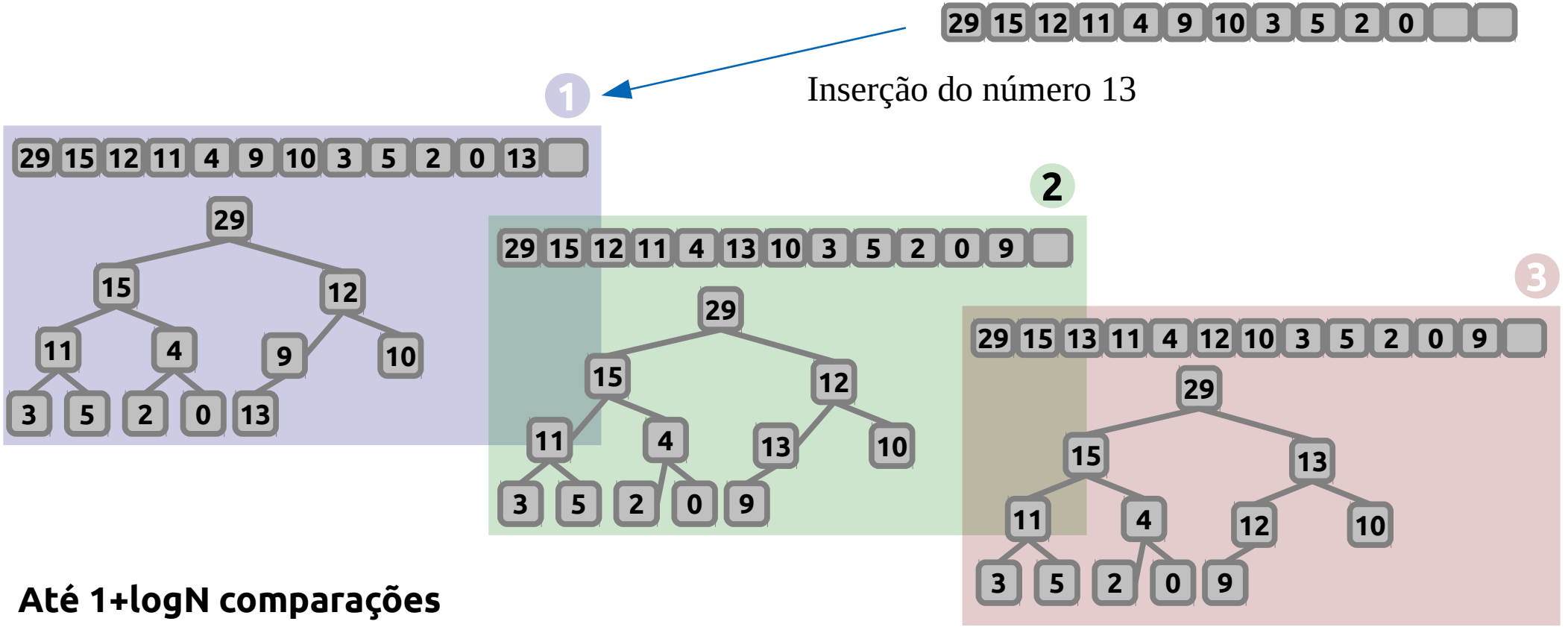
Estrutura *heap*

- Inserção:
  - Inserimos o novo elemento na última posição do vetor (árvore completa!)
  - Precisamos atualizar a estrutura *heap*, corrigindo as violações do critério que a define
    - Bottom-Up!



Manutenção da estrutura *heap*

- Elemento violando a condição **heap**
  - Valor maior que o pai
    - O elemento precisa "subir" na árvore
    - Bottom-Up heapify (*swim*)

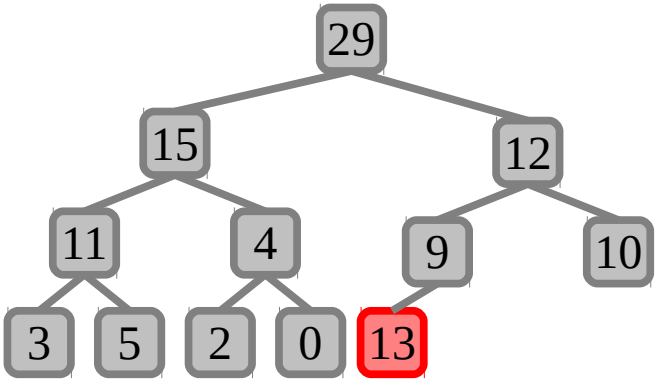


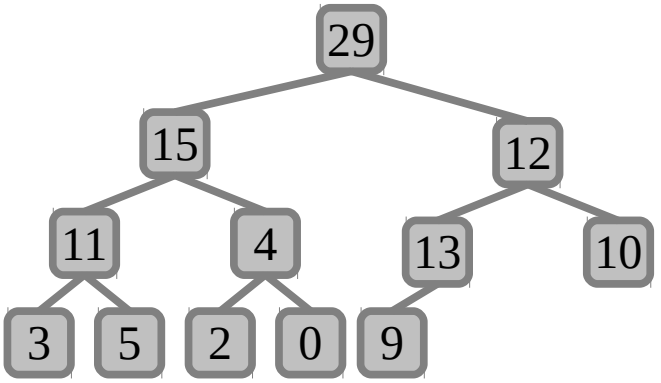
Até 1+logN comparações

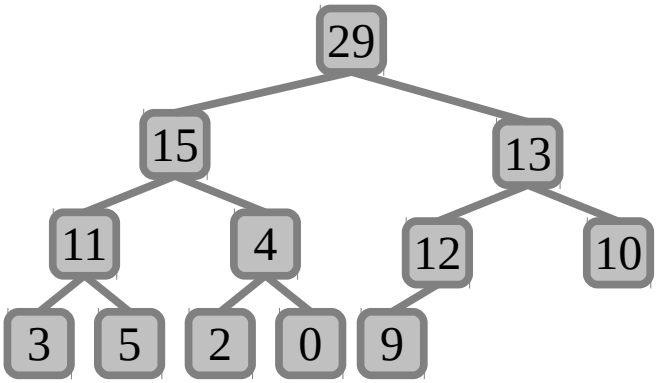
Bottom-Up Heapify

```
void fixUp(Item vetor[], int k)
{
    while (k > 0 && vetor[(k-1)/2] < vetor[k])
    {
        swap(vetor[k], vetor[(k-1)/2]);
        k = (k-1)/2;
    }
}
```

O(logN)

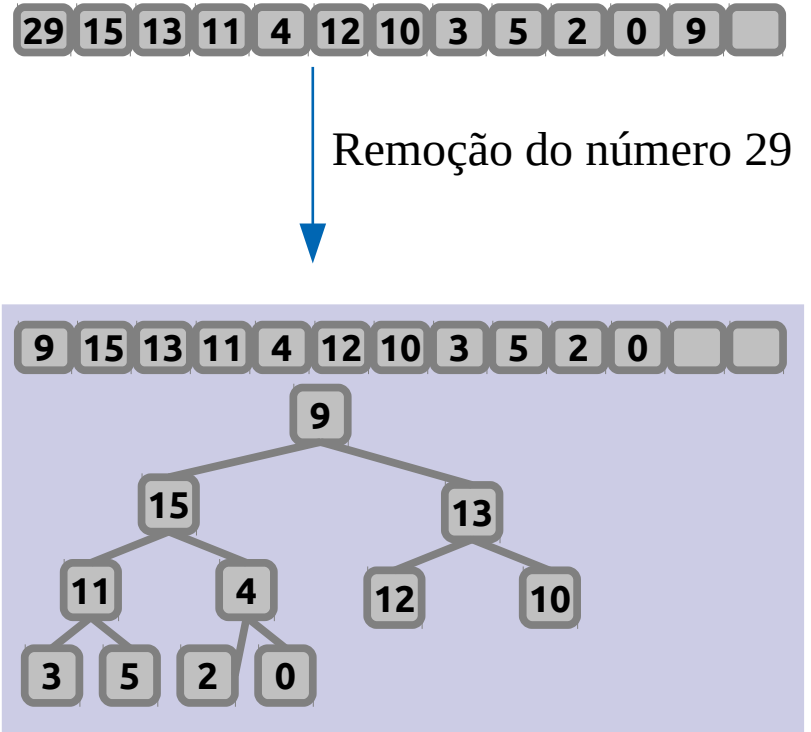






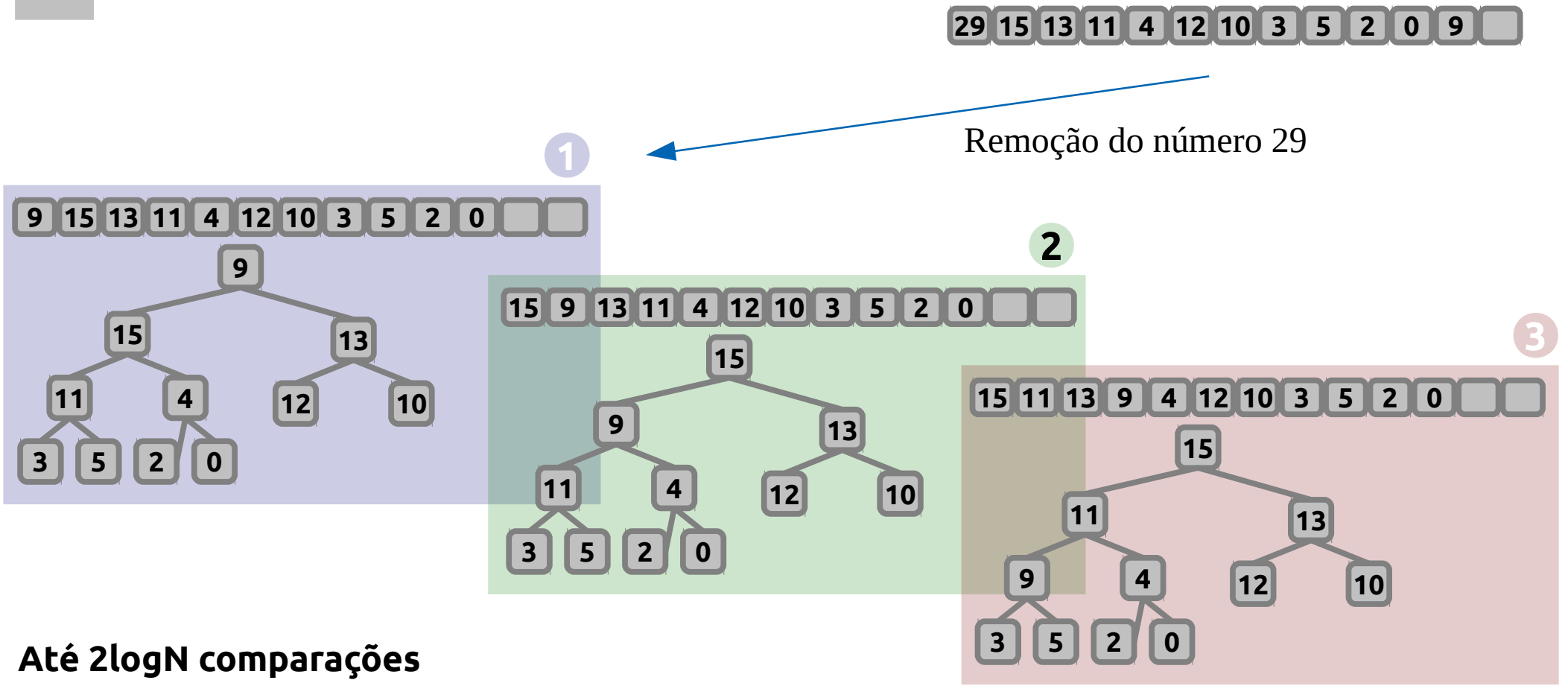
Estrutura *heap*

- Remoção:
  - Colocamos o último elemento do vetor na raiz
  - Diminuímos em 1 o tamanho do vetor
  - Precisamos atualizar a estrutura *heap*, corrigindo as violações do critério que a define
    - Top-Down!



Manutenção da estrutura *heap*

- Elemento violando a condição **heap**
  - Valor menor que os filhos (um ou dois)
    - O elemento precisa "descer" na árvore
    - Top-Down heapify (*sink*)

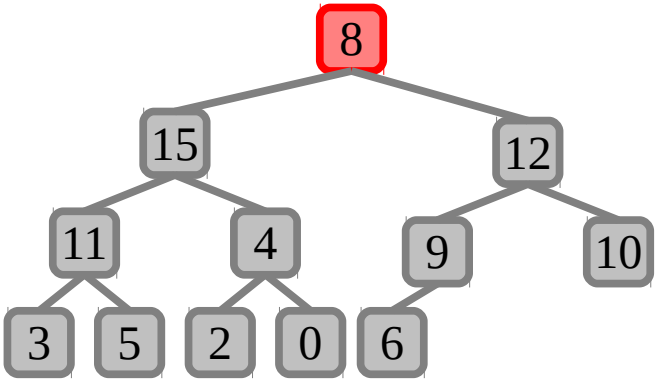


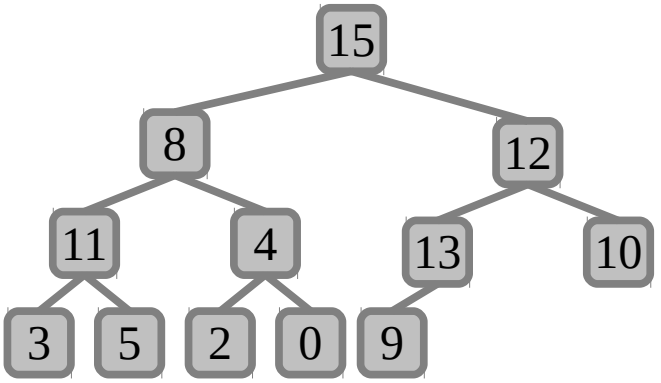
Até 2logN comparações

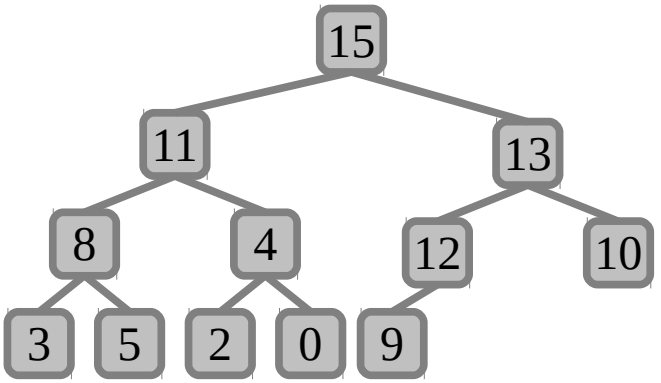
Top-Down heapify

```
void fixDown(Item vetor[], int k, int N)
{
    while (2*k+1 < N)
    {
        int j = 2*k+1;
        if (j < N-1 && vetor[j] < vetor[j+1])
            j++;
        if (!(vetor[k] < vetor[j]))
            break; //ooops
        swap(vetor[k], vetor[j]);
        k = j;
    }
}
```

O(logN)





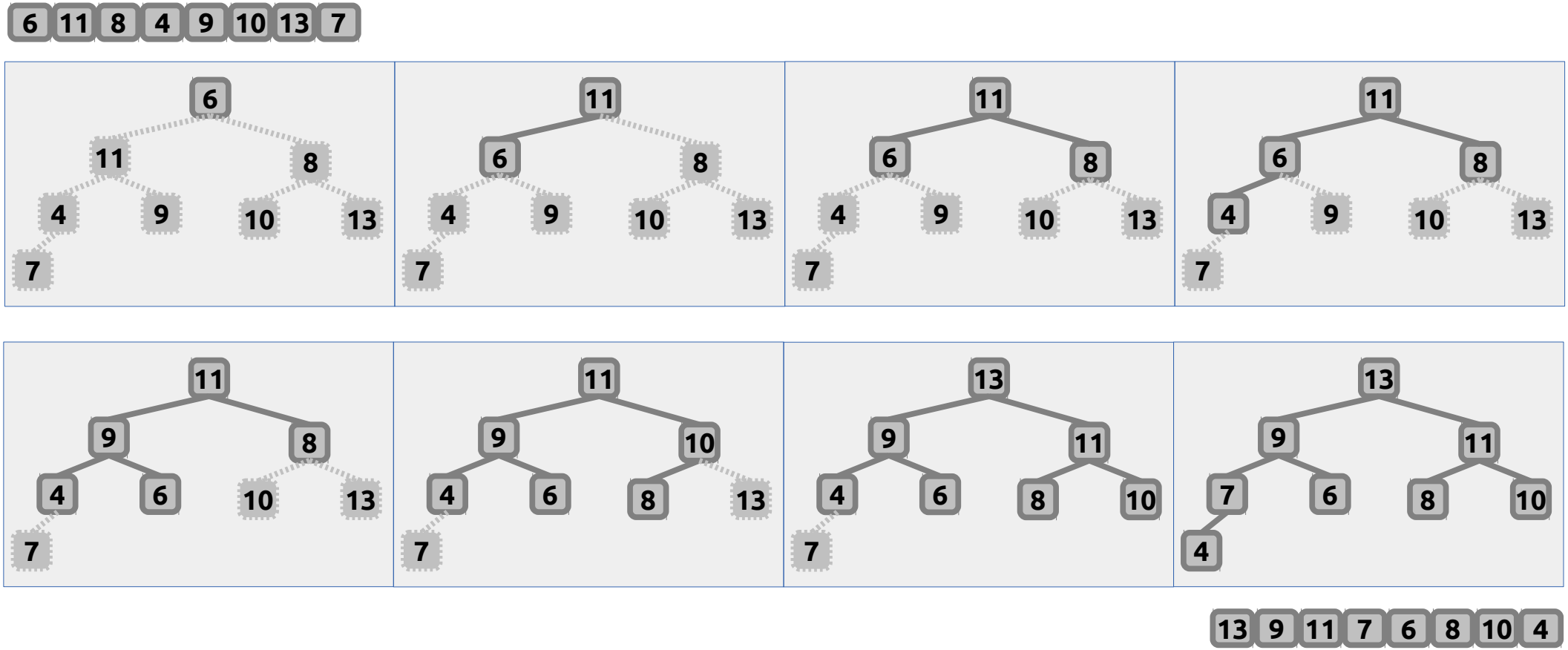


HeapSort

- Ordenação utilizando a estrutura ***heap***
- Duas etapas:
  - Construção da estrutura *max-heap*
  - Ordenação pela concatenação dos valores máximo obtidos
    - Iteração: removendo um elemento (maior) por vez da heap

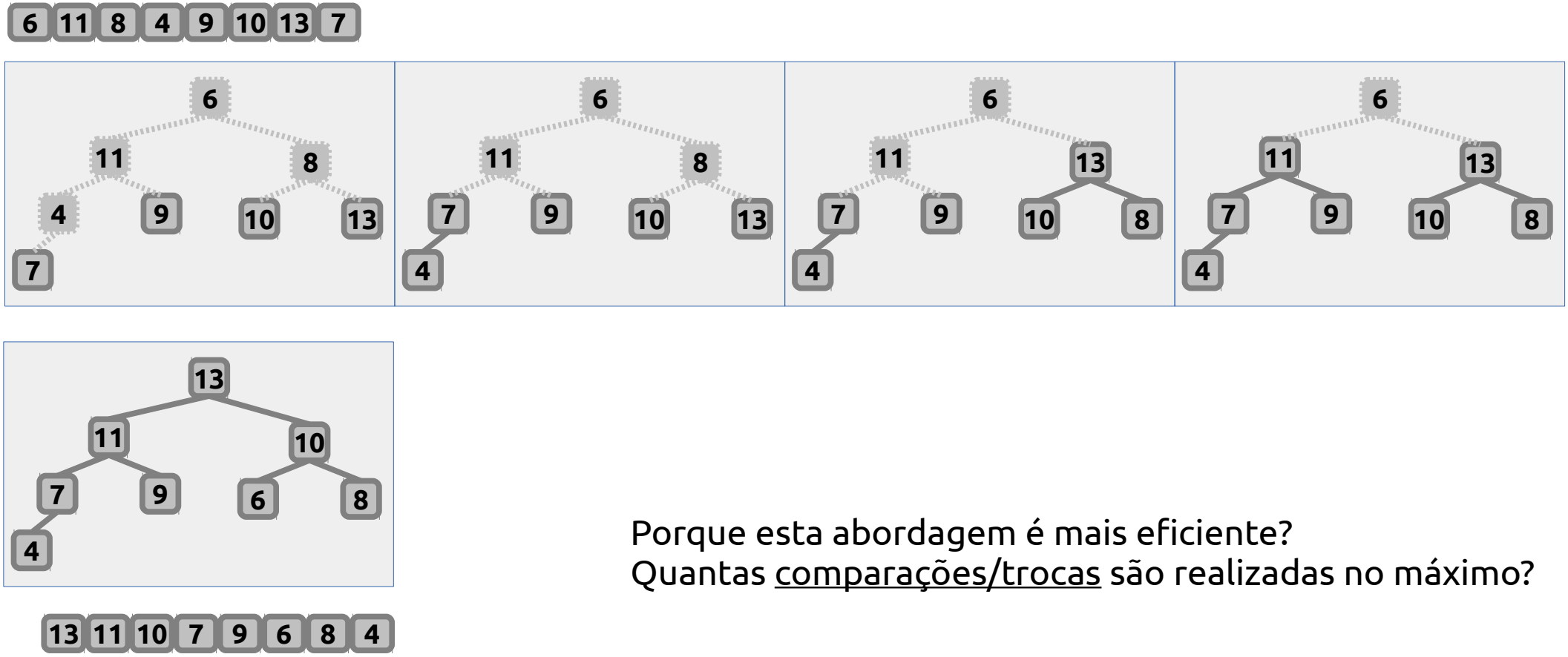
HeapSort - Construção da ***heap***

- Abordagem 1: da esquerda para a direita, adicionar um elemento por vez na ***heap*** à esquerda, utilizando o *bottom-up*



HeapSort - Construção da ***heap***

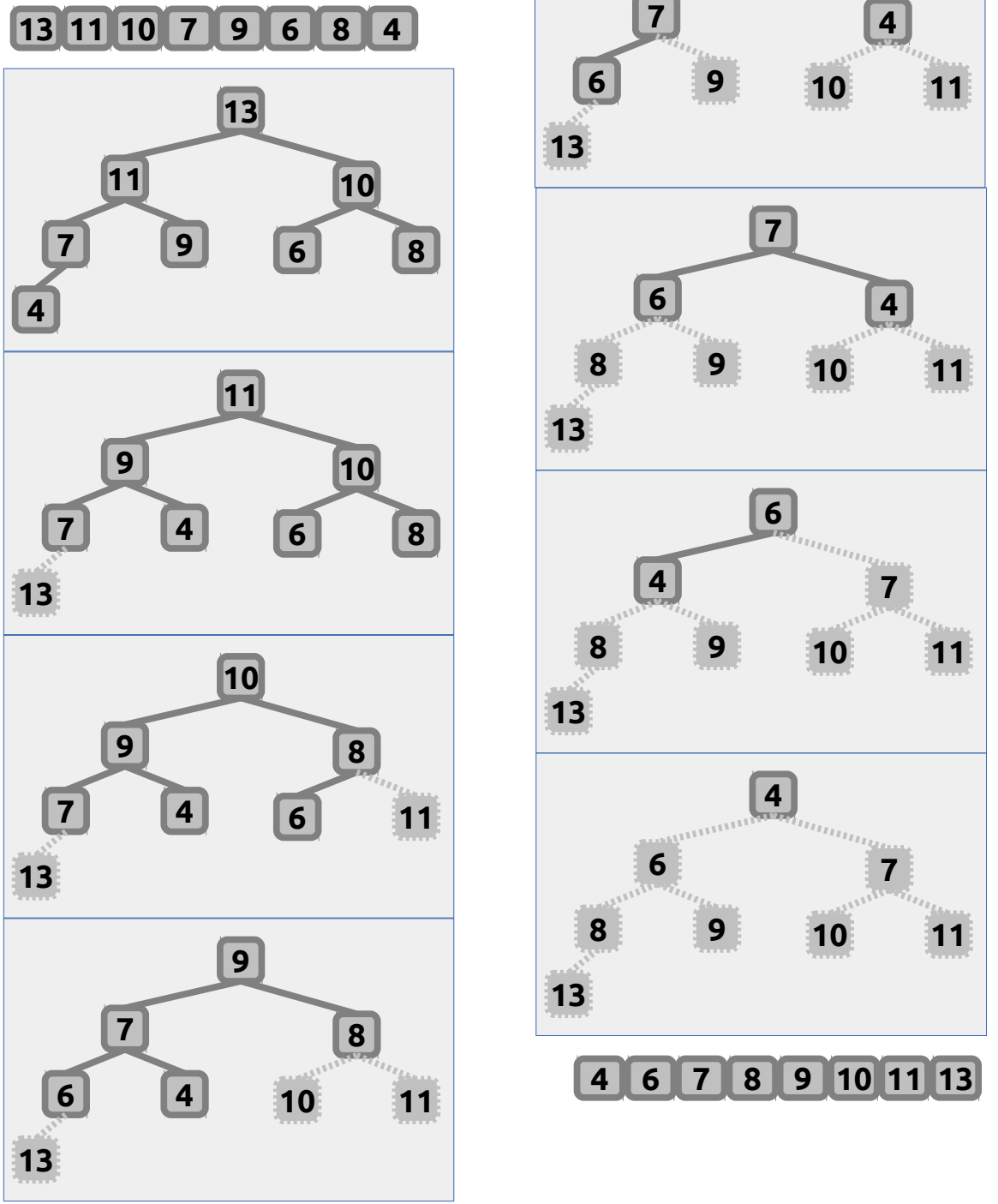
- Abordagem 2: da direta para a esquerda, construir sub-árvores e unir cada uma delas, utilizando o *top-down*



Porque esta abordagem é mais eficiente?
Quantas comparações/trocas são realizadas no máximo?

HeapSort

```
void heapsort(Item vetor[], int n)
{
    for (int k = n/2; k >= 0; k--)
        fixDown(vetor, k, n);

    while (n > 1)
    {
        swap(vetor[0], vetor[n-1]);
        n--;
        fixDown(vetor, 0, n);
    }
}
```



HeapSort

- Quantas comparações são executadas?
  - Comparações: **linear-logarítmico**
- É estável? Não
- Quantidade de memória necessária?
  - Constante → **In-place**

- Seria o algoritmo ideal (assintoticamente) …. mas não é estável e faz um uso muito ruim de memória *cache*

Priority *Queue –* ***Heap*** *based*

- Inserção:
  - No final, atualizando a ***heap*** (*fixUp*)
  - *O(logN)*
- Remoção
  - Raiz, substituindo pelo último e atualizando a ***heap*** (*fixDown*)
  - *O(logN)*

- Implementar em C uma fila com prioridade, utilizando a estrutura *heap*
  - Defina o TAD
  - Especifique no inicializador o comparador do dado abstrato
  - Utilize um vetor dinâmico redimensionável

Referências:

- Livro do Drozdek
- Livro do Cormen
- Livro do Robert Sedgewick
  - https://algs4.cs.princeton.edu