



Algoritmos e Estrutura de Dados

Aula 10: Algoritmos de Ordenação

Prof. Rodrigo Mafort

Listas Não Ordenadas e Ordenadas

Operação	Listas Não Ordenadas	Listas Ordenadas
Busca	$O(n)$: Busca sequencial	$O(\log_2 n)$: Busca binária
Inserção – Sem chaves repetidas	$O(n)$ – Busca para identificar se o elemento já está na lista $O(1)$ – Para inserir Total: $O(n)$	$O(\log_2 n)$ – Busca para identificar se o elemento já está na lista $O(n)$ – Para inserir Total: $O(n)$
Inserção – Com chaves repetidas	$O(1)$ – Inserção na primeira posição livre no final da lista	$O(\log_2 n)$ – Busca para a posição correta da inserção $O(n)$ – Para inserir Total: $O(n)$
Remoção	$O(n)$ – Busca para identificar a posição do elemento (e se ele está na lista) $O(1)$ – Para remover Total: $O(n)$	$O(\log_2 n)$ – Busca para identificar a posição do elemento (e se ele está na lista) $O(n)$ – Para remover Total: $O(n)$

Listas Ordenadas

- Como construir uma lista ordenada a partir de uma lista não ordenada?
- Duas possibilidades:
 - Construir uma lista ordenada, inserindo os elementos da lista um de cada vez.
 - Ordenar a lista usando Algoritmos de Ordenação

Opção 1: Construir uma lista ordenada

```
void Transformar(int L[], int LOrd[], int tam)
{
    int tamOrd = 0;
    for (int i = 0; i < tam; i++)
        InserirOrdenado(LOrd, &tamOrd, L[i]);
}
```

- São realizadas n inserções em uma lista ordenada.
- Cada inserção, tem custo $O(n)$
- Logo: a complexidade desse método é $O(n^2)$

Opção 2: Algoritmos de Ordenação

Existe um número muito grande de métodos de ordenação:

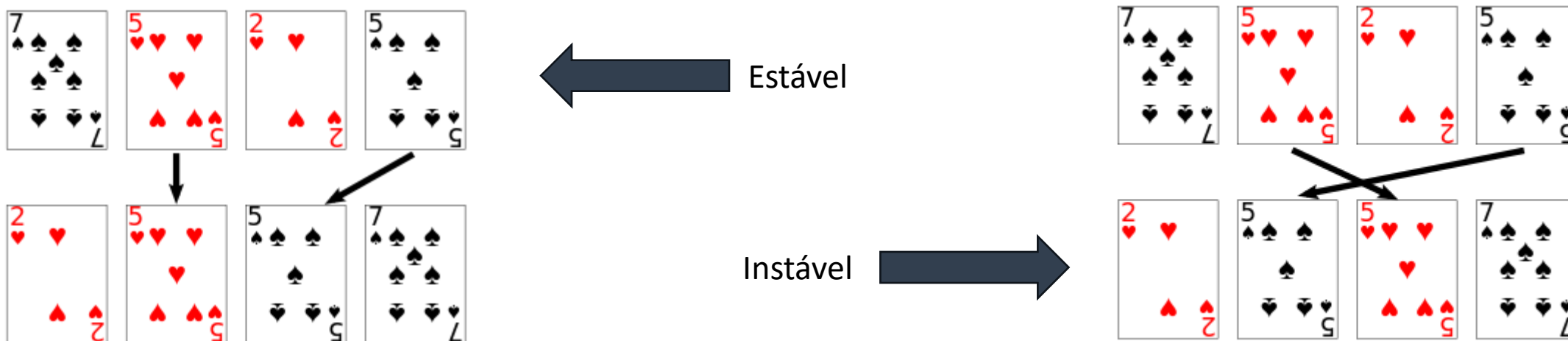
- Bogo Sort
- Bubble Sort
- Selection Sort
- Insertion Sort
- Cocktail Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Bucket Sort
- E muitos outros

Antes de continuar...

- Precisamos falar sobre alguns conceitos sobre algoritmos de ordenação:
 - Algoritmos de Ordenação Estável
 - Algoritmos de Ordenação *In-place*
 - Algoritmos de Ordenação Online

Algoritmo de Ordenação Estável

- Um algoritmo de ordenação é dito estável se ele retorna elementos iguais na mesma ordem em que aparecem na entrada. Isso é: o algoritmo não muda a ordem de elementos de mesma chave na lista. A ordem permanece a da entrada.
- Considere um algoritmo para ordenar cartas pelo número, independente do naipe.



Algoritmo de Ordenação *In-place*

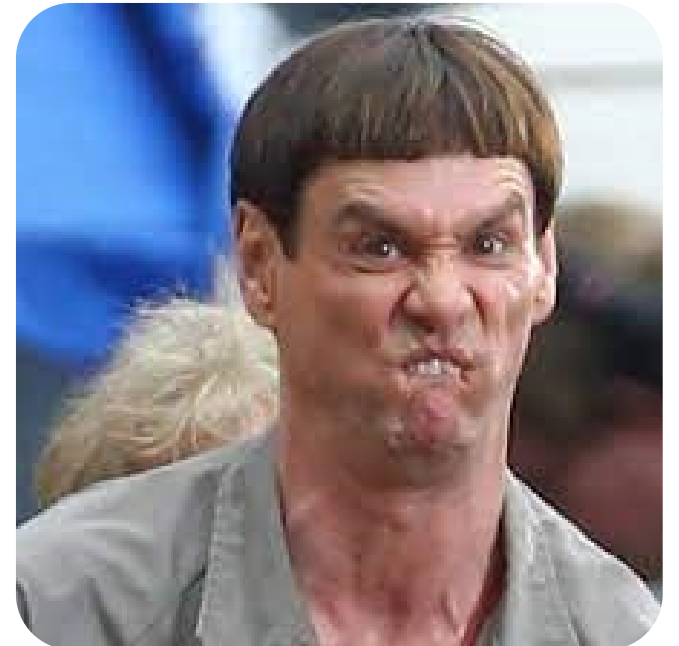
- Algoritmo de ordenação *in-place* operam diretamente na estrutura de dados de entrada sem exigir espaço extra proporcional ao tamanho de entrada.
- O algoritmo altera os elementos na própria lista, sem criar cópias da estrutura de dados fornecida.
- Essa característica permite avaliar também a complexidade de espaço do algoritmo. Isto é, seu uso de memória.
- Vale observar que o uso de memória para controlar as recursões pode ou não ser contabilizado nesse quesito. Depende da implementação: se essa memória armazenar dados da lista (por exemplo, um valor que deve ocupar uma determinada posição), então ela deve ser contabilizada e o algoritmo é *out-of-place*.

Algoritmos de Ordenação Online

- Algoritmos online (de ordenação ou não) são métodos capazes de lidar com os dados na ordem e no momento em que eles são fornecidos. Isto é, eles operam sem ter toda a entrada disponível desde o início. A medida que os dados são recebidos, o algoritmo efetua seu processamento.
- Algoritmos de ordenação online são capazes de recuperar a ordenação da lista em tempo linear ($O(n)$) a cada novo elemento.

Bogo Sort

- Ideia básica: Está ordenado?
 - Sim: Fim do Algoritmo
 - Não: "Bagunce" os números de forma aleatória
- O pior algoritmo de ordenação...
- Conta apenas com a sorte
- Complexidade de Pior Caso: $O((n + 1)!)$
- Complexidade do Melhor Caso: $O(n)$



Bogo Sort

- Complexidade de pior caso: $O(n!)$
 - Estável: Não
 - *In-place*: Sim
 - Online: Não
-
- Esse é o pior algoritmo de ordenação possível.
 - Apenas para ilustrar a complexidade.

Bubble Sort – V1

- O método mais simples de ordenação
- Ideia: Vamos percorrer a lista, trocando elementos fora de ordem.
No pior caso, vamos precisar de n passagens pela lista

```
void bubbleSort(int L[], int tam)
```

```
{
```

```
    for (int i = tam - 1; i >= 0; i--)  
        for (int j = 0; j < i; j++)  
            if (L[j] > L[j+1])  
                Trocar(&L[j], &L[j+1]);
```

```
}
```

Qual é a complexidade?

Repete $n - 1$ vezes

Repete n vezes

Operação Dominante:

Comparação

$O(n^2)$

Bubble Sort – V2

- O método mais simples de ordenação
- Ideia: Vamos percorrer a lista, trocando elementos fora de ordem. Paramos quando nenhuma troca for realizada

```
void bubbleSort(int L[], int tam)
{
    int ord = 0;
    while (ord == 0)
    {
        ord = 1;
        for (int i = 0; i < tam; i++)
            if (L[i] > L[i+1])
            {
                Trocar(&L[i], &L[i+1]);
                ord = 0;
            }
    }
}
```

Qual é a complexidade?

No pior caso, repete n vezes

Repete n vezes

Operação Dominante: Comparação

$O(n^2)$

Mas e se a lista já estiver ordenada?

Bubble Sort

- Complexidade de pior caso:
 - Versão 1: $O(n^2)$
 - Versão 2: $O(n^2)$
- Estável: Sim – Como a ordenação começa da esquerda para direita e dois elementos de mesma chave não são trocados, a ordem inicial entre eles permanece.
- *In-place*: Sim
- Online: Não – Se um novo elemento for inserido, ele precisa ser movido ao longo de várias iterações uma posição à frente por iteração.

Selection Sort

- O método de ordenação por seleção é o mais natural dentre os métodos:
- Selecione o menor (ou o maior) elemento.
- Mova para o início (ou para o final) da lista.
- Continue o processo para os demais elementos.

Selection Sort

```
void SelectionSort(int L[], int tam)
{
    for (int i = 0; i < tam - 1; i++)
    {
        int posMin = i;
        for (int j = i+1; j < tam; j++)
            if (L[j] < L[posMin])
                posMin = j;

        Trocar(&L[i], &L[posMin]);
    }
}
```

Qual é a complexidade?

$O(n^2)$

Selection Sort

- Complexidade de pior caso: $O(n^2)$
- Estável: Sim.
- *In-place*: Sim.
- Online: Não – O algoritmo precisa identificar na i -ésima iteração quem é o i -ésimo menor elemento. Se a lista não está completa, isso não é possível.

Insertion Sort

- O algoritmo de ordenação por inserção também é intuitivo:
 - Dado um elemento, varremos a lista buscando um elemento maior do que o atual. Assim que encontramos, inserimos o elemento antes desse maior.
- O método aplica uma outra ideia:
 - Começamos a lista com 1 elemento. Ele está ordenado.
 - Passamos para o segundo. Ele é inserido no lugar, resultando em uma lista com dois elementos ordenada.
 - Passamos para os próximos elementos: Eles são inseridos no lugar correto e resultam em uma lista com +1 elemento.
 - Observe que a lista vai “crescendo” a medida que novos elementos são inseridos. E essa lista está sempre ordenada.
- Esse método aplica a ideia da construção de uma lista ordenada através de n operações de inserção.

Insertion Sort

```
void InsertionSort(int L[], int tam)
{
    for (int i = 1; i < tam; i++)
    {
        int pos = 0;
        while (pos < i && L[i] > L[pos])
            pos++;

        int aux = L[i];
        for (int j = i; j > pos; j--)
            L[j] = L[j-1];

        L[pos] = aux;
    }
}
```

Qual é a complexidade?

$O(n^2)$

Insertion Sort

- Complexidade de pior caso: $O(n^2)$
- Estável: Sim.
- *In-place*: Sim.
- Online: Sim – O novo elemento vai ser movido para o seu lugar considerando a lista já fornecida. Isso pode ser feito em $O(n)$, tal como na inserção em listas ordenadas.
- Vale observar que o Insertion Sort é equivalente à primeira forma de transformar listas não ordenadas em listas ordenadas: uma chamada do algoritmo se inserção para cada elemento da lista inicial.

MergeSort

- O algoritmo de ordenação MergeSort é o primeiro que foge a intuição natural da ordenação.
- Ele é baseado na ideia de dividir para conquistar.
- **MergeSort**
 1. Dividir o vetor ao meio até o ponto em que podemos ordenar de forma trivial (cada vetor tem somente um elemento)
 2. Conquistar a ordenação, recompondo o vetor original a partir das pares ordenadas, visando manter a ordenação entre elas.

Divisão e Conquista: MergeSort

- Considere um vetor V com n posições.
- O algoritmo consiste das seguintes fases:
 1. Dividir V em 2 subcoleções de tamanho $\approx n/2$.
 2. Conquistar: ordenar cada subcoleção chamando MergeSort recursivamente;
 3. Combinar as subcoleções ordenadas formando uma única coleção ordenada.
- Base: Se uma subcoleção tem apenas um elemento, ela já está ordenada.

MergeSort

Algoritmo: Merge Sort

Entrada: Vetor não ordenado $V = \{e_1, e_2, \dots, e_n\}$ com n elementos

Saída: Vetor ordenado V

```
1 Função MergeSort( $V, i, f$ )
2   se  $i < f$  então
3      $meio \leftarrow (i + f)/2$ 
4     MergeSort( $V, i, meio$ )
5     MergeSort( $V, meio + 1, fim$ )
6     Merge( $V, i, meio, fim$ )
7   retorne  $V$ 
```

Chamada Inicial: MergeSort($V, 1, n$)

MergeSort

Algoritmo: Merge

Entrada: Vetor V , Posições ini , $meio$ e fim

Saída: Vetor V

```
1 Função Merge( $V, ini, meio, fim$ )
2    $n \leftarrow fim - ini + 1$ ; /*  $n$  é o tamanho do vetor que vamos juntar */
3   Seja  $A$  um vetor auxiliar com  $n$  posições
4    $p_1 \leftarrow ini$             $p_2 \leftarrow meio + 1$ 
5   para  $i = 1$  até  $n$  faça
6     se  $p_1 \leq meio$  E  $p_2 \leq fim$  então
7       se  $V[p_1] < V[p_2]$  então
8          $A[i] \leftarrow V[p_1]$             $p_1 \leftarrow p_1 + 1$ 
9       senão
10         $A[i] \leftarrow V[p_2]$             $p_2 \leftarrow p_2 + 1$ 
11      senão
12        se  $p_1 \leq meio$  então
13           $A[i] \leftarrow V[p_1]$             $p_1 \leftarrow p_1 + 1$ 
14        senão
15           $A[i] \leftarrow V[p_2]$             $p_2 \leftarrow p_2 + 1$ 
16      para  $i = 1$  até  $n$  faça
17         $V[ini + i] = A[i]$ 
18  retorne  $V$ 
```

MergeSort

```
void MergeSort(int L[], int ini, int fim)
{
    if (ini < fim)
    {
        int meio = (ini+fim) / 2;
        MergeSort(L, ini,  meio);
        MergeSort(L, meio+1,fim);
        Merge(L,ini,meio,fim);
    }
}
```

MergeSort - Merge

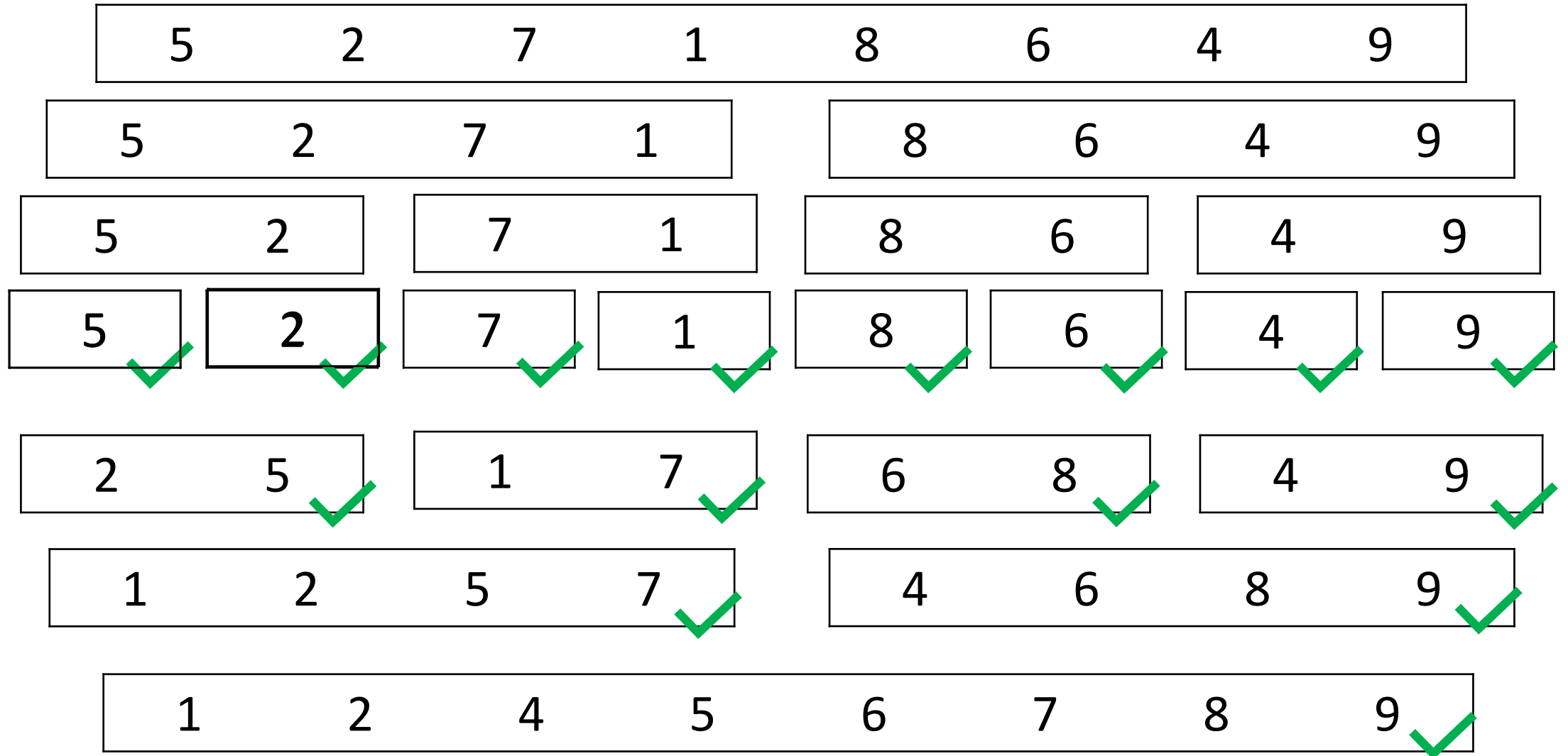
```
void Merge(int L[], int ini, int meio, int fim)
{
    int tam = fim - ini + 1;
    int* auxL = (int*) malloc(tam * sizeof(int));
    int p1 = ini, p2 = meio + 1;
    for (int i = 0; i < tam; i++)
        if (p1 <= meio && p2 <= fim)
            if (L[p1] < L[p2]) auxL[i] = L[p1++];
            else auxL[i] = L[p2++];
        else
            if (p1 <= meio) auxL[i] = L[p1++];
            else auxL[i] = L[p2++];
    for (int i = 0; i < tam; i++) L[ini + i] = auxL[i];
    free(auxL);
}
```

Para lembrar: p1++ é pós-incremento:

- 1) Acessar o valor em p1
- 2) Incrementar o valor de p1

Motivo: Economia de espaço apenas

Merge Sort



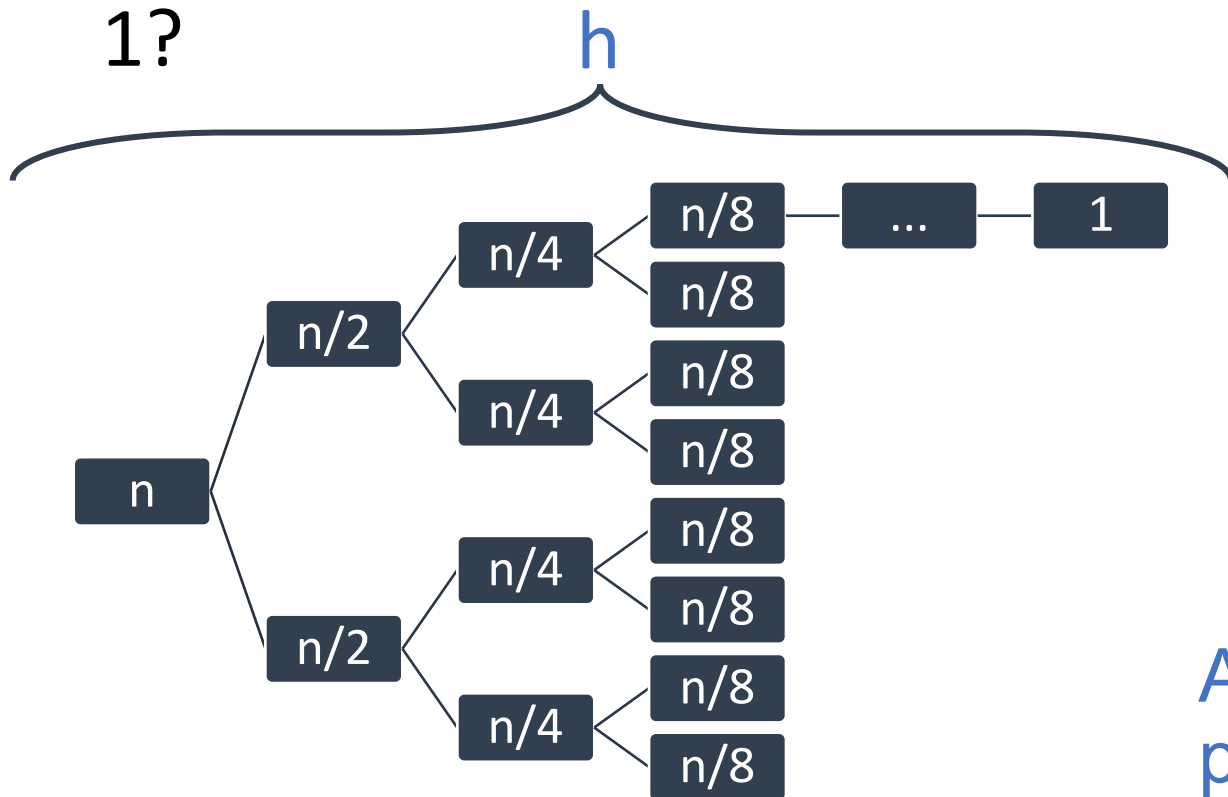
Merge Sort

- Complexidade Merge Sort

$$T(n) = \begin{cases} T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n), & \text{se } n > 1 \\ 1, & \text{caso contrário} \end{cases}$$

Merge Sort

- Começamos uma lista com n elementos
- A cada nova chamada, dividimos a lista em duas
- Quantas chamadas podemos fazer até atingir listas de tamanho 1?



$$\frac{n}{2} \rightarrow \frac{n}{4} \rightarrow \dots \rightarrow 1$$

$$\frac{n}{2^h} = 1$$

$$n = 2^h$$

$$h = \log_2 n$$

Até atingir uma lista de tamanho 1
precisamos efetuar $\log_2 n$ divisões

Merge Sort

- Complexidade Merge Sort

$$T(n) = \begin{cases} T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + T\left(\left\lceil \frac{n}{2} \right\rceil\right) + \Theta(n), & \text{se } n > 1 \\ 1, & \text{caso contrário} \end{cases}$$

- Considerando que o número máximo de divisões possíveis (a altura máxima da árvore de divisões) é $\log_2 n$
- E que para a união de cada nível demanda n comparações
- Complexidade do MergeSort: $\Theta(n \log_2 n)$

Merge Sort

- Complexidade: $O(n \log_2 n)$
- Estável: Sim – mas depende da implementação (a implementação apresentada é estável). Por que?
- *In-place*: Não – ele requer uma lista auxiliar no momento em que as duas metades são unidas.
- Online: Não.
- Curiosidade: Existe uma versão in-place do MergeSort, mas ela apresenta complexidade $O(n^2)$

Quick Sort

- Assim como o MergeSort, segue a ideia da divisão e conquista.
- Para uma lista com n elementos, selecionamos de forma aleatória um elemento chamado pivô.
- Em seguida, movemos todos os elementos seguindo uma regra:
 - Os elementos menores do que o pivô ficam a sua esquerda
 - Os elementos maiores do que o pivô ficam a sua direita
- Depois das movimentações, o pivô estará no seu lugar definitivo
- Para cada parte da lista (esquerda e direita) aplicamos o próprio QuickSort
- Ele só termina quando cada lista tem somente um elemento (e portanto está ordenada)

Quick Sort

- Duas fases:
 - Dividir
 - Escolher um pivô
 - Mover todos os menores do que o pivô para um lado
 - Mover todos os maiores do que o pivô para o outro lado
 - Conquistar
 - Vamos conquistar cada metade usando próprio quicksort

QuickSort

- Escolha do Pivô:
 - A escolha desse elemento que divide a lista é crítica para o funcionamento do algoritmo.
 - Uma escolha ruim, vai reduzir muito a eficiência do algoritmo
 - Como escolher?
- Três alternativas:
 - Escolher o primeiro elemento da lista
 - Escolher o último elemento da lista
 - Escolher o elemento no meio da lista

Como mover os demais elementos?

1. Vamos escolher uma posição fixa para o pivô não atrapalhar essa movimentação: Uma vez escolhido, vamos colocar o pivô no final da lista
2. Em seguida:
 - Buscar da esquerda para a direita o primeiro elemento maior do que o pivô
 - Buscar da direita para a esquerda o primeiro elemento menor do que o pivô
 - Se o da esquerda não ultrapassou o elemento da direita, vamos trocar e continuar a busca. Caso contrário, terminamos o algoritmo.
3. Vamos mover o pivô para o seu local definitivo

QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```

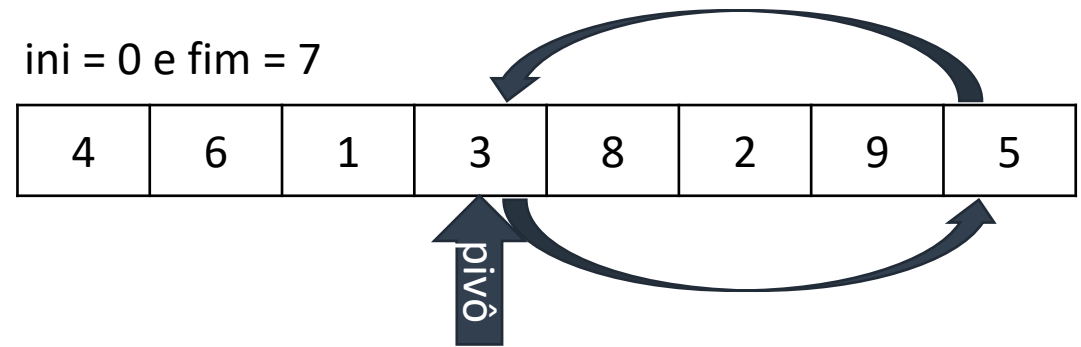
ini = 0 e fim = 7

4	6	1	3	8	2	9	5
---	---	---	---	---	---	---	---

↑
pivô

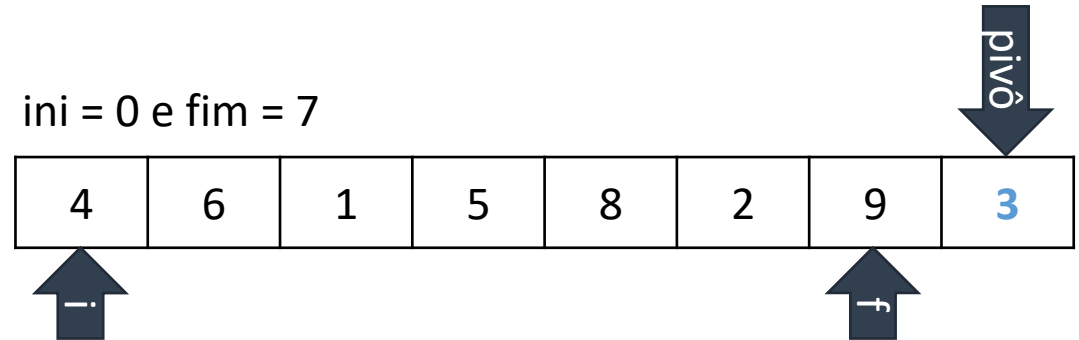
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



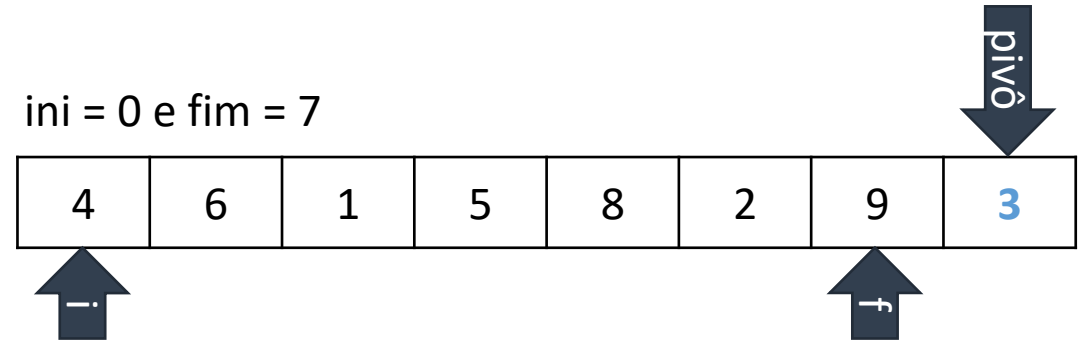
Quick Sort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



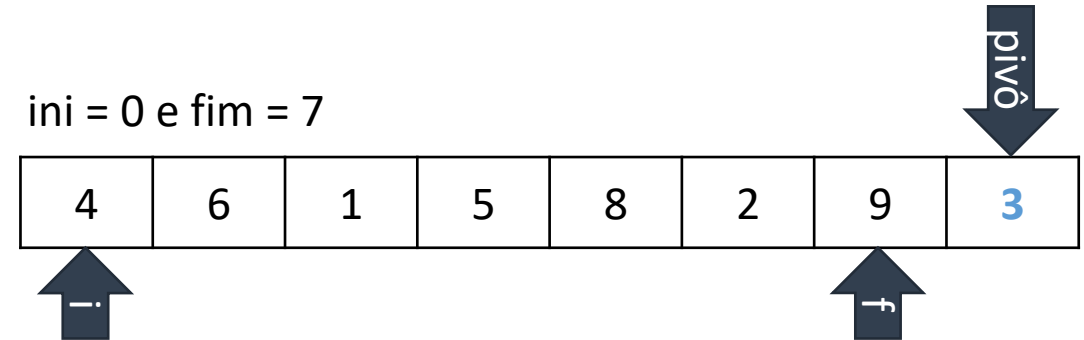
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



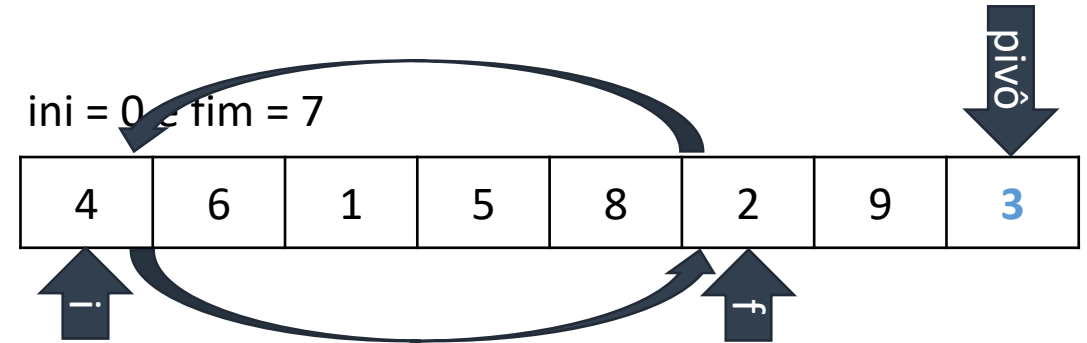
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



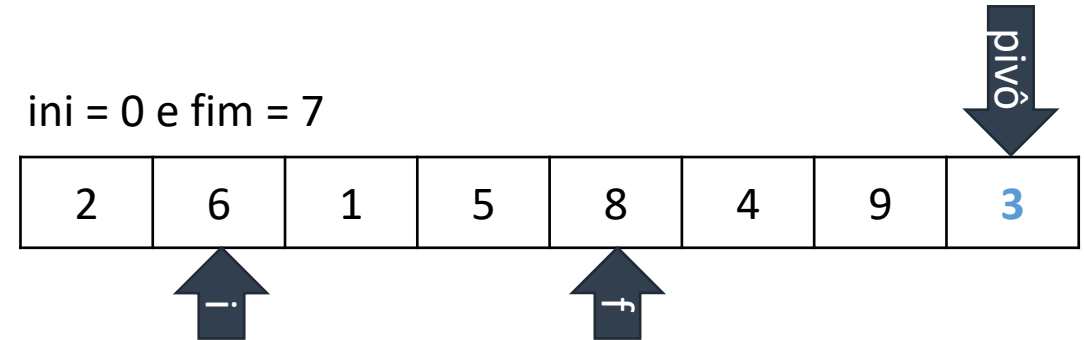
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



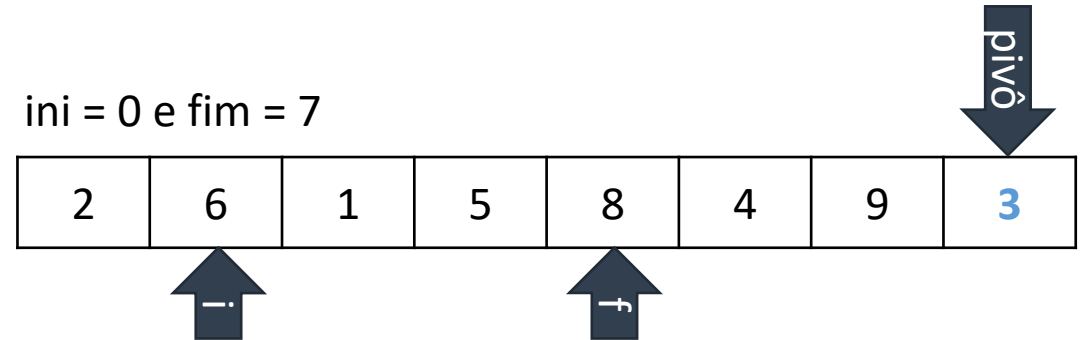
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L,ini,i-1);
QuickSort(L,i+1,fim);
```



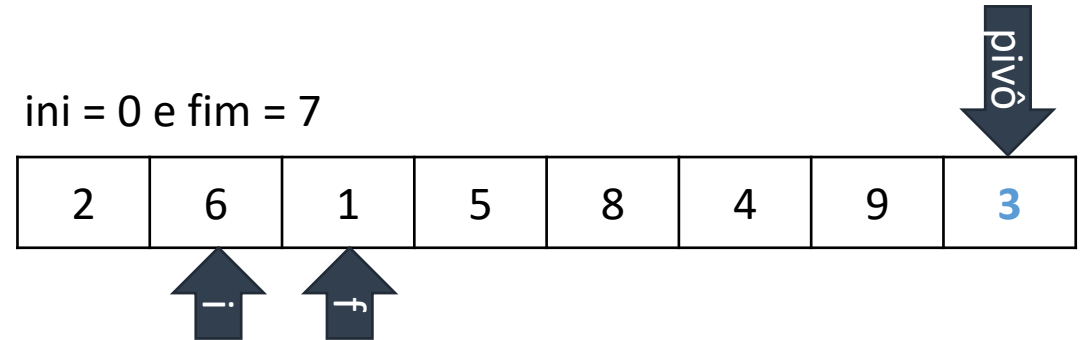
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



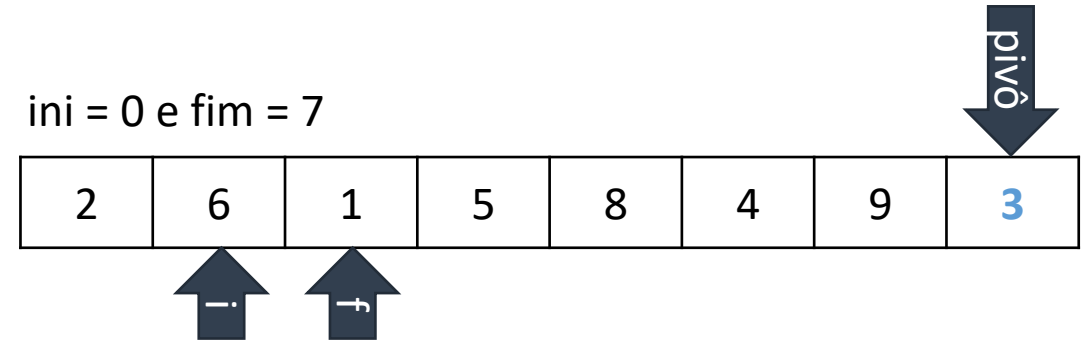
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



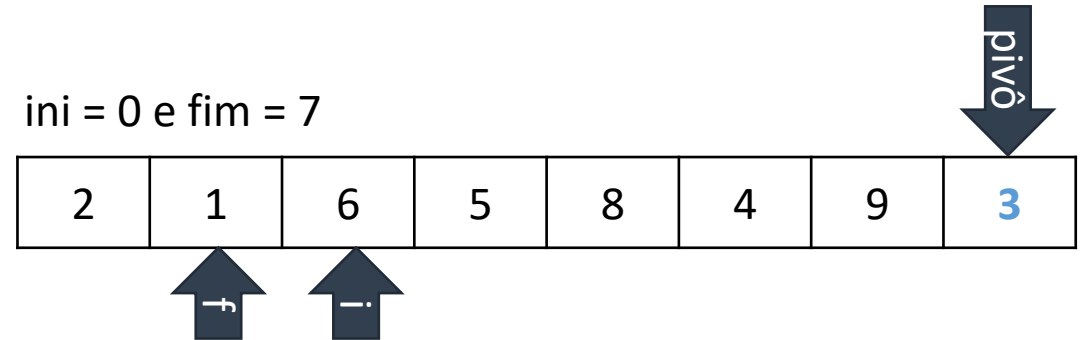
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



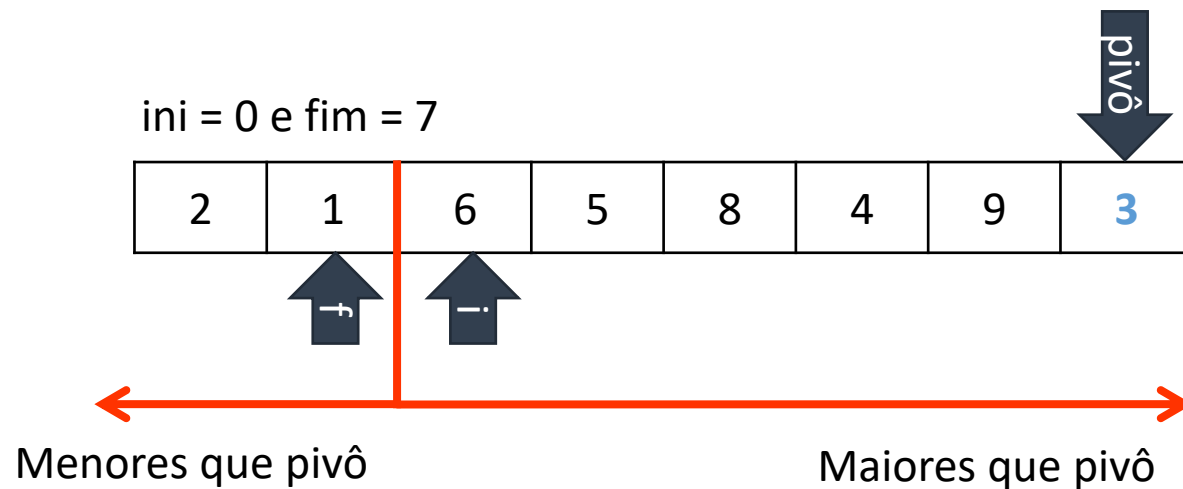
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



QuickSort

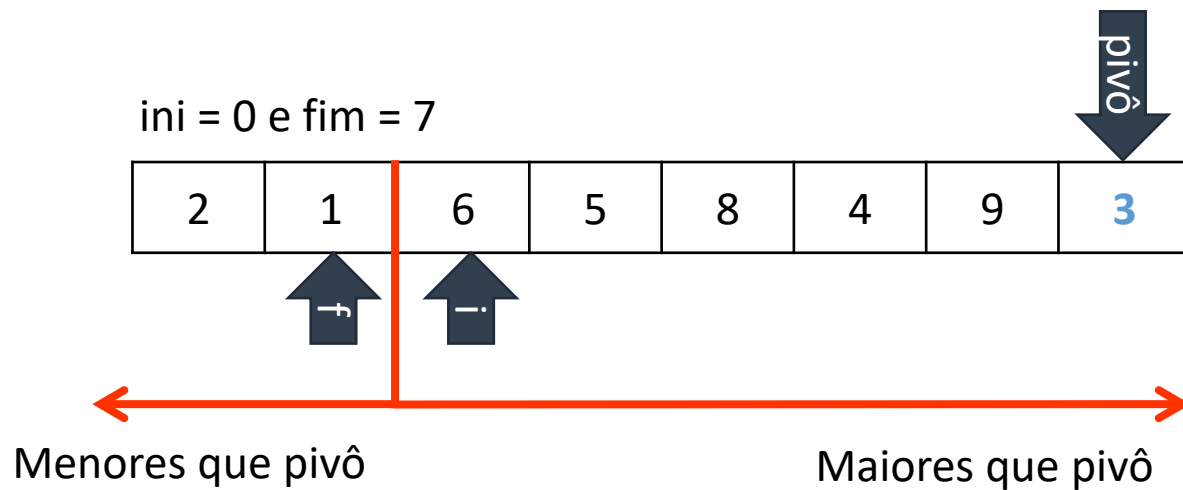
```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



Falta colocar pivô em seu lugar

QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



Falta colocar pivô em seu lugar

QuickSort

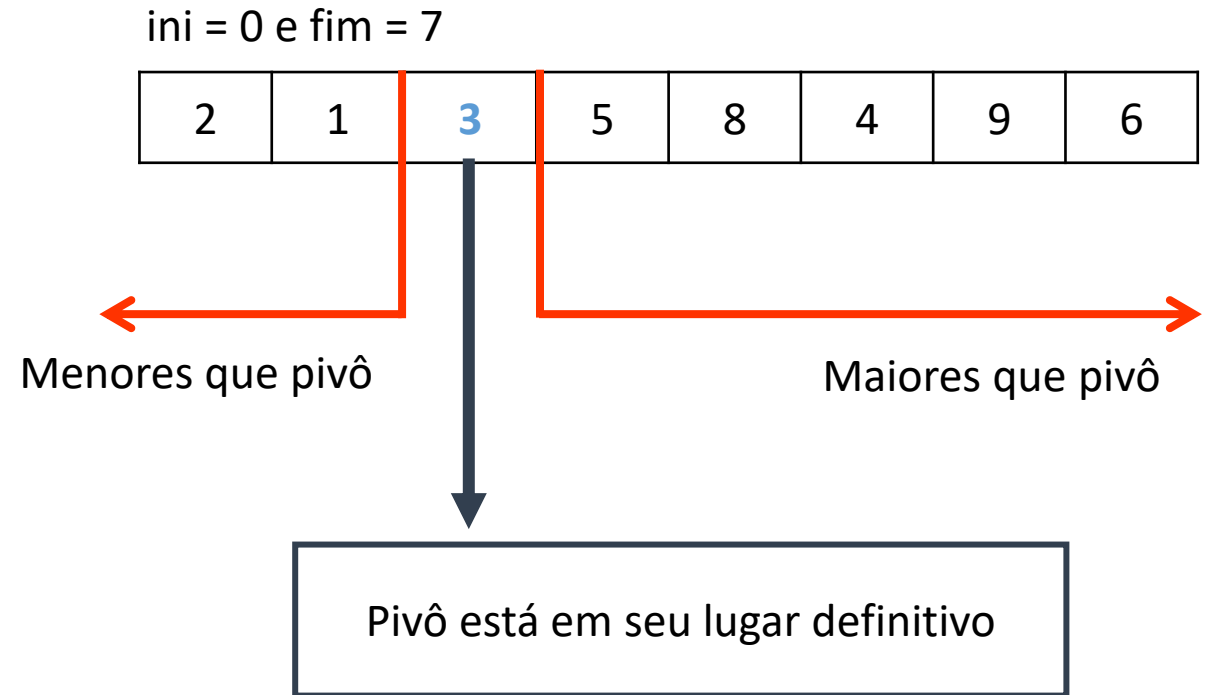
```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```

ini = 0 e fim = 7

2	1	3	5	8	4	9	6
---	---	---	---	---	---	---	---

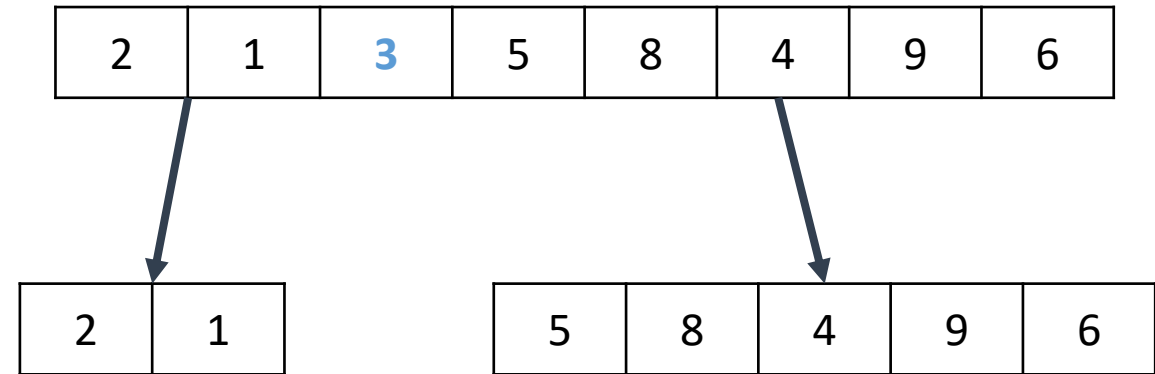
QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



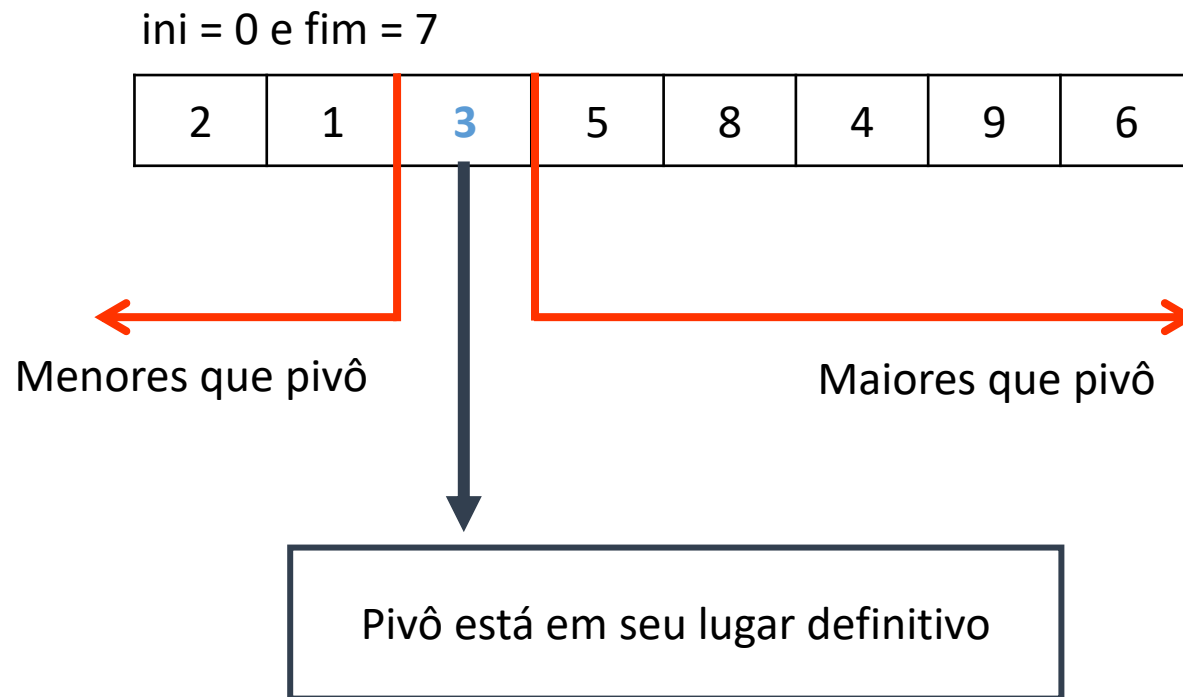
E agora?

- Como ordenar as duas metades?
- Usando QuickSort!
- E quando a ordenação termina?
 - Quando cada lista possuir somente um único elemento.
 - Listas com somente um elemento estão ordenadas por definição.



QuickSort

```
int pivo = L[(fim+ini) / 2];
Trocar(&L[(fim+ini) / 2], &L[fim]);
int i = ini;
int f = fim - 1;
while (f >= i)
{
    while (L[i] < pivo) i++;
    while (L[f] > pivo) f--;
    if (f >= i)
    {
        Trocar(&L[i], &L[f]);
        i++; f--;
    }
}
Trocar(&L[fim], &L[i]);
QuickSort(L, ini, i-1);
QuickSort(L, i+1, fim);
```



```
void QuickSort(int L[], int ini, int fim)
```

```
{
```

```
    if (ini < fim)
```

```
    {
```

```
        int pivo = L[(fim+ini) / 2];
```

```
        Trocar(&L[(fim+ini) / 2], &L[fim]);
```

```
        int i = ini;
```

```
        int f = fim - 1;
```

```
        while (f >= i)
```

```
        {
```

```
            while (L[i] < pivo) i++;
```

```
            while (L[f] > pivo) f--;
```

```
            if (f >= i)
```

```
            {
```

```
                Trocar(&L[i], &L[f]);
```

```
                i++; f--;
```

```
            }
```

```
        }
```

```
        Trocar(&L[fim], &L[i]);
```

```
        QuickSort(L, ini, i-1);
```

```
        QuickSort(L, i+1, fim);
```

```
    }
```

```
}
```

Se a lista possui só um elemento, então $ini = fim$. É o fim do algoritmo

Em seguida, selecionamos o pivô e o movemos para o final da lista

Agora vamos arrumar a lista:

Os menores que pivô ficam a sua esquerda

Os maiores que pivô ficam a sua direita

Vamos mover pivô para seu local definitivo

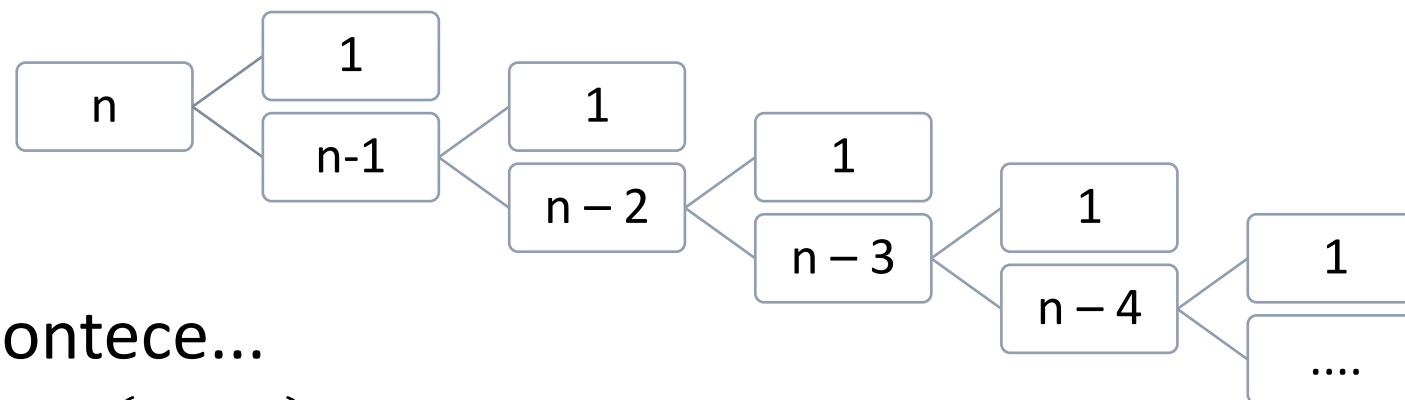
Por último, chamamos QuickSort para ordenar cada metade da lista (os menores que pivô e os maiores que pivô)

E a complexidade?

- Melhor caso:
 - Todo pivô escolhido divide a lista em duas metades iguais.
 - $T(n) = T(n/2) + T(n/2) + n$ ou seja $O(n \log_2 n)$
- Mas é igual ao MergeSort? Sim... Então os dois tem a mesma "velocidade"?
- Não:
 - O QuickSort não precisa fazer cópias das listas
 - Assim ele economiza no tempo de copiar as listas e no uso de memória
- Então o QuickSort é melhor do que o MergeSort?
 - Depende... Vamos analisar o pior caso

E a complexidade?

- Pior caso:
 - Todas as escolhas de pivô são muito ruins e não dividem a lista.



- Quando isso acontece...
 - $T(n) = T(1) + T(n - 1) + n$
 - Quando resolvemos essa expressão, temos que:
 - No pior caso, a complexidade do QuickSort é $O(n^2)$
 - Isso é igual ao BubbleSort
- Mas por que ainda usamos Quick Sort??
 - Porque esse caso é pouco comum

E a complexidade:

- Melhor caso: $O(n \log_2 n)$
- Pior caso: $O(n^2)$
- Mas o caso médio é bem próximo ao melhor caso...
- O pior caso raramente ocorre...
- Dessa forma:
 - O QuickSort é um dos métodos mais usados para ordenação.



Quick Sort

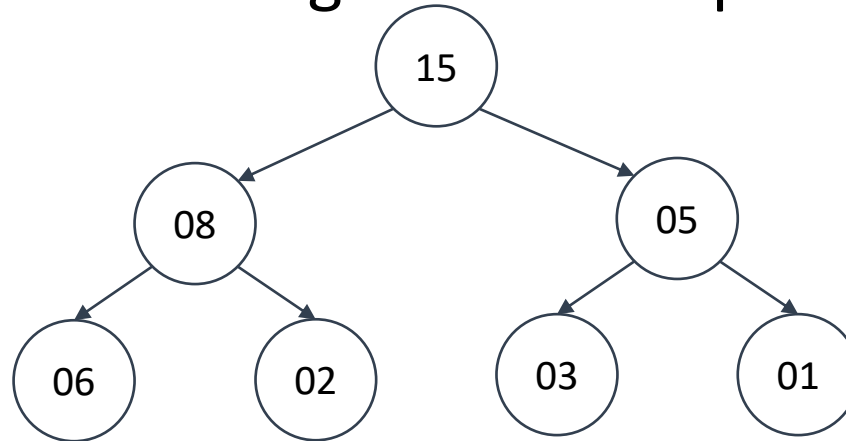
- Complexidade de Pior Caso: $O(n^2)$
- Complexidade de Caso Médio: $O(n \log_2 n)$
- Estável: Não – o elemento escolhido como pivô a cada iteração pode ser movimentado em relação aos outros elementos de mesma chave.
- *In-place*: Sim.
- Online: Não.

Heap Sort

- Antes de falar do HeapSort, vamos falar de uma estrutura de dados chamada de Heap.
- Uma estrutura (seja ela uma árvore ou uma lista) é um heap se cada elemento satisfaz a **Propriedade dos Heaps**.
- **Propriedade dos Heaps**: A chave de um elemento deve ser maior do que a chave de seus filhos.
- Vale observar que essa propriedade é hereditária:
 - Se a chave de um elemento é maior do que a dos filhos
 - E se a chave dos filhos é maior do que a de seus filhos
 - Então: a chave do elemento é maior do que a chave de todos os seus descendentes.
 - Note que essa propriedade **não** vale entre irmãos ou primos: não existe ordenação entre eles.

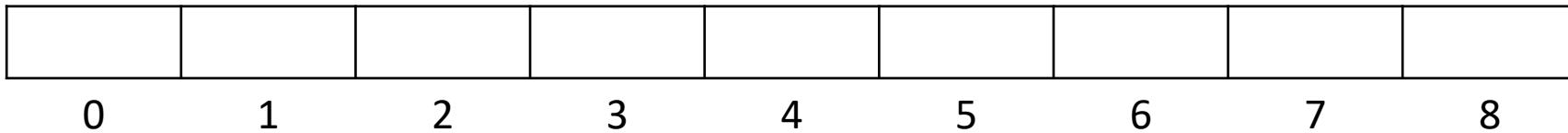
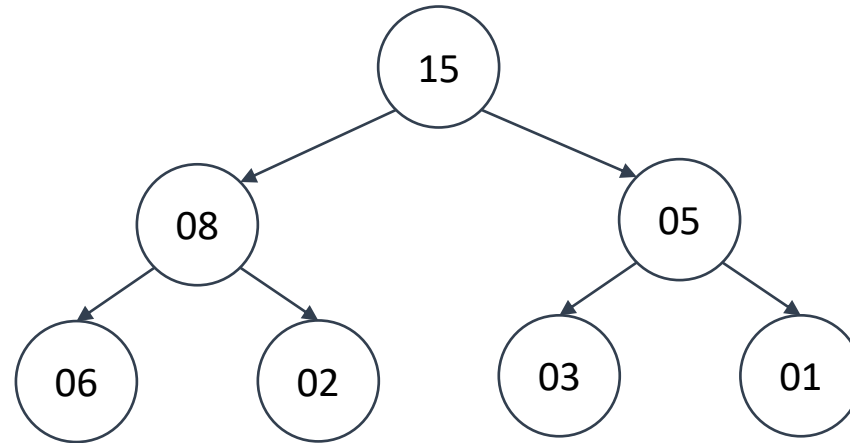
Heap

- Os Heaps podem ser implementados de duas formas:
 - Como árvores – Não chegamos nesse ponto do curso ainda

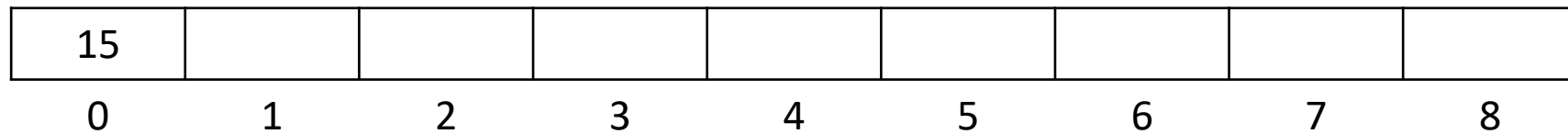
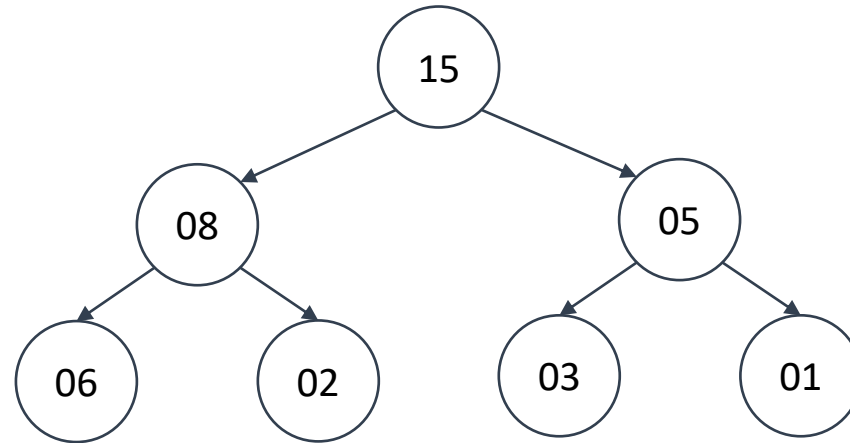


- Como listas – Já sabemos listas (vetores)
 - Mas como armazenar essa estrutura de pais e filhos em um vetor?
 - Simples: Vamos usar o índice dos elementos.

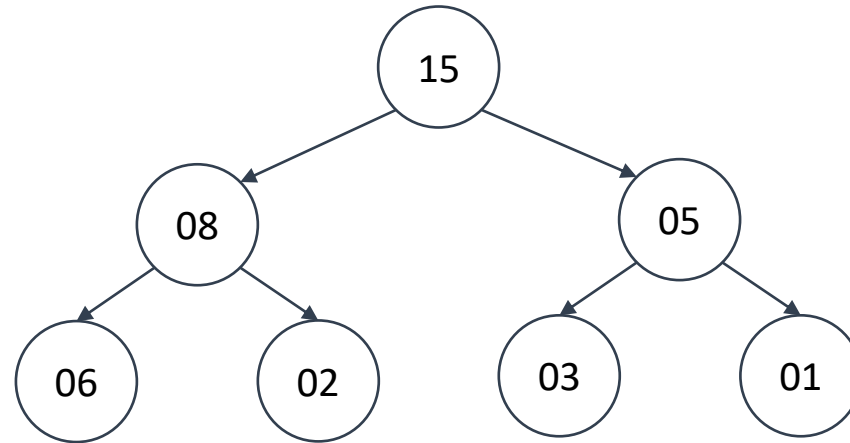
Heap



Heap

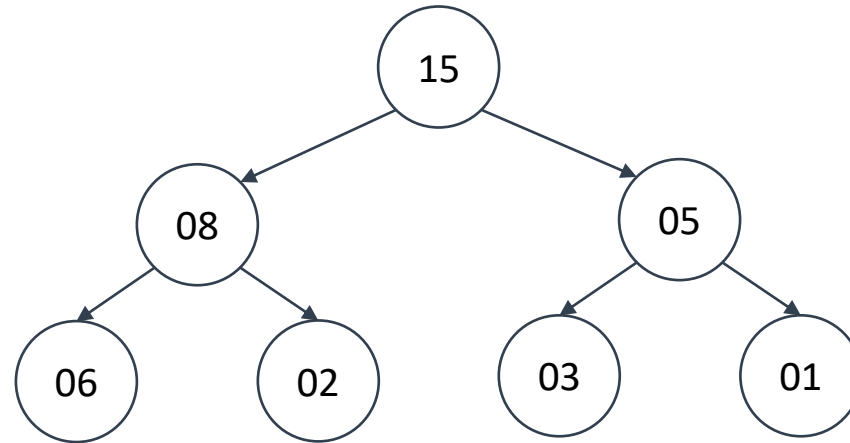


Heap



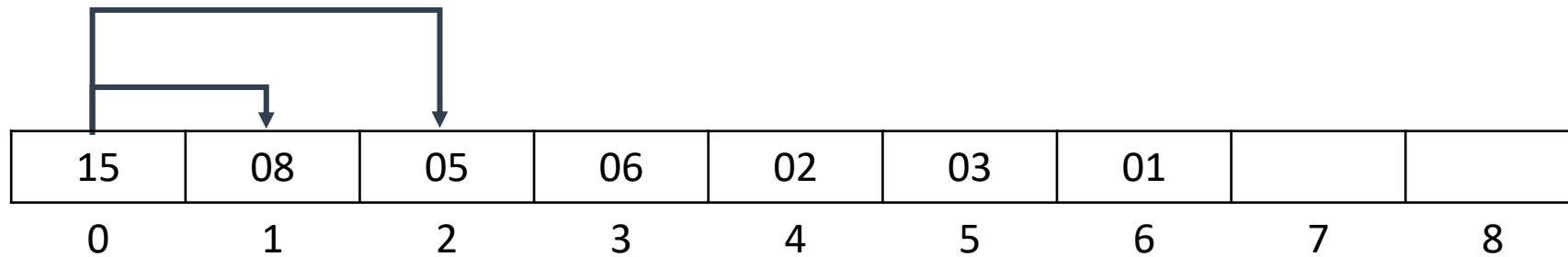
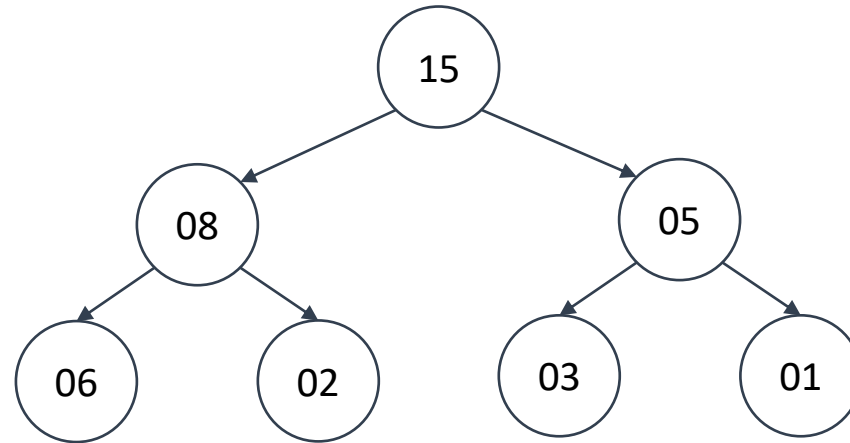
15	08	05						
0	1	2	3	4	5	6	7	8

Heap

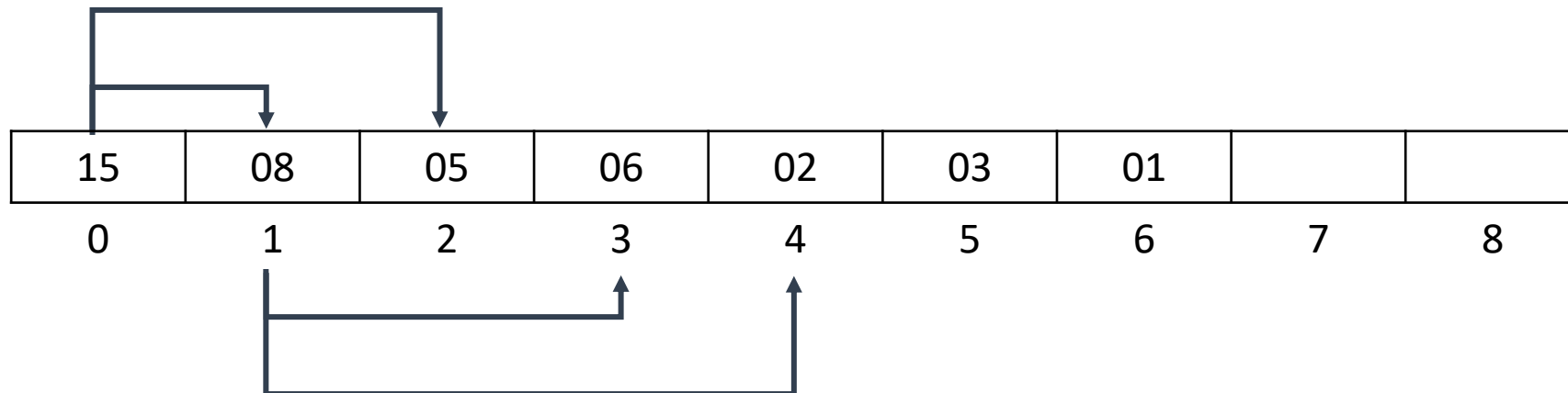
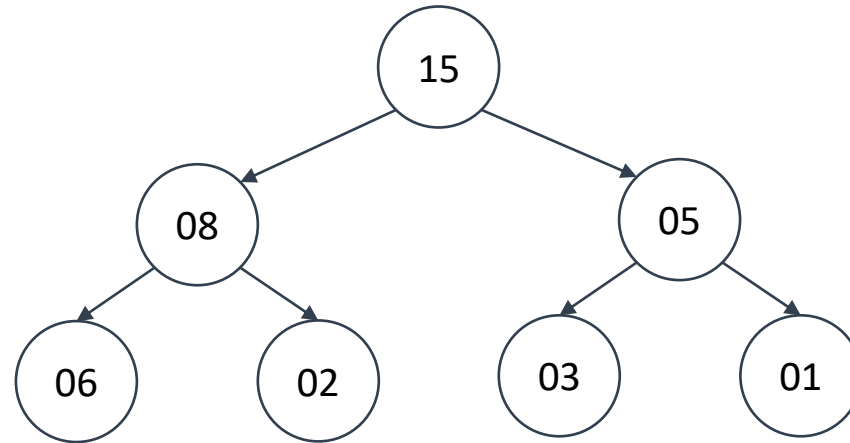


15	08	05	06	02	03	01		
0	1	2	3	4	5	6	7	8

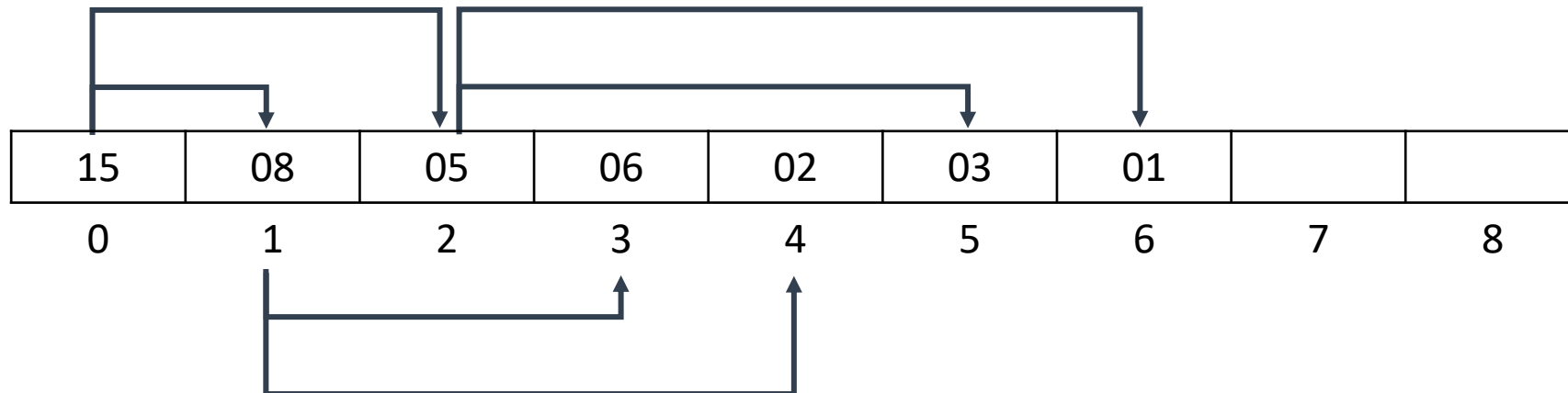
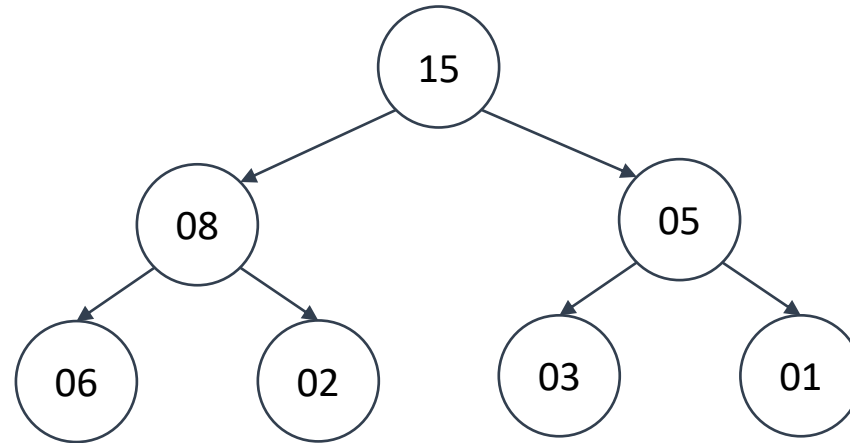
Heap



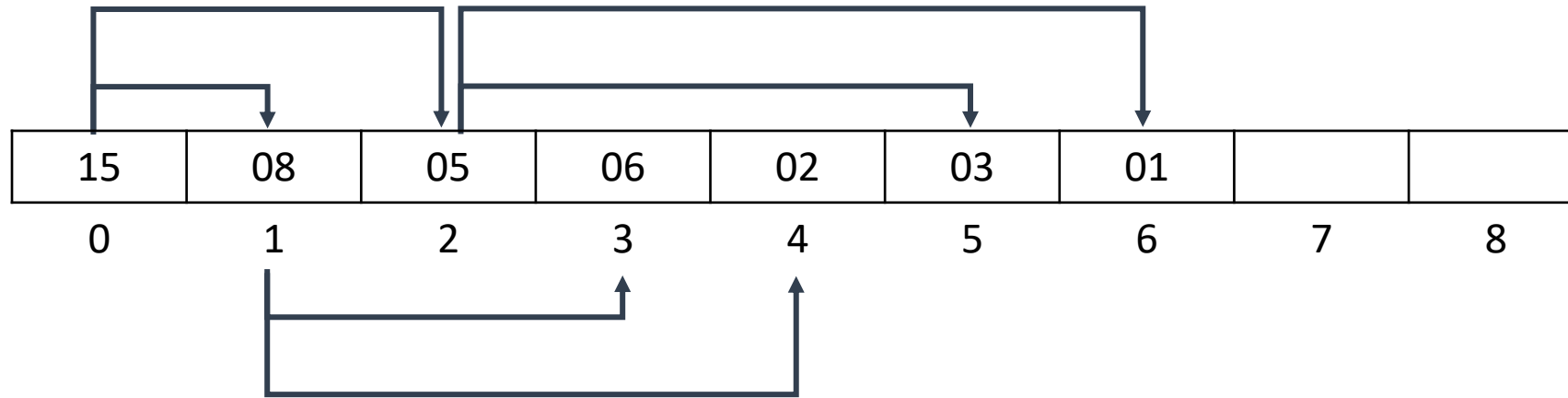
Heap



Heap



Heap



- Considerando o esquema acima, é possível construir uma regra que indique em quais posições estão os filhos de um elemento situado na posição i do vetor?
 - Filho 1: $2i + 1$
 - Filho 2: $2i + 2$
- Sabendo a posição j do filho é possível encontrar seu pai? Como?
 - Pai: $j - 1/2$

Heap: Verificando se vetor é um Heap

```
int VerificaHeap(int L[], int tam)
{
    for (int i = 0; i < tam; i++)
    {
        if (2*i + 1 < tam && L[i] < L[2*i+1])
            return 0;
        if (2*i + 2 < tam && L[i] < L[2*i+2])
            return 0;
    }
    return 1;
}
```

Heap: Como transformar um vetor em Heap

```
void ConstruirHeap(int L[], int tam, int i)
{
    int posMaior = i;

    if (2 * i + 1 < tam && L[2 * i + 1] > L[posMaior]) posMaior = 2 * i + 1;
    if (2 * i + 2 < tam && L[2 * i + 2] > L[posMaior]) posMaior = 2 * i + 2;

    if (posMaior != i)
    {
        Trocar(&L[i], &L[posMaior]);
        ConstruirHeap(L, tam, posMaior);
    }
}

for (int i = (tam/2) - 1; i >= 0; i--) ConstruirHeap(V, tam, i);
```

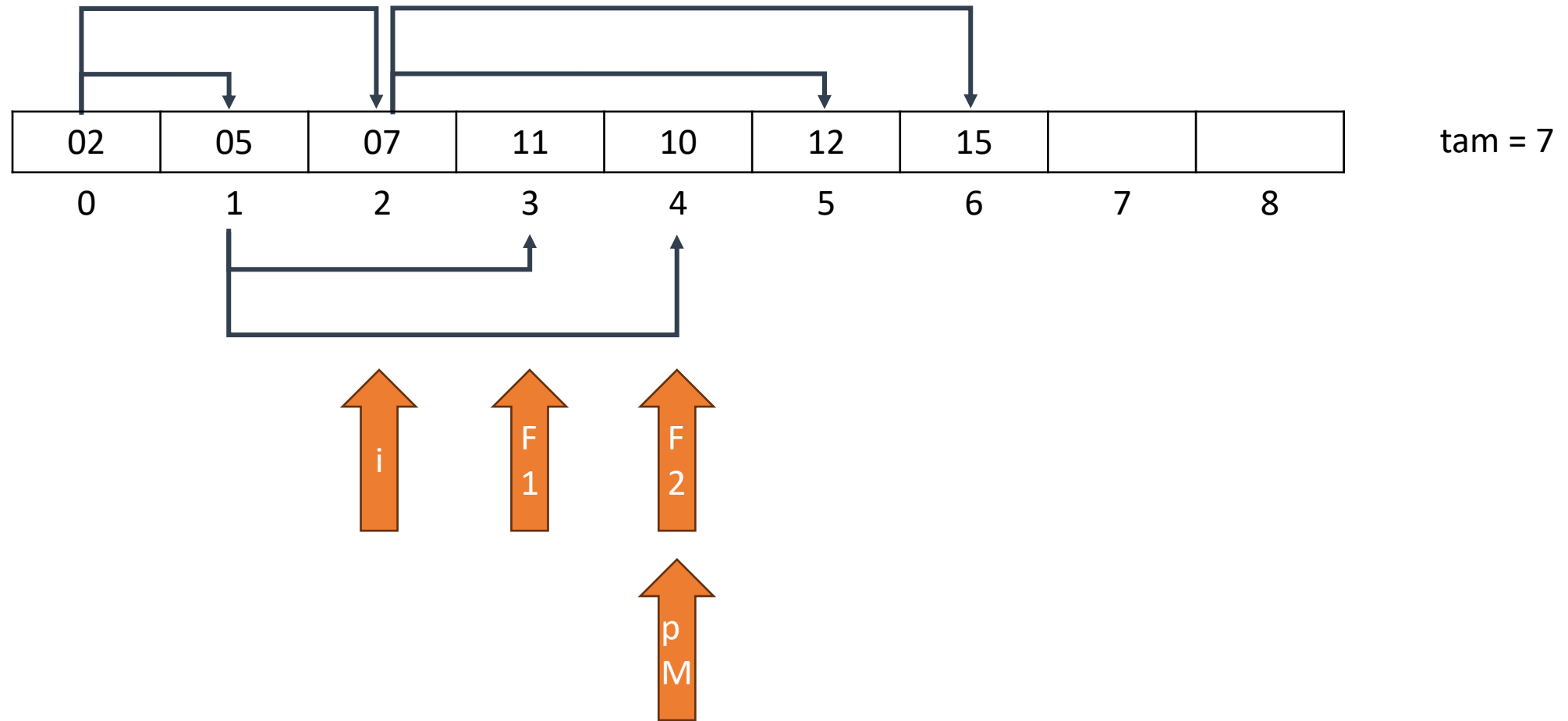
Heap: Como transformar um vetor em Heap

- O procedimento `ConstruirHeap` identifica se dentro os filhos de um elemento existe um que apresenta chave maior do que o elemento. Isso violaria a propriedade Heap.
- Se isso for verdade, ele troca as chaves: a maior passa a ocupar a posição do pai.
- Entretanto, isso pode violar a propriedade Heap do filho alterado.
- Para assegurar que esse erro permaneça, chamamos a função `ConstruirHeap` para o filho alterado.

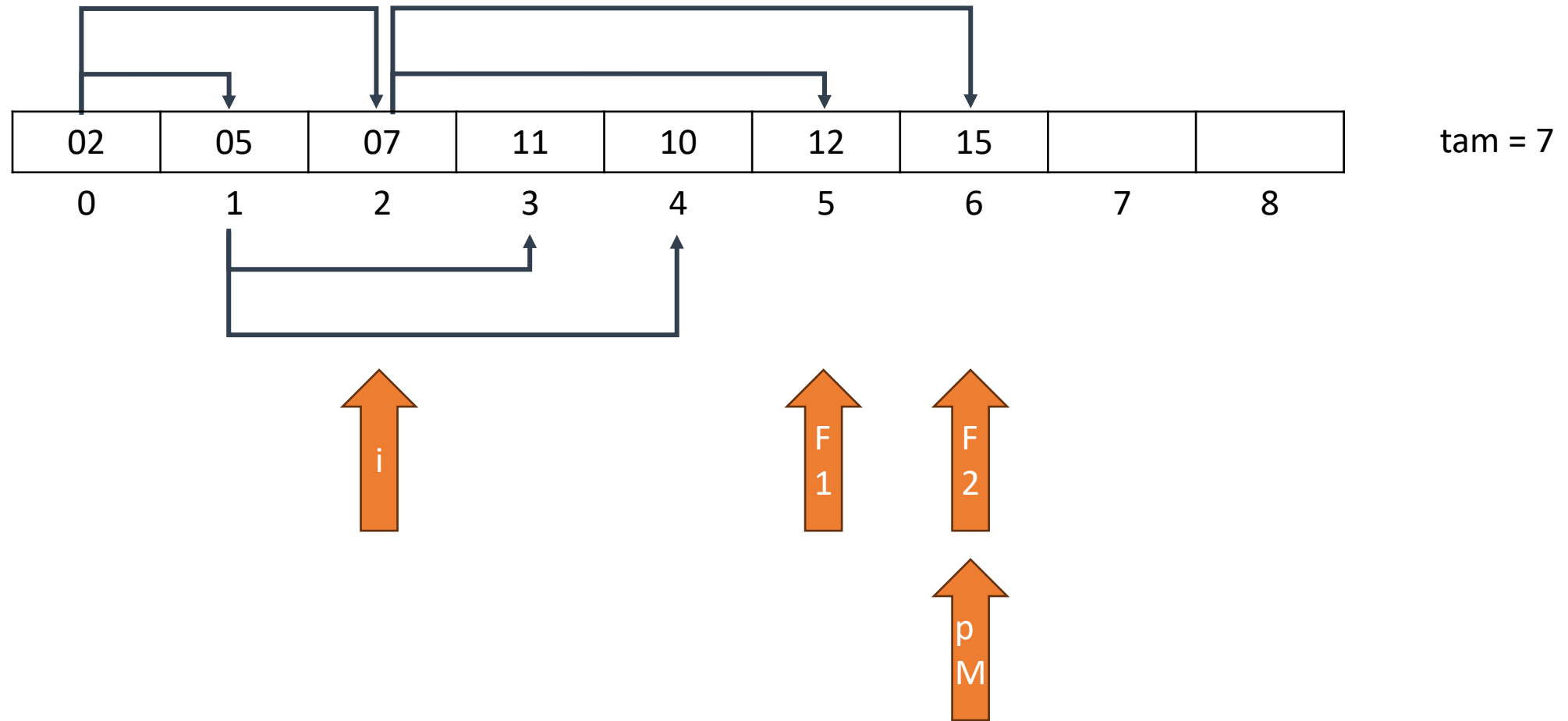
Heap: Como transformar um vetor em Heap

- Vale observar que a chamada inicial a função `ConstruirHeap` começa pelo final do Heap.
- Como não faz sentido chamar o procedimento para elementos sem filhos (eles não podem violar a condição heap), começamos a chamada pelo último elemento da lista que possui pelo menos um filho.
- Esse elemento ocupa a posição $(\text{tam}/2) - 1$

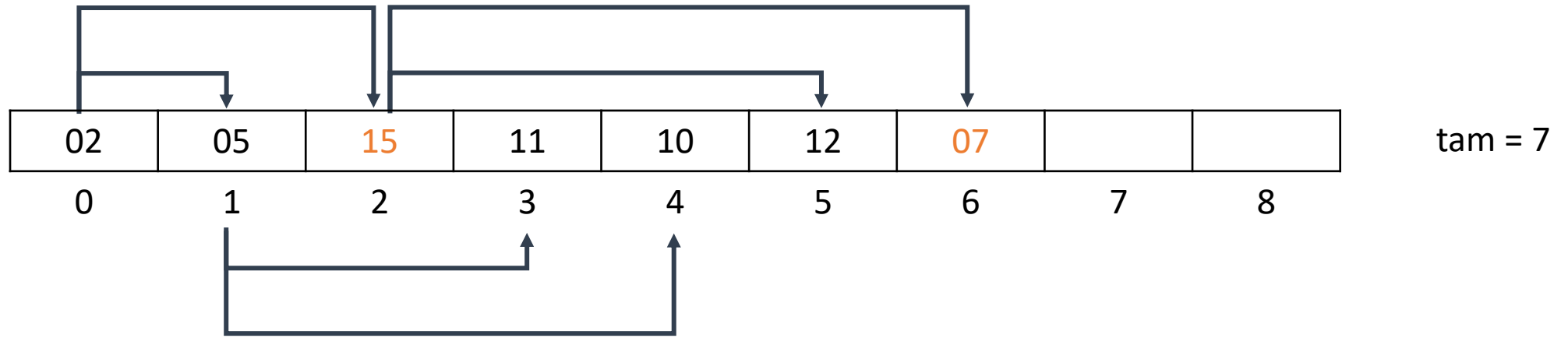
Heap: Como transformar um vetor em Heap



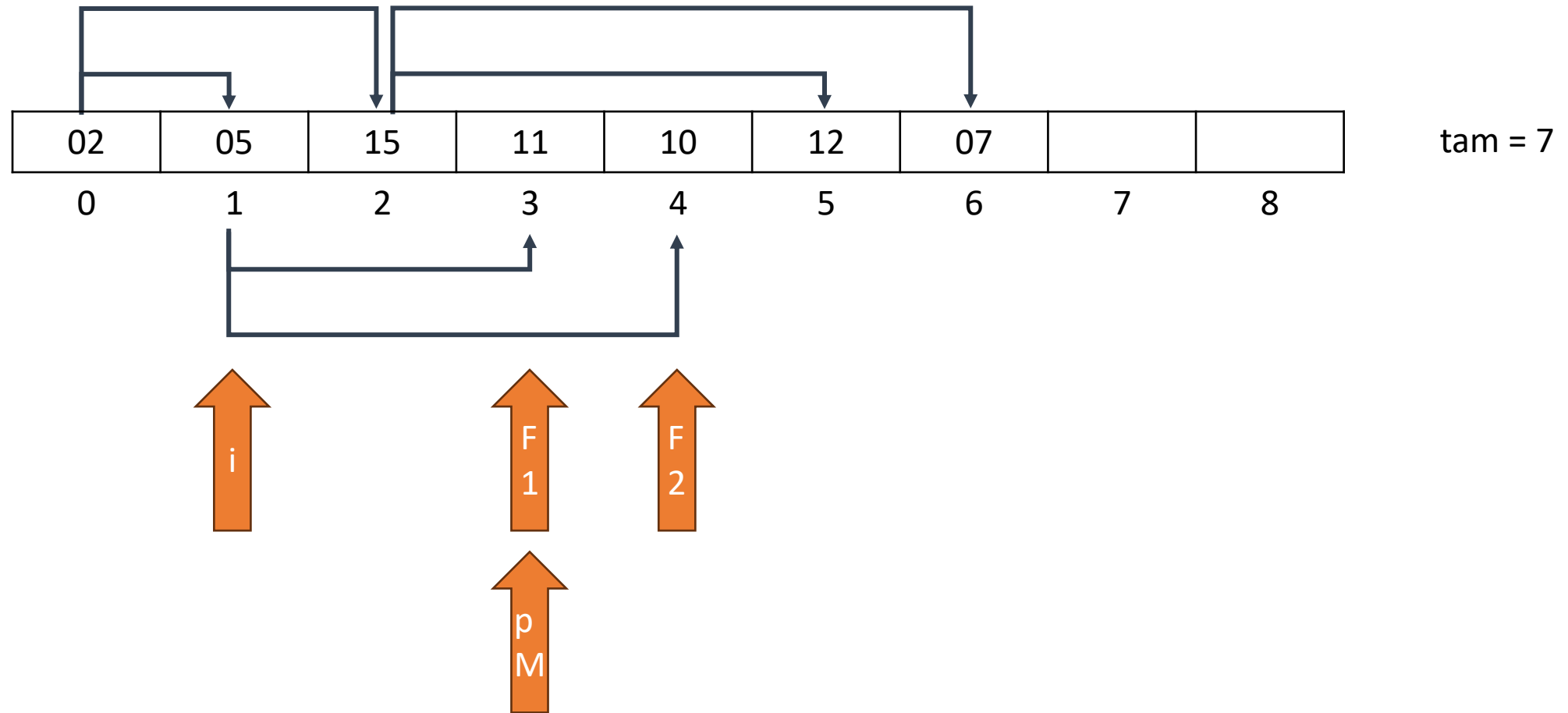
Heap: Como transformar um vetor em Heap



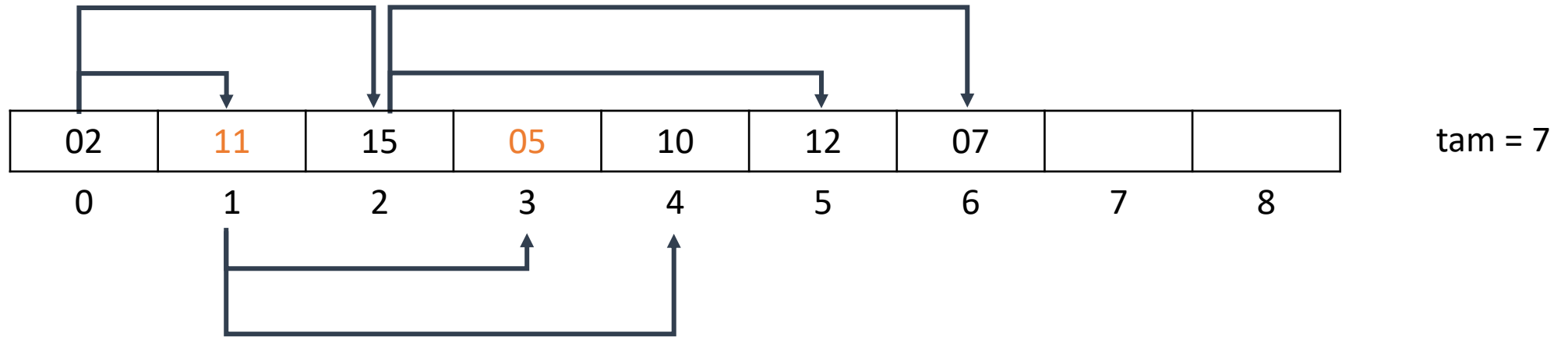
Heap: Como transformar um vetor em Heap



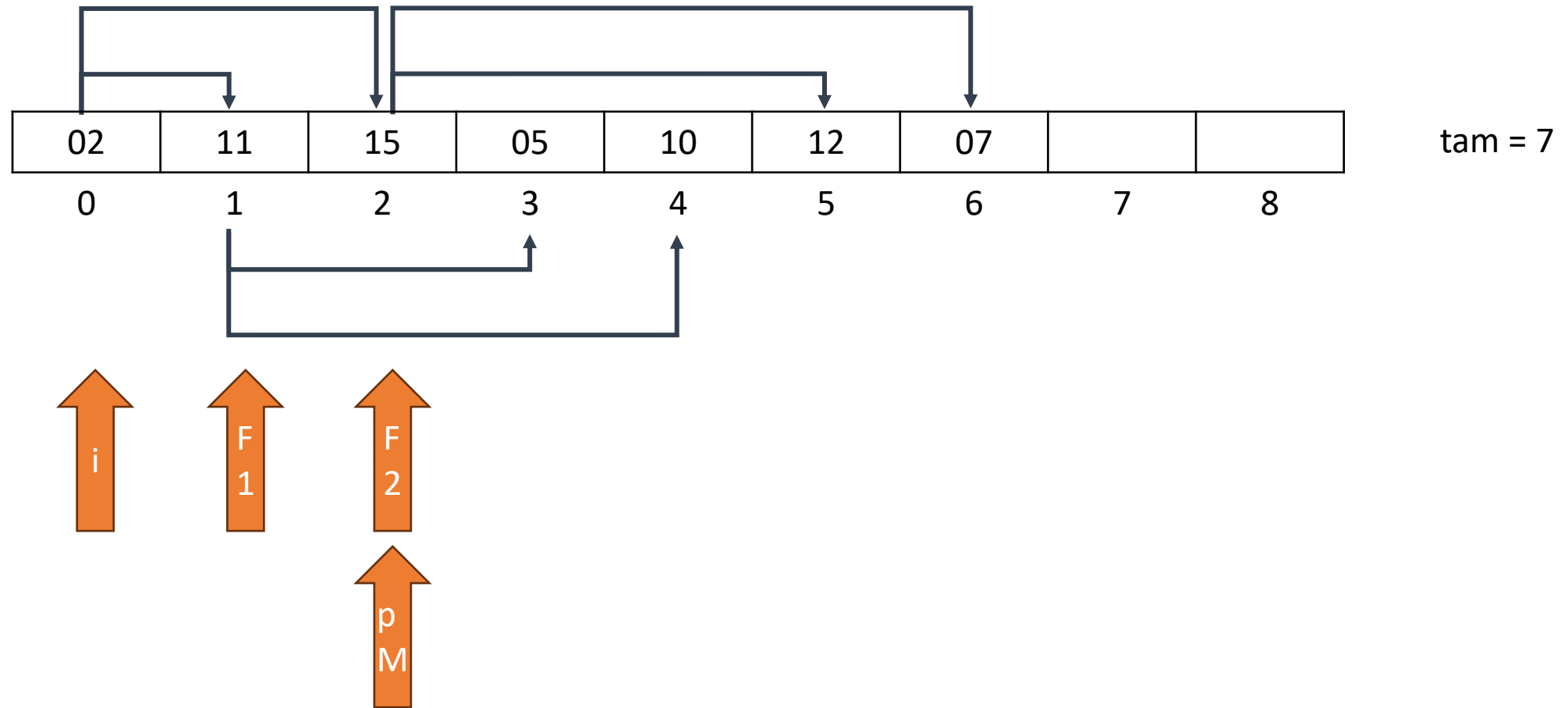
Heap: Como transformar um vetor em Heap



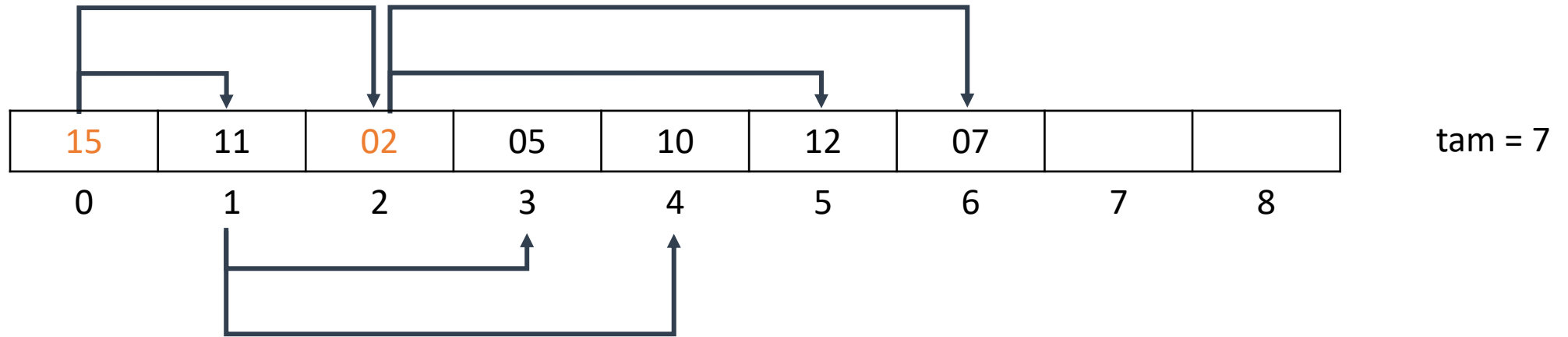
Heap: Como transformar um vetor em Heap



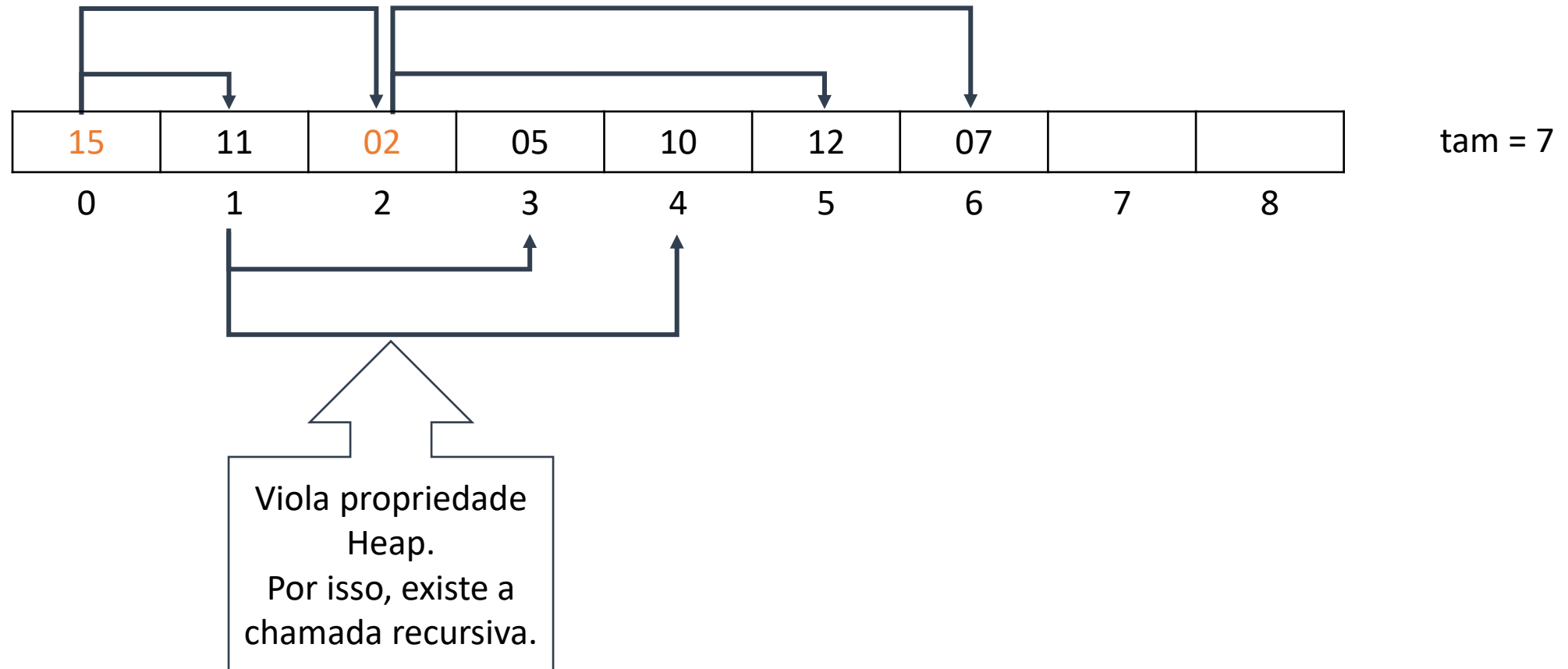
Heap: Como transformar um vetor em Heap



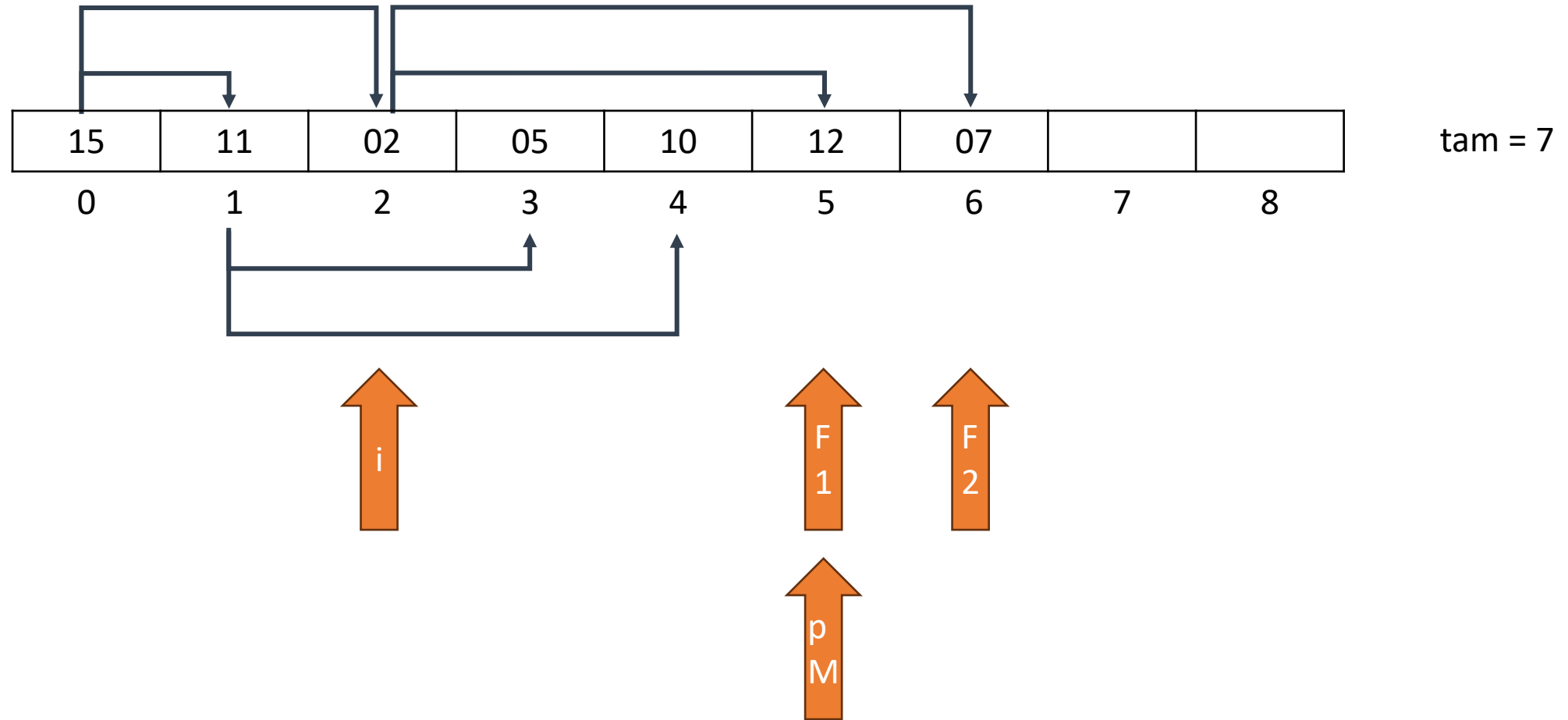
Heap: Como transformar um vetor em Heap



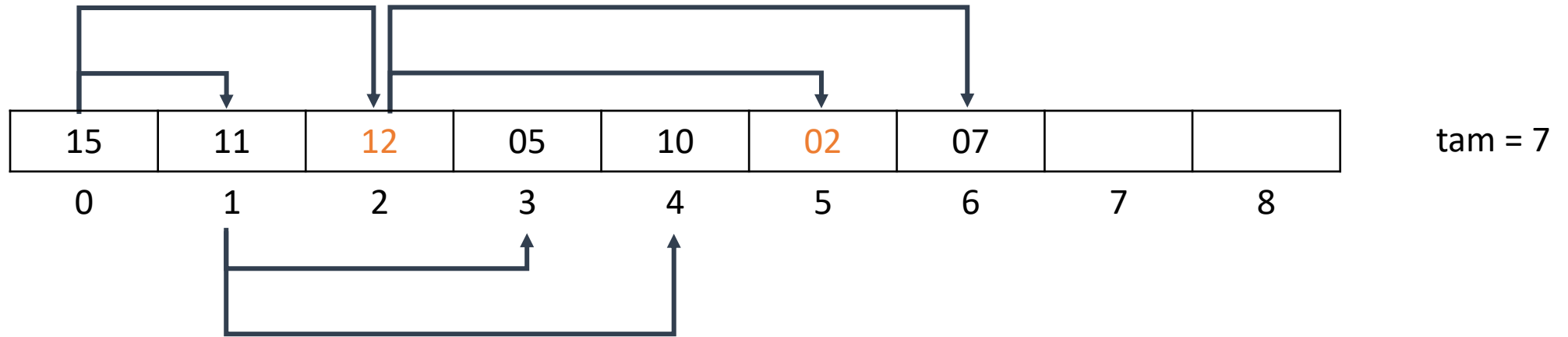
Heap: Como transformar um vetor em Heap



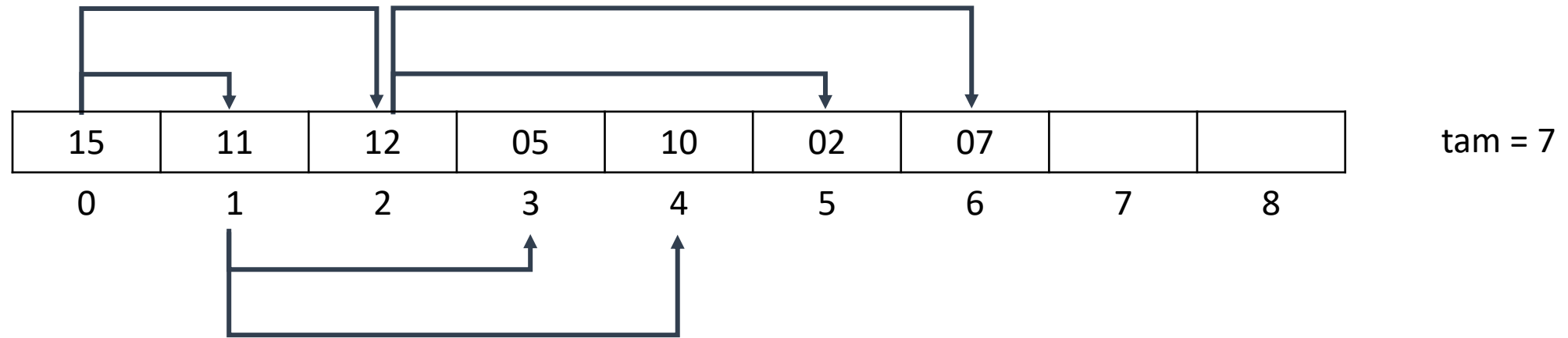
Heap: Como transformar um vetor em Heap



Heap: Como transformar um vetor em Heap



Heap: Como transformar um vetor em Heap



- Todos os elementos do vetor resultante atendem a propriedade Heap.
- O maior elemento ocupa a primeira posição do vetor.

Heap: Como transformar um vetor em Heap

- Qual é a complexidade da inicialização do Heap?
- 1) São realizadas $n/2$ chamadas ao procedimento `ConstruirHeap`.
- 2) Cada chamada ao `ConstruirHeap` tem complexidade $O(\log n)$
 - Cada chamada gera uma chamada recursiva para um dos filhos do elemento.
 - Como a altura dessa “árvore” é de no máximo $\log_2 n$ e cada chamada faz pelo menos uma comparação, então $O(\log n)$.
- No total: Temos custo de $O(n \log n)$

Transformar em Heap \neq Ordenar

- Vale observar que transformar um vetor em heap não ordena o vetor.
- Atender a propriedade do Heap só garante que o maior elemento é o primeiro elemento da lista.
- Sabendo disso, como usar esses conceitos para ordenar?

Heap: Como usar na ordenação?

- Após construir o Heap, o maior elemento é o primeiro da lista.
- Podemos mover esse elemento para o final do vetor. Essa é sua posição definitiva.
- Precisamos então substituir o primeiro da lista por um outro elemento que atenda a propriedade heap.
- Para isso, vamos substituir o elemento por um outro qualquer.
- E usar a função `ConstruirHeap` para corrigir a propriedade Heap.
- Ela já faz o trabalho de assegurar que o maior elemento ocupe a primeira posição.
- Vamos repetir essa ideia até que todos os elementos do vetor ocupem suas posições definitivas.

HeapSort


```
void HeapSort(int L[], int tam)
{
    for (int i = (tam/2) - 1; i >= 0; i--)
        ConstruirHeap(L, tam, i);

    for (int i = tam - 1; i >= 0; i--)
    {
        Trocar(&L[0], &L[i]);
        ConstruirHeap(L, i, 0);
    }
}
```

HeapSort

```
void HeapSort(int L[], int tam)
{
    for (int i = (tam/2) - 1; i >= 0; i--)
        ConstruirHeap(L, tam, i);

    for (int i = tam - 1; i >= 0; i--)
    {
        Trocar(&L[0], &L[i]);
        ConstruirHeap(L, i, 0);
    }
}
```



Importante: O tamanho do Heap precisa ser reduzido ao longo do processo. As últimas posições do vetor não devem mais ser incluídas no Heap, pois já correspondem a posições definitivas da ordenação.

HeapSort

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

Lista inicial

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

Heap Construído

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

9 movido para posição definitiva
5 viola propriedade Heap

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

Heap ajustado

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

8 movido para posição definitiva
0 viola propriedade Heap

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

Heap ajustado

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

7 movido para posição definitiva
2 viola propriedade Heap

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

Heap ajustado

HeapSort

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

Heap ajustado

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

6 movido para posição definitiva
1 viola propriedade Heap

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

Heap ajustado

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

5 movido para posição definitiva
1 viola propriedade Heap

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

Heap ajustado

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

4 movido para posição definitiva
0 viola propriedade Heap

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

Heap ajustado

HeapSort

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

Heap ajustado

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

3 movido para posição definitiva
1 viola propriedade Heap

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

Heap ajustado

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

2 movido para posição definitiva
0 viola propriedade Heap

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

Heap ajustado

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

1 movido para posição definitiva
0 não viola propriedade Heap

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

Vetor resultante

Heap Sort

- Complexidade de Caso Médio: $O(n \log_2 n)$
- Estável: Não – a propriedade Heap pode implicar no movimento de elementos de mesma chave.
- *In-place*: Sim.
- Online: Não – a aplicação do Heap requer conhecimento do maior elemento.

Algoritmos de tempo $O(n \log_2 n)$

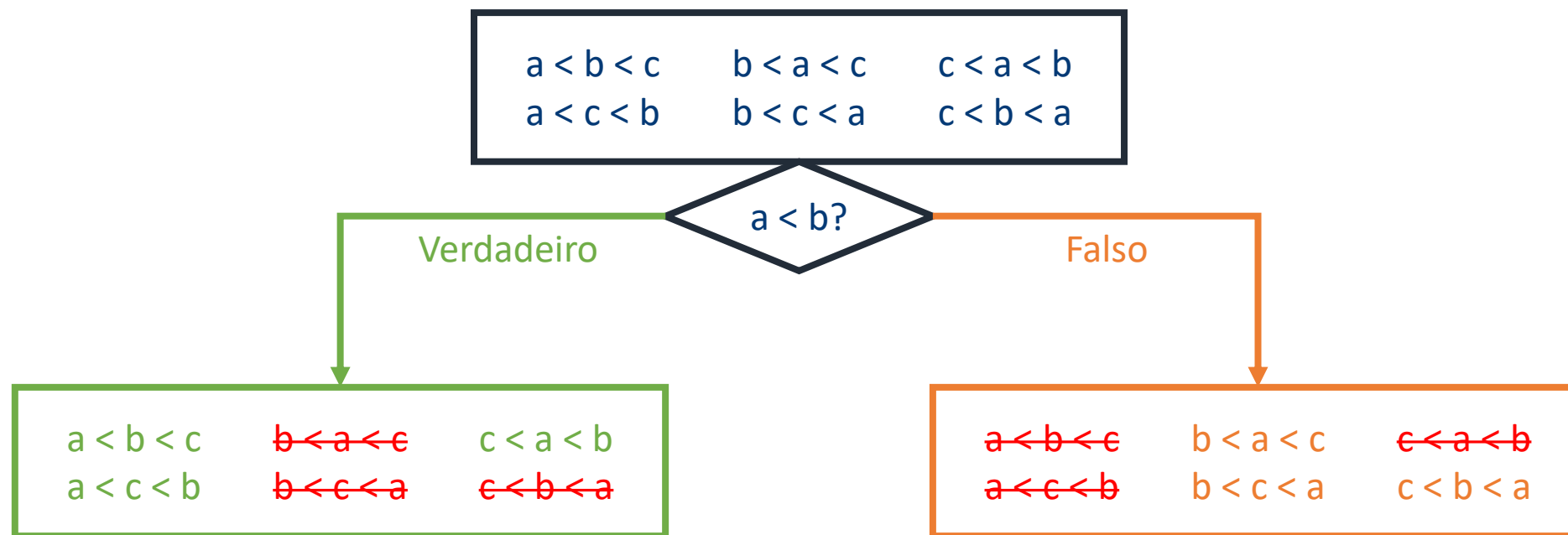
- Observamos três algoritmos de tempo $O(n \log_2 n)$:
 - MergeSort (no pior caso)
 - HeapSort (no pior caso)
 - QuickSort (no caso médio)
- Isso quer dizer que todos tem o mesmo desempenho, considerando tempo de execução em um mesmo computador?
 - Não.
- Na maioria dos casos, o QuickSort é mais rápido do que os demais.
 - MergeSort: Perde tempo com cópias da estrutura auxiliar
 - HeapSort: A construção e sucessivas correções do Heap torna o processo mais demorado.
- Entre HeapSort e MergeSort, o MergeSort é ligeiramente mais rápido.

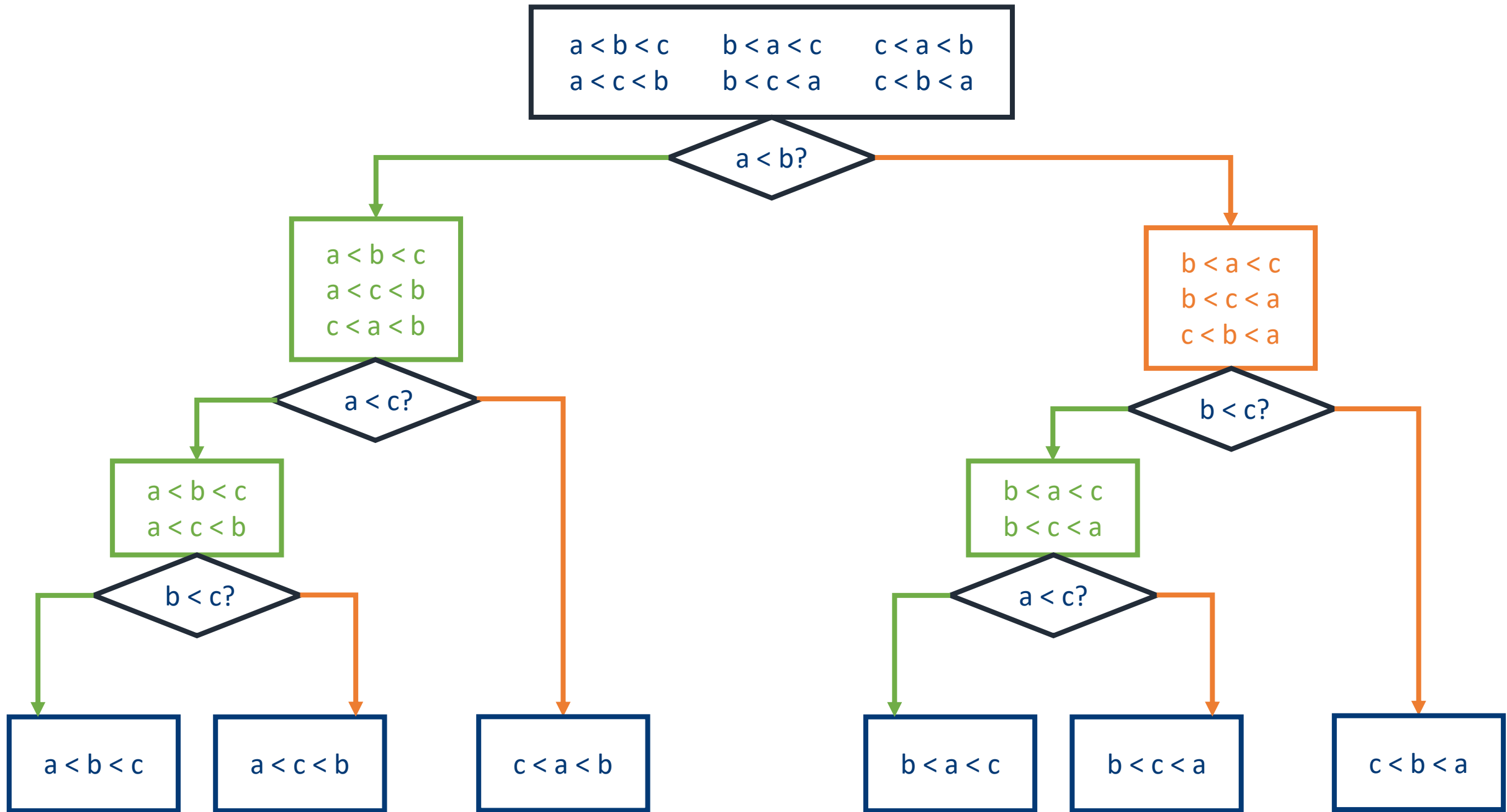
Algoritmos de tempo $O(n \log_2 n)$

- Existe algum algoritmo de ordenação com complexidade de pior caso inferior a $O(n \log_2 n)$?
 - Se o intervalo dos dados for conhecido, SIM.
 - Caso contrário, NÃO.
- Mas como afirmar com certeza absoluta que $O(n \log_2 n)$ é o limite mínimo para algoritmos de ordenação cujo intervalo de dados é desconhecido?

Limite Mínimo para Ordenação

- Existe um limite mínimo para ordenar um conjunto com n elementos?
- Vamos considerar uma árvore com cada pergunta que precisamos fazer para ordenar os elementos.
- Considerando a ordenação de três elementos a , b e c :





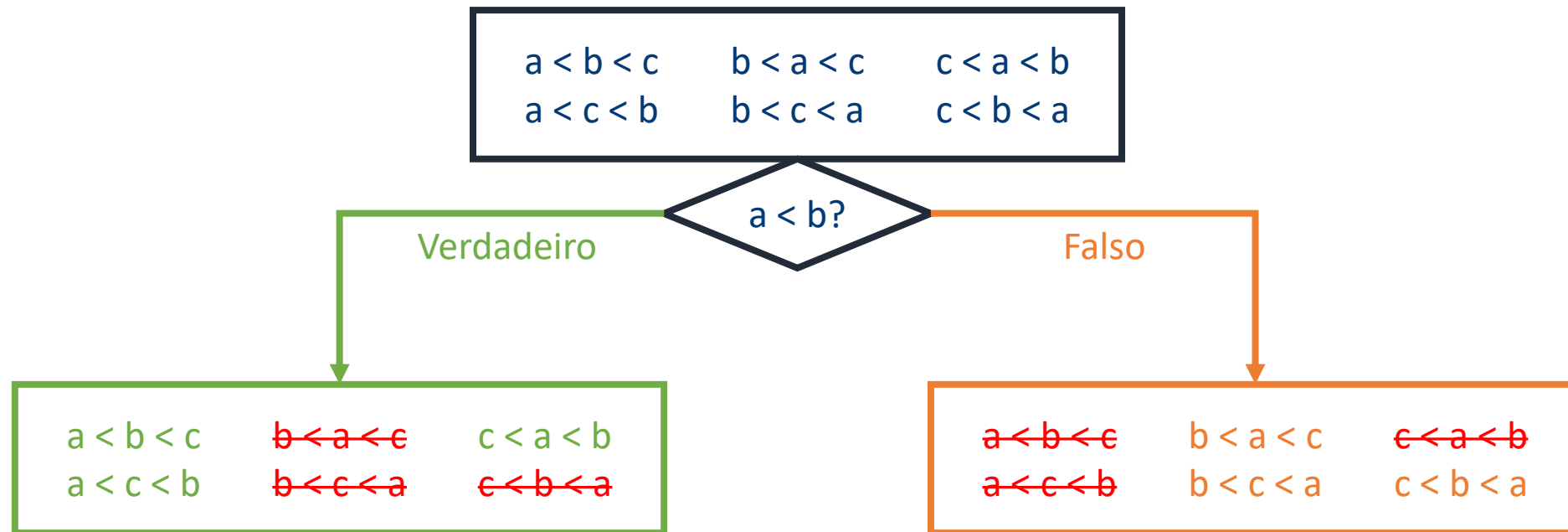
```
def ordenar(a,b,c):  
    if a < b:  
        if a < c:  
            if b < c:  
                return (a,b,c)  
            else:  
                return (a,c,b)  
        else:  
            return (c,a,b)  
    else:  
        if b < c:  
            if a < c:  
                return (b,a,c)  
            else:  
                return (b,c,a)  
        else:  
            return (c,b,a)
```

```
import itertools  
for l in list(itertools.permutations([1, 2, 3])):  
    print(f"{l} -> {ordenar(l[0],l[1],l[2])}")
```

(1, 2, 3) -> (1, 2, 3)
(1, 3, 2) -> (1, 2, 3)
(2, 1, 3) -> (1, 2, 3)
(2, 3, 1) -> (1, 2, 3)
(3, 1, 2) -> (1, 2, 3)
(3, 2, 1) -> (1, 2, 3)

Árvore de Combinações

- Se temos n itens, existem $n!$ possibilidades (3 itens = 6 permutações)
- A cada nível da árvore de combinações (a cada pergunta realizada) reduzimos as possibilidades de resposta em 50%:



Árvore de Combinações

- Se temos n itens, existem $n!$ possibilidades (3 itens = 6 permutações)
- A cada nível da árvore de combinações (a cada pergunta realizada) reduzimos as possibilidades de resposta em 50%
- Quantas perguntas vamos precisar fazer até conseguir a resposta?

$n!$	$\frac{n!}{2}$	$\frac{n!}{4}$	$\frac{n!}{8}$...	1
------	----------------	----------------	----------------	-----	---

Limite inferior para ordenação

- Precisamos saber quantos níveis a árvore de combinações vai ter!
- Seja h a altura dessa árvore
- $\frac{n!}{2^h} = 1 \rightarrow 2^h = n! \rightarrow h = \log_2 n!$
- Aproximando esse resultado, temos que $h \approx n \log_2 n$
- Isto é: Para ordenar n elementos, vamos precisar realizar **no mínimo** $n \log_2 n$ comparações.
- Logo, qualquer algoritmo de ordenação terá complexidade de no mínimo $O(n \log_2 n)$.

Ordenação em Tempo Linear

- É possível ordenar um conjunto com n elementos quaisquer em tempo linear?
- Não! O limite mínimo é $O(n \log_2 n)$.
- Mas e se conhecermos os elementos?
- Por exemplo:
 - Em uma turma com n alunos, onde cada aluno tem uma nota inteira de 0 até 10. Podemos ordenar os alunos por suas notas em tempo linear - $O(n)$?
 - Sim! Mas como?!?
- Quanto o **intervalo dos dados é conhecido**, podemos usar algoritmos de ordenação de tempo linear.

Bucket Sort (ou ordenação por caixas)

- Em uma turma com n alunos, onde cada aluno tem uma nota inteira de 0 até 10. Podemos ordenar os alunos por suas notas em tempo linear - $O(n)$?
- Ideia:
- Vamos criar 10 caixas:
 - Caixa para alunos com nota 0
 - Caixa para alunos com nota 1
 - ...
 - Caixa para alunos com nota 10
- A cada aluno da turma, identificamos a caixa e inserimos o aluno

Bucket Sort (ou ordenação por caixas)

- Uma vez que cada aluno foi inserido em uma caixa, vamos ordenar a turma.
- Turma = [Alunos da Caixa 0] + [Alunos da Caixa 1] + ... [Alunos da Caixa 10]

```
def Ordenar(listaAlunos):  
    caixas = [ [] for i in range(11) ]  
    for a in listaAlunos:  
        caixas[a.nota].append(a)  
  
    resultado = []  
    for i in range(11):  
        for a in caixas[i]:  
            resultado.append(a)  
    return resultado
```

Criar caixas vazias para cada nota de 0 até 10

Para cada aluno:
Inserir o aluno na caixa associada a sua nota

Juntar os alunos armazenados nas caixas
Começando da Caixa 0 até a Caixa 10

Retornar a lista ordenada

Bucket Sort (ou ordenação por caixas)

- Importante: A ordenação por caixas (ou qualquer outro método de ordenação em tempo linear) requer um conhecimento prévio dos dados de entrada.
- No exemplo dos alunos: Já sabíamos que existiam apenas 11 categorias (uma para cada nota).
- Se as notas estivessem no universo dos números reais (um infinito de possibilidades de notas), não seria possível utilizar a ordenação por caixas.

Bucket Sort

- Complexidade: $O(n)$
- Estável: Sim – mas depende da implementação.
- *In-place*: Não – ele requer uma estrutura auxiliar para armazenar os “baldes”.
- Online: Não.

Outros algoritmos de ordenação

- Existem muitos outros algoritmos de ordenação.
- A Wikipedia contém a definição de 46 métodos de ordenação, de vários tipos.
- Vale conhecer mais alguns:
 - RadixSort
 - CountingSort
 - TimSort
 - ShellSort