PANC: Projeto e Análise de Algoritmos

Aula 12: Algoritmos de Ordenação III - Seleção:

Select Sort e Heap Sort

e outros Algoritmos:

Merge Sort e RadixSort

Breno Lisi Romano

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Sumário

- Revisão de Conteúdo
- Introdução
- Algoritmos de Ordenação por Seleção:
  - Select Sort
  - Heap Sort
- Outros Algoritmos de Ordenação:
  - Merge Sort
  - Radix Sort
- Exemplos Práticos

Recapitulando...

- Ordenação por Inserção:
  - Caracterizados pelo princípio no qual se divide o array em dois segmentos, sendo um já ordenado e o outro a ser ordenado
  - A partir disto, iterações são desenvolvidas sendo que, em cada uma delas, um elemento do segmento não ordenado é transferido para o segmento ordenado, na sua posição correta
  - Métodos de Ordenação por Inserção:
    - Direta: $T(n) = O(n^2)$
    - Binária: $T(n) = \lg (n-1)!$
    - Shell Sort: $T(n)$ = Conjectura 1: $O(n^{1,25})$ e Conjectura 2: $O(n (\lg n)^2)$
- Ordenação por Troca:
  - Diferentemente dos Métodos de Ordenação por Inserção, os Métodos de Ordenação por Troca fazem a permutação dos valores de um array para buscar a ordenação do mesmo
  - Métodos de Ordenação por Troca:
    - Bubble Sort: $T(n) = O(n^2)$
    - Shake Sort: $T(n) = O(n^2)$
    - Comb Sort: $T(n) = O(n^2)$
    - Quick Sort: $T(n) = O(n \lg n)$

Método de Ordenação por Seleção

- Esta classe de ordenação é composta por algoritmos que constroem a sequência ordenada com um elemento de cada vez
- A cada passo, o elemento a ser adicionado a sequência ordenada é selecionado entre os elementos restantes, e o array a ser classificado fica reduzido de um elemento
- Os dois principais métodos por Seleção são:
  - Select Sort
  - Heap Sort

Detalhando...

ALGORITMOS DE ORDENAÇÃO POR SELEÇÃO

Select Sort e Heap Sort

Ordenação por Seleção: Select Sort

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Met. Ord. Seleção: Select Sort (1)

- Neste método, seleciona-se o menor elemento por pesquisa sequencial
- Uma vez encontrada a menor chave, ela é permutada com a que ocupa a posição inicial do array, que fica reduzido de um elemento
- O processo de seleção é repetido para a parte restante do array, sucessivamente, até que todos os elementos tenham sido selecionados e colocados em suas posições definitivas

Met. Ord. Seleção: Select Sort (2)

- Os passos necessários à execução destas tarefas são enumeradas a seguir:

1. Selecionar o elemento que representa o menor valor

2. Trocá-lo com o primeiro elemento do array

3. Repetir estas operações, envolvendo agora apenas N-1 elementos restantes, depois os N-2 elementos, etc... até restar um só elemento, o maior deles

Met. Ord. Seleção: Select Sort (3)

- Simule a ordenação do seguinte array utilizando o Select Sort:

i	0	1	2	3	4	5	6	7
Vet[i]	44	55	12	42	92	18	06	67

Solução:

Iteração	0	1	2	3	4	5	6	7	Resultado
1	44	55	12	42	92	18	06	67	Troca(Vet[0], Vet[6])
2	06	55	12	42	92	18	44	67	Troca(Vet[1],Vet[2])
3	06	12	55	42	92	18	44	67	Troca(Vet[2],Vet[5])
4	06	12	18	42	92	55	44	67	Troca(Vet[3],Vet[3])
5	06	12	18	42	92	55	44	67	Troca(Vet[4],Vet[6])
6	06	12	18	42	44	55	92	67	Troca(Vet[5],Vet[5])
7	06	12	18	42	44	55	92	67	Troca(Vet[6],Vet[7])
Array Ordenado	06	12	18	42	44	55	67	92	Não precisa ordenar a última posição, pois nela já se encontra o maior valor

Met. Ord. Seleção: Select Sort (2)

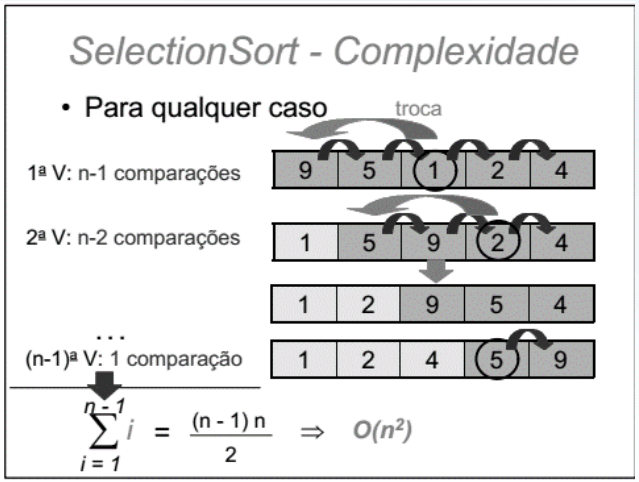
```
/*OrdenaSelecao(): Função que Ordena o array Vet[Max] aplicando o Método da
                   Família de Classificação por Seleção chamado de Select Sort. */
void OrdenaSelecao(int Vet[])
{
    //Variaveis Locais
  int i, j, k, x, Comp;

  for(i=0; i<Max-1; i++)
  {
    Comp=0;
    k=i;
    x = Vet[i];
    for(j=i+1; j< Max; j++)
    {
      if(Vet[j] < x)
      {
        k = j;
        x = Vet[k];
        Comp = 1;
      }
    }
    if(Comp == 1)
    {
                Vet[k] = Vet[i];
                Vet[i] = x;
    }
  }
}
```

Ordenação por Select Sort: Análise da Complexidade

- Principais pontos a se destacar:
  - Realiza sempre uma busca sequencial pelo menor valor no segmento não ordenado a cada iteração

- Análise da Complexidade do Algoritmo:
  - **Todos os Casos (Melhor, Pior e Caso Médio):**
    - Definida pelo número de comparações envolvendo a quantidade de dados do Array
    - Número de comparações: (n-1) + (n-2) + (n-3) + ... + 2 + 1
    - Complexidade T(n): $\sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \rightarrow O(n^2)$



Ordenação por Seleção: Heap Sort

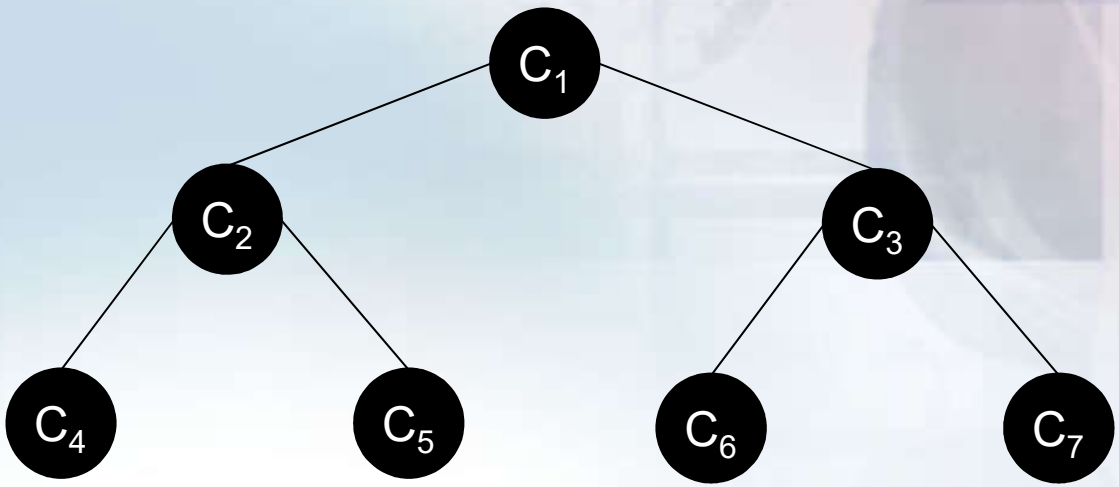
INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

Met. Ord. Seleção: Heap Sort (1)

- HeapSort também é um método de seleção:
  - Ordena através de sucessivas seleções do elemento correto a ser posicionado em um segmento ordenado
- Proposto por J. Williams (1964) apresentando uma melhoria drástica em relação aos métodos mais convencionais de ordenação
- O HeapSort utiliza um **Heap Binário** para manter o próximo elemento a ser selecionado:
  - Heap Binário: árvore binária mantida na forma de array
  - O Heap é gerado e mantido no próprio array a ser ordenado (no segmento não ordenado)

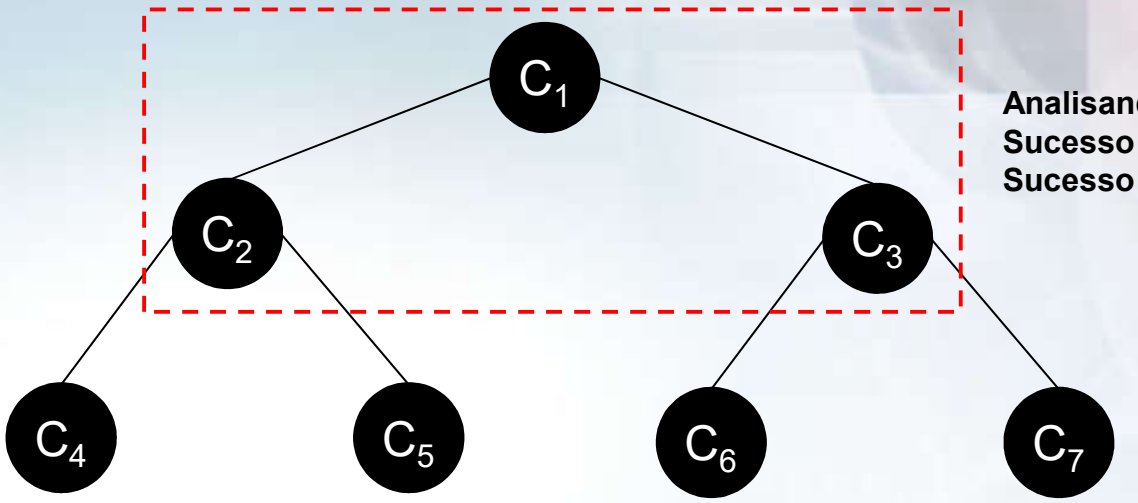
Met. Ord. Seleção: Heap Sort (2)

- Árvore representada pelo array A [] = {C_1, C_2, C_3, C_4, ..., C_7}



Met. Ord. Seleção: Heap Sort (3)

- O **"Heap"** é definido como sendo uma sequência de elementos C_L, C_{L+1}, C_{L+2}, ... C_R tal que as seguintes regras sejam observadas:
  - **Regra 01:** C_i representa a raiz da árvore
  - **Regra 02:** C_{2i} representa o sucessor a esquerda de C_i (se $C_{2i} \leq C_i$) e ($2_i < n$)
  - **Regra 03:** C_{2i+1} representa o sucessor a direita de C_i (se $C_{2i+1} \leq C_i$) e ($2_{i+1} < n$)



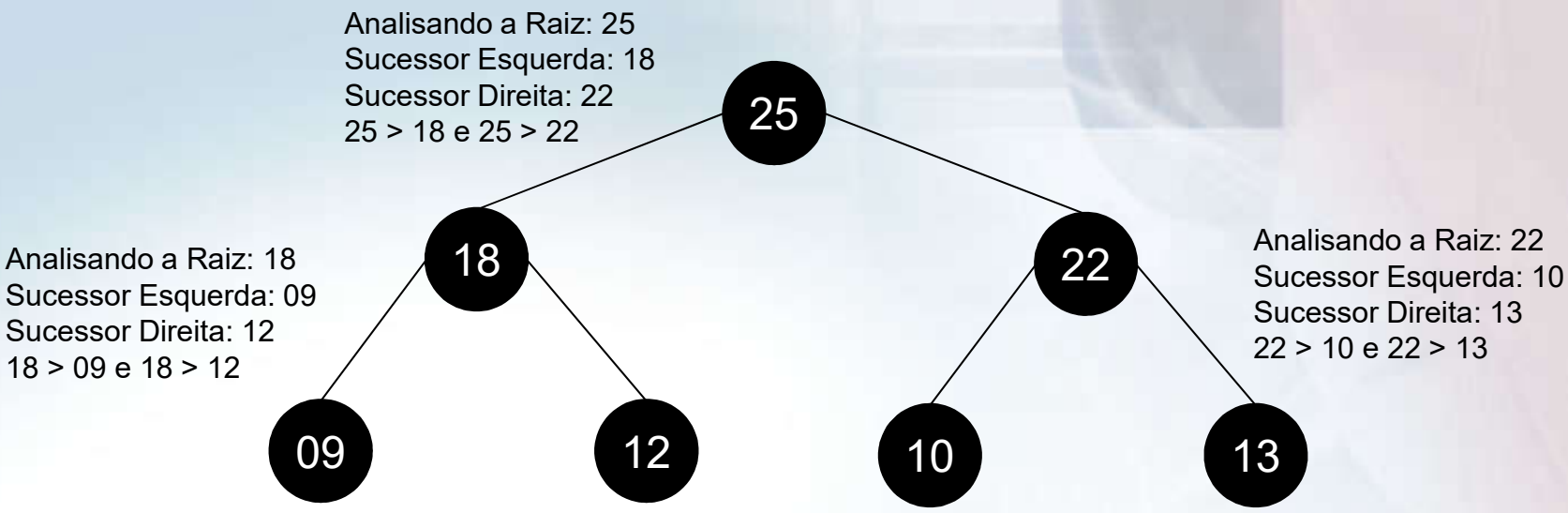
Analisando a Raiz: C_1
Sucesso Esquerda: C_2 (C_{2i})
Sucesso Direita: C_3 (C_{2i+1})

Como todas as raízes das subárvores satisfazem as condições $C_{2i} \leq C_i$ e $C_{2i+1} \leq C_i$ → Árvore é um Heap!

Met. Ord. Seleção: Heap Sort (4)

- Exemplo de um **"Heap"** para o Array A[]:

i	1	2	3	4	5	6	7
A[]	12	09	13	25	18	10	22



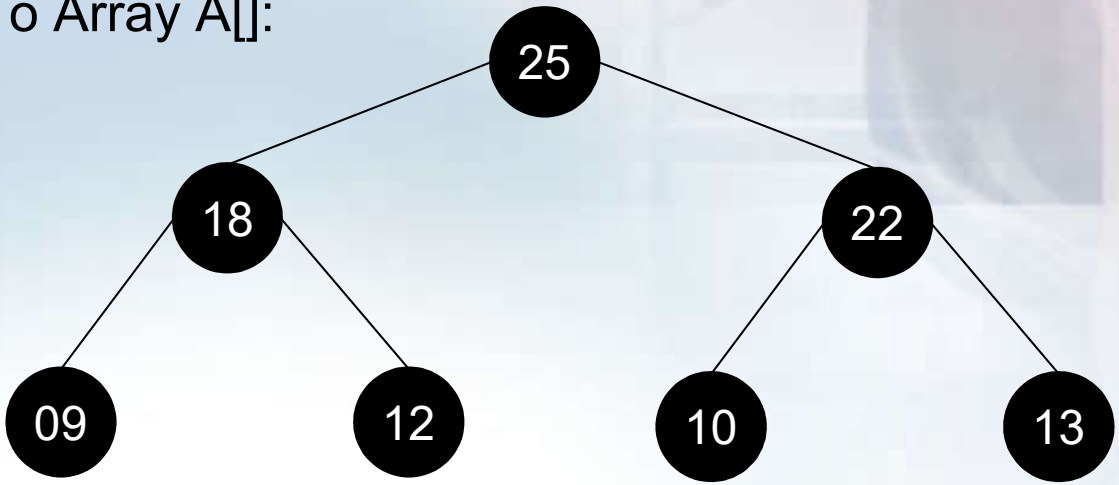
Observe que o maior elemento do Array está na Raiz da Árvore!

Met. Ord. Seleção: Heap Sort (5)

- Array Inicial antes de transformar em um Heap:

i	1	2	3	4	5	6	7
A[]	12	09	13	25	18	10	22

- Heap para o Array A[]:



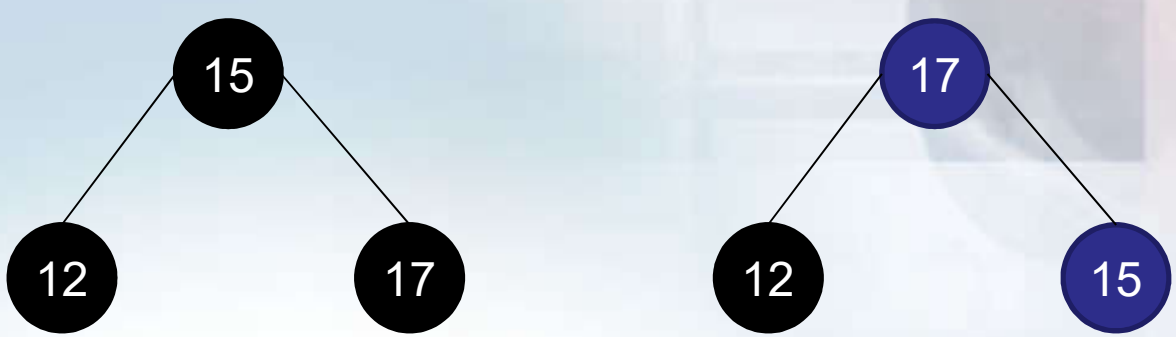
- Nova representação do Array A[] após a transformação para um Heap:

i	1	2	3	4	5	6	7
A[]	25	18	22	09	12	10	13

Met. Ord. Seleção: Heap Sort (6)

- Sempre que for encontrada uma subárvore que não forme um Heap, seus componentes devem ser rearranjados de modo a formar um Heap:

- Exemplo de uma transformação de Árvore em Heap:



- Pode ocorrer também que ao rearranjar uma subárvore isto venha a afetar outra subárvore diretamente relacionada a ela → A Árvore Completa deixa de ser Heap
  - **Obriga** a verificação de que, sempre que for rearranjada uma subárvore, se a sucessora do nível abaixo não teve a sua condição de Heap desfeita

Met. Ord. Seleção: Heap Sort Algoritmo

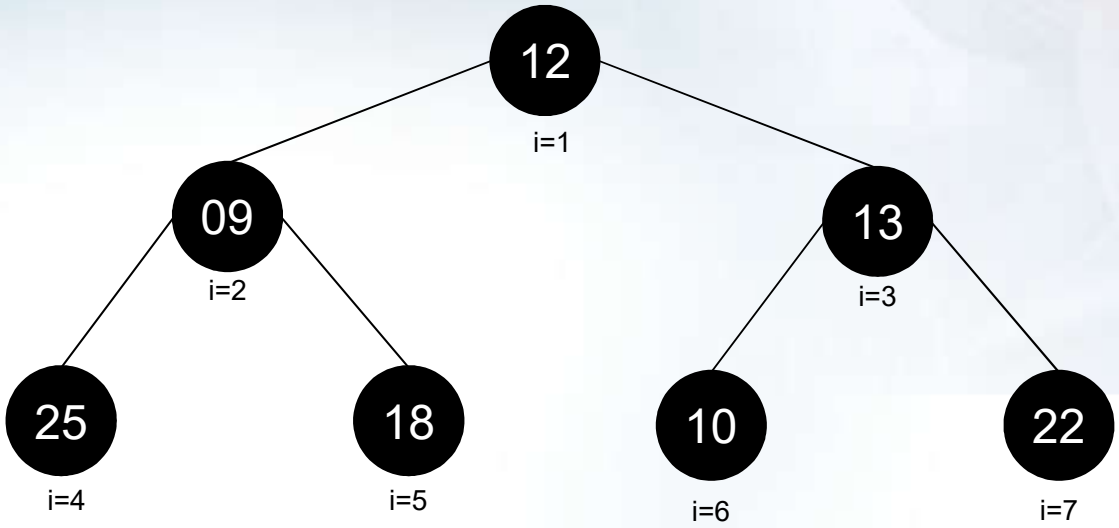
- **Algoritmo do HeapSort é apresentado com 02 passos:**
  1. **Uma árvore binária (Heap) é construída com todos os elementos do Array**
     - Sabe-se que em uma árvore binária, o valor contido em qualquer nó é maior do que os valores de seus sucessores
     - A árvore binária é estruturada no próprio array da seguinte maneira:
       - Sucessor à esquerda de A[i]: A[2i] (se 2i < n)
       - Sucessor à direita de A[i]: A[2i+1] (se 2i+1 < n)
     - A árvore é transformada em um Heap
       - Nesta transformação, que é realizada do maior nível da árvore até a raiz, troca-se cada nó com o maior de seus sucessores imediatos
       - Este passo é repetido até que cada nó seja maior que seus sucessores imediatos
  2. **Após a construção do Heap, é realizada a ordenação propriamente dita**
     - O valor que está na raiz da árvore é o maior valor contido em toda árvore → É colocado na sua posição correta, trocando-o com o elemento de maior índice da árvore
       - A árvore fica com um elemento a menor
     - Existe a possibilidade desse novo elemento colocado na raiz violar a propriedade do Heap
       - Se sim, é necessário restaurar o Heap novamente
     - Repete este procedimento até que a árvore fique com um único elemento

Met. Ord. Seleção: Heap Sort Exemplo Ilustrativo (1)

- Ordenar em ordem crescente o Array A[]:

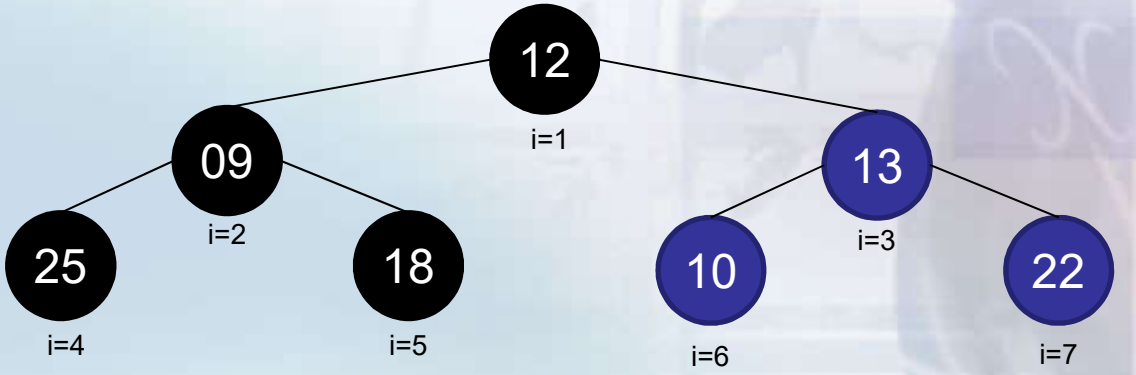
i	1	2	3	4	5	6	7
A[]	12	09	13	25	18	10	22

- Primeiro Passo: Representar o Array Inicial como uma Árvore Binária:
  - Sucessor à esquerda de A[i]: A[2i] (se 2i < n)
  - Sucessor à direita de A[i]: A[2i+1] (se 2i+1 < n)

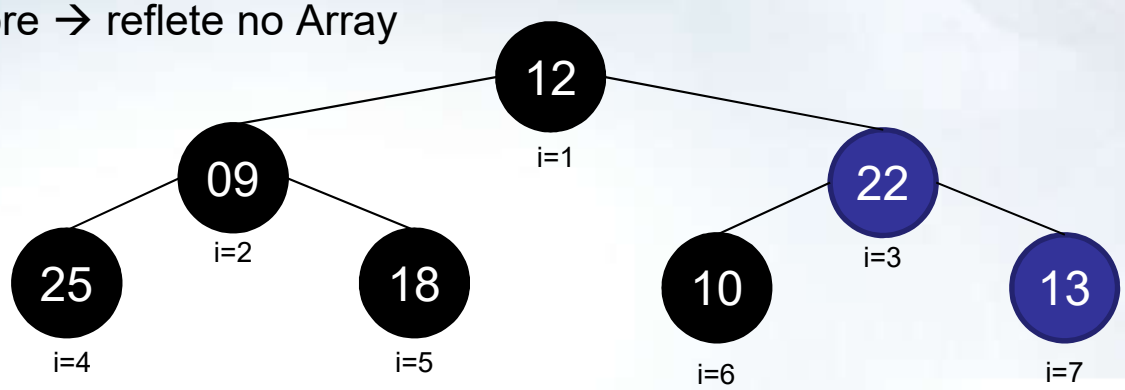


Met. Ord. Seleção: Heap Sort
Exemplo Ilustrativo (2)

- **Busca da primeira maior chave:**
  - A transformação desta árvore em um Heap é iniciada pela subárvore cuja raiz é 13



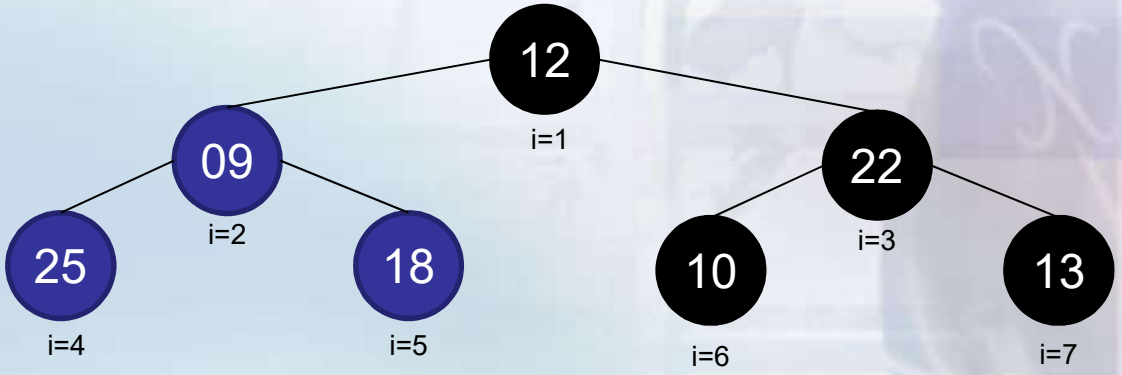
  - Comparamos a raiz da subárvore 13 com seus sucessores a esquerda e direita verificando qual é o maior → 22 é o maior e deve ser trocado de posição com o 13 na árvore → reflete no Array



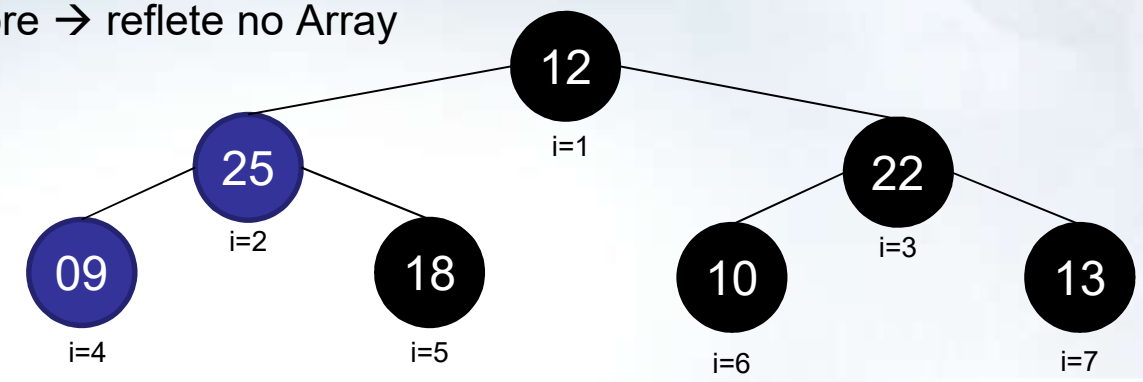
i	1	2	3	4	5	6	7
A[]	12	09	**22**	25	18	10	**13**

Met. Ord. Seleção: Heap Sort
Exemplo Ilustrativo (3)

- **Busca da primeira maior chave:**
  - Próxima transformação é rearranjar a subárvore cuja raiz é 09



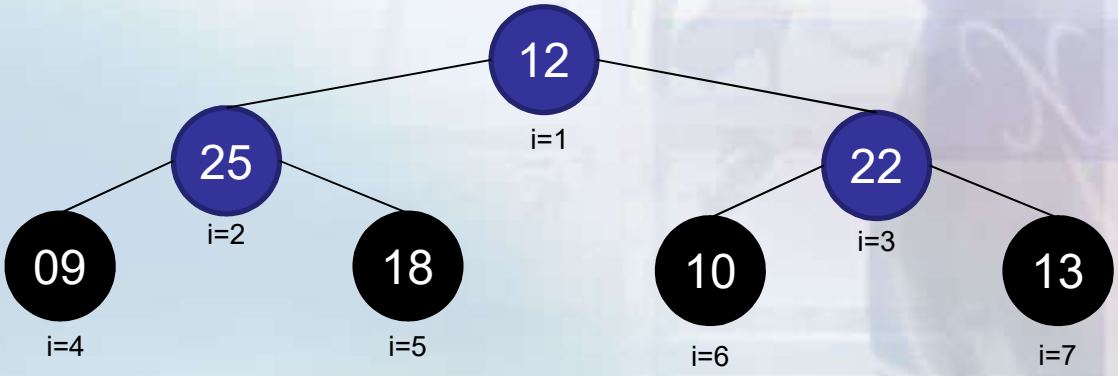
  - Comparamos a raiz da subárvore 09 com seus sucessores a esquerda e direita verificando qual é o maior → 25 é o maior e deve ser trocado de posição com o 09 na árvore → reflete no Array



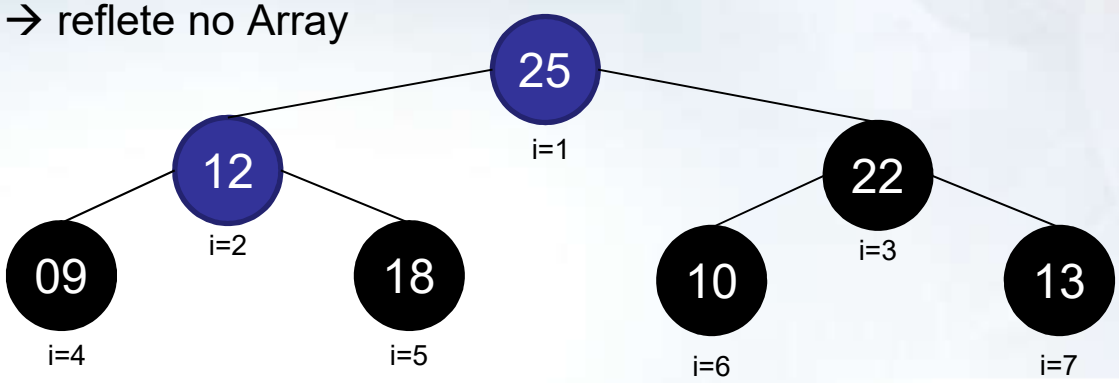
i	1	2	3	4	5	6	7
A[]	12	**25**	22	**09**	18	10	13

Met. Ord. Seleção: Heap Sort
Exemplo Ilustrativo (4)

- **Busca da primeira maior chave:**
  - Próxima transformação é rearranjar a subárvore cuja raiz é 12



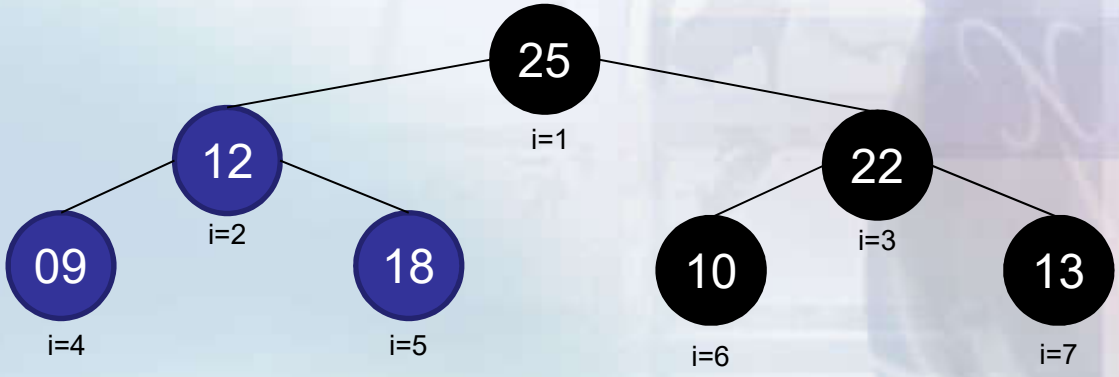
  - Comparamos a raiz da subárvore 12 com seus sucessores a esquerda e direita verificando qual é o maior → 25 é o maior e deve ser trocado de posição com o 12 na árvore → reflete no Array



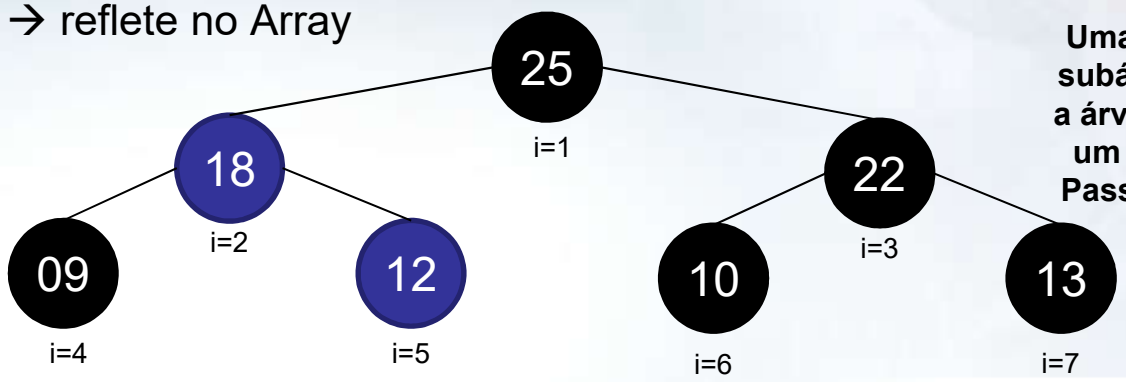
i	1	2	3	4	5	6	7
A[]	25	12	22	09	18	10	13

Met. Ord. Seleção: Heap Sort
Exemplo Ilustrativo (5)

- **Busca da primeira maior chave:**
  - A última transformação afetou uma subárvore → Deve rearranjar a subárvore em questão



  - Comparamos a raiz da subárvore 12 com seus sucessores a esquerda e direita verificando qual é o maior → 18 é o maior e deve ser trocado de posição com o 12 na árvore → reflete no Array

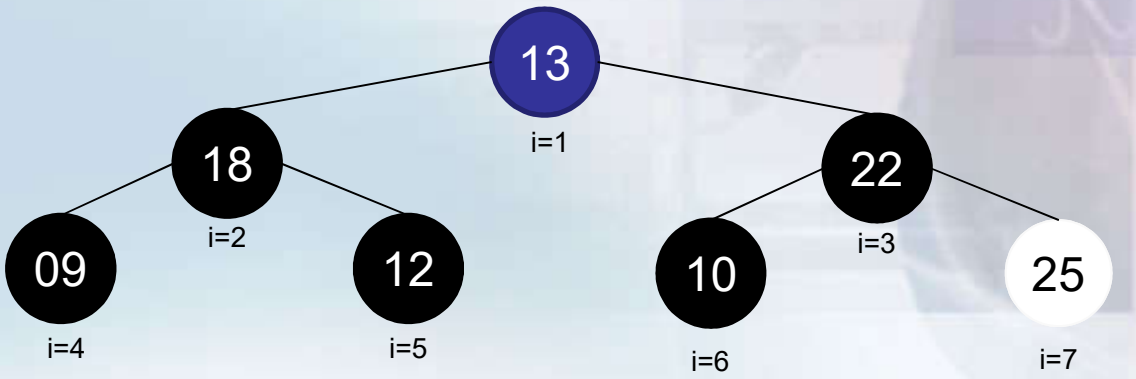


Uma vez que todas as subárvores são Heaps, a árvore toda também é um Heap → Executar Passo 02 do Algoritmo

i	1	2	3	4	5	6	7
A[]	25	18	22	09	12	10	13

Met. Ord. Seleção: Heap Sort
Exemplo Ilustrativo (6)

- **Busca da primeira maior chave:**
  - Elemento da raiz é o maior elemento do Array → Posição correta é no final do Array
    - Deve-se trocar o elemento da raiz com o elemento da última posição do Array → Trocar A[1] com A[n]
    - Após esta troca, consideramos a árvore com n-1 elementos



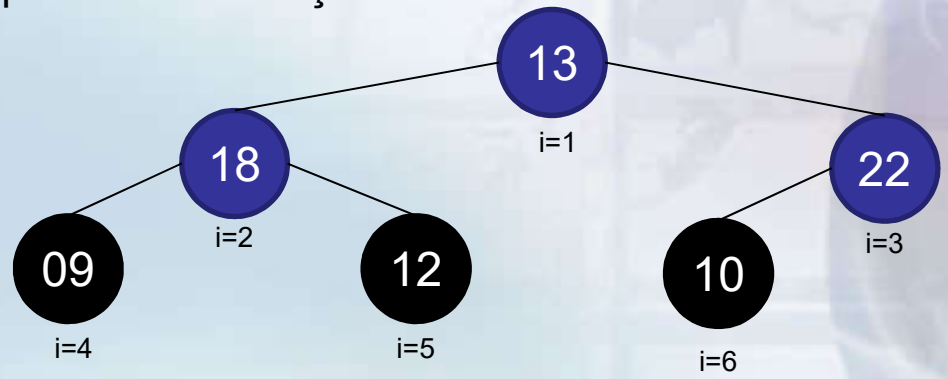
i	1	2	3	4	5	6	7
A[]	13	18	22	09	12	10	25

Array Ordenado

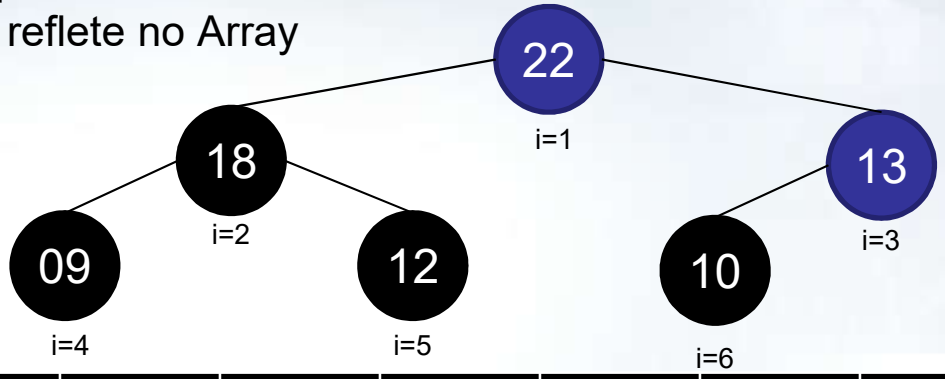
  - Com este procedimento, encerramentos a primeira fase do Algoritmo HeapSort
  - Fazemos o mesmo procedimento para encontrar os maiores elementos restantes da árvore, desconsiderando a última posição do Array

Met. Ord. Seleção: Heap Sort
Exemplo Ilustrativo (7)

- **Busca da segunda maior chave:**
  - Devemos fazer com que a árvore volte a ser um Heap → Manipular apenas a subárvore que sofreu alteração



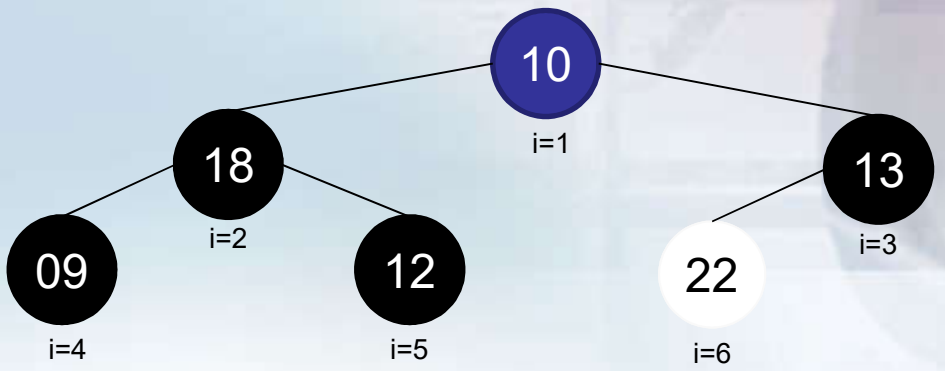
  - Comparamos a raiz da subárvore 13 com seus sucessores a esquerda e direita verificando qual é o maior → 22 é o maior e deve ser trocado de posição com o 13 na árvore → reflete no Array



i	1	2	3	4	5	6	7
A[]	22	18	13	09	12	10	**25**

Met. Ord. Seleção: Heap Sort Exemplo Ilustrativo (8)

- **Busca da segunda maior chave:**
  - Elemento da raiz é o maior elemento do Array → Posição correta é no final do Array
    - Deve-se trocar o elemento da raiz com o elemento da última posição do Array → Trocar A[1] com A[n]
    - Após esta troca, consideramos a árvore com n-1 elementos



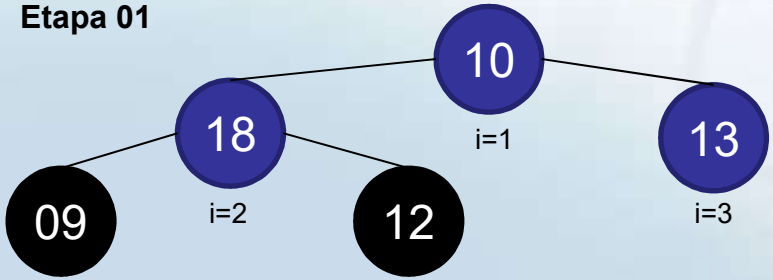
i	1	2	3	4	5	6	7
A[]	10	18	13	09	12	22	25

Array Ordenado

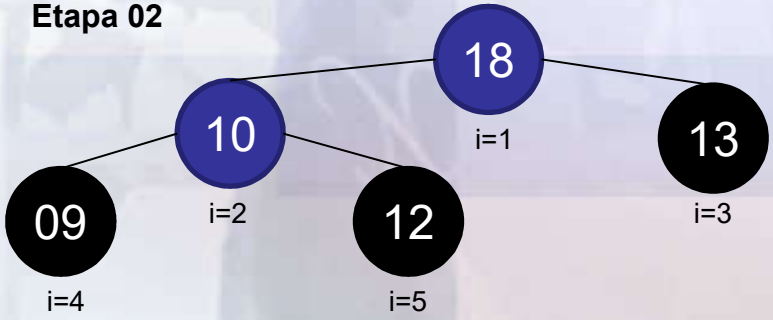
Met. Ord. Seleção: Heap Sort
Exemplo Ilustrativo (8)

- **Busca da terceira maior chave (Resumido):**

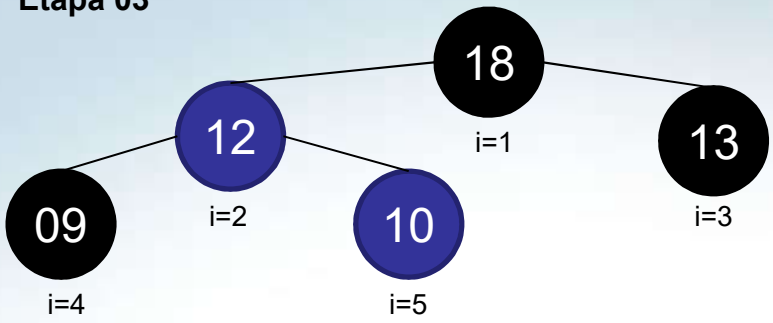
Etapa 01



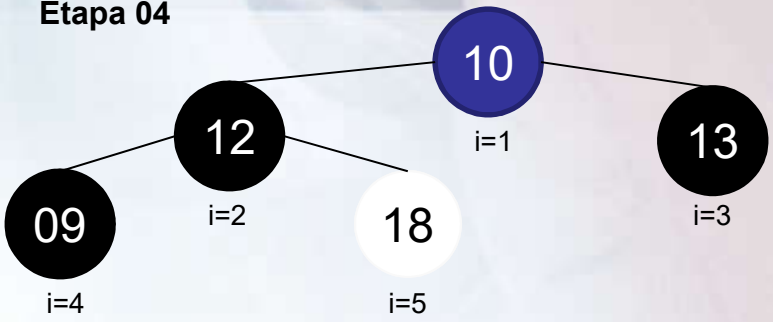
Etapa 02



Etapa 03



Etapa 04



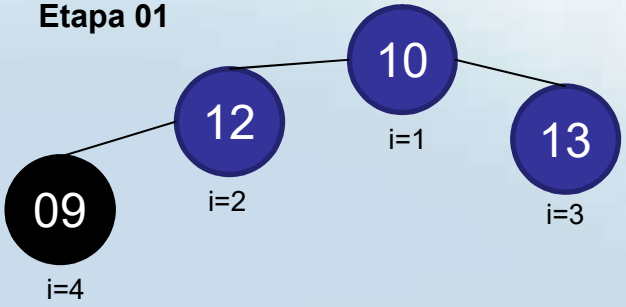
i	1	2	3	4	5	6	7
A[]	10	12	13	09	**18**	**22**	**25**

Array Ordenado

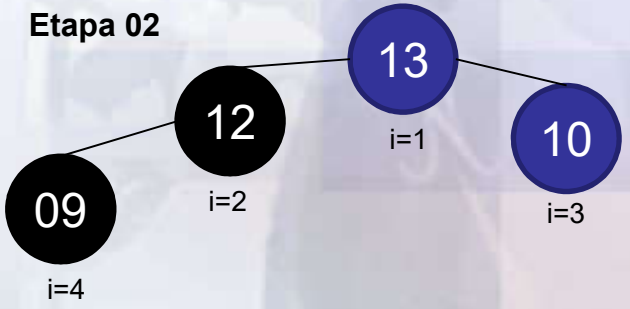
Met. Ord. Seleção: Heap Sort
Exemplo Ilustrativo (8)

- **Busca pelos demais elementos (Resumido):**

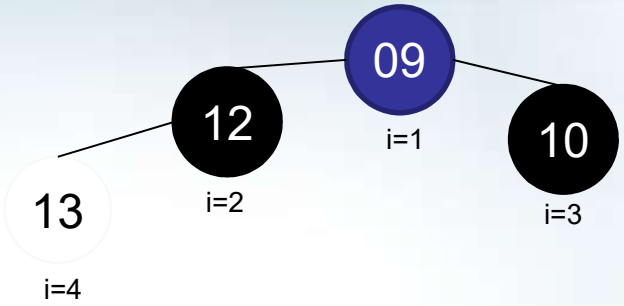
Etapa 01



Etapa 02



Etapa 03

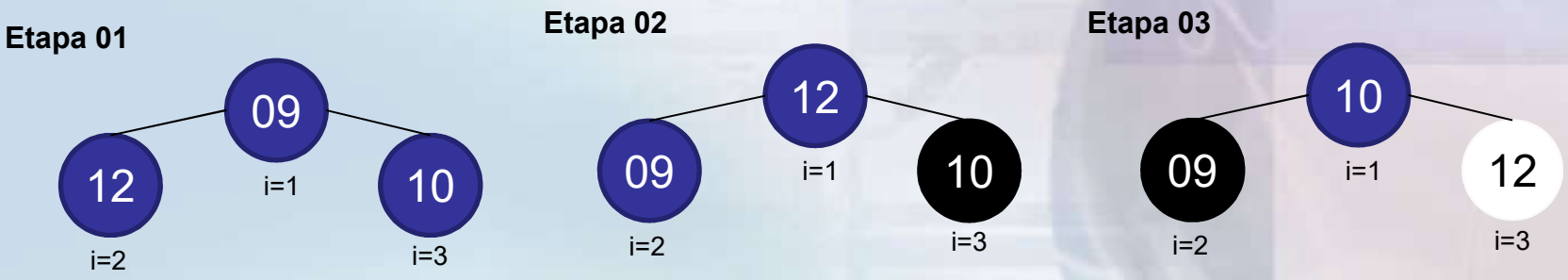


i	1	2	3	4	5	6	7
A[]	09	12	10	**13**	**18**	**22**	**25**

Array Ordenado

Met. Ord. Seleção: Heap Sort
Exemplo Ilustrativo (9)

- **Busca pelos demais elementos (Resumido):**



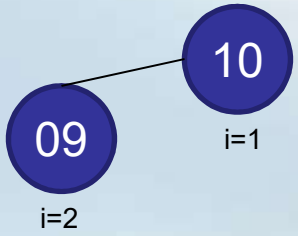
i	1	2	3	4	5	6	7
A[]	10	09	**12**	**13**	**18**	**22**	**25**

Array Ordenado

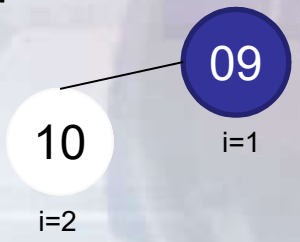
Met. Ord. Seleção: Heap Sort Exemplo Ilustrativo (10)

- **Busca pelos demais elementos (Resumido):**

Etapa 01



Etapa 02



i	1	2	3	4	5	6	7
A[]	09	**10**	**12**	**13**	**18**	**22**	**25**

Array Ordenado

Met. Ord. Seleção: Heap Sort Exemplo Ilustrativo (11)

- **Busca pelos demais elementos (Resumido):**

Etapa 01

i=1

i	1	2	3	4	5	6	7
A[]	09	**10**	**12**	**13**	**18**	**22**	**25**

Array Ordenado

Met. Ord. Seleção: Heap Sort() Algoritmo Principal

```
/*OrdenaHeap(): Função que Ordena o array Vet[Max] aplicando o Método da
                Família de Classificação por Seleção chamado de Heap Sort. */
void OrdenaHeap(int Vet[])
{
    //Variaveis Locais
  int L, R, x;

  L = Max/2+1;
  R = Max-1;

  while(L > 0)
  {
    L--;
    Heap(L, R, Vet);
  }

  while(R > 0)
  {
    x = Vet[0];
    Vet[0] = Vet[R];
    Vet[R] = x;
    R--;
    Heap(L, R, Vet);
  }
}
```

Met. Ord. Seleção: Heap Sort()
Algoritmo Auxiliar para Montar o Heap

```
/*Heap(): Função Auxiliar a Função OrdenaHeap, que ajuda a Ordenar o array
          Vet[Max] aplicando o Método da Família de Classificação por Seleção
          chamado de Heap Sort*/
void Heap(int L, int R, int Vet[])
{
  //Variaveis Locais
  int i, j, x;

  //Inicializacao das Variaveis
  i = L;
  j = 2*L;
  x = Vet[L];

  if((j<R)&&(Vet[j]<Vet[j+1])) j++;
  while((j<=R)&&(x<Vet[j]))
  {
    Vet[i] = Vet[j];
    i = j;
    j = 2*j;
    if((j<R)&&(Vet[j]<Vet[j+1])) j++;
  }
  Vet[i] = x;
}
```

Ordenação por HeapSort: Análise da Complexidade

- Etapas:
  - Construção do Heap → O(n)
    - Número de Comparações está sempre dentro do intervalo de (n, 2n)
  - Ordenação:
    - Troca (raiz com o final do segmento não ordenado) → O(1)
    - Ajuste do novo elemento raiz para transformar a Árvore em Heap novamente → O(lg n)
      - Executa até que o elemento seja transferido para a posição i > n/2

Executa-se n-1 vezes → (n-1) . lg n → O (n. lg n)

  - **Análise da Complexidade do Algoritmo (para qualquer caso):**
    - **T(n) = O (n. lg n)**

Detalhando...

OUTROS ALGORITMOS DE ORDENAÇÃO

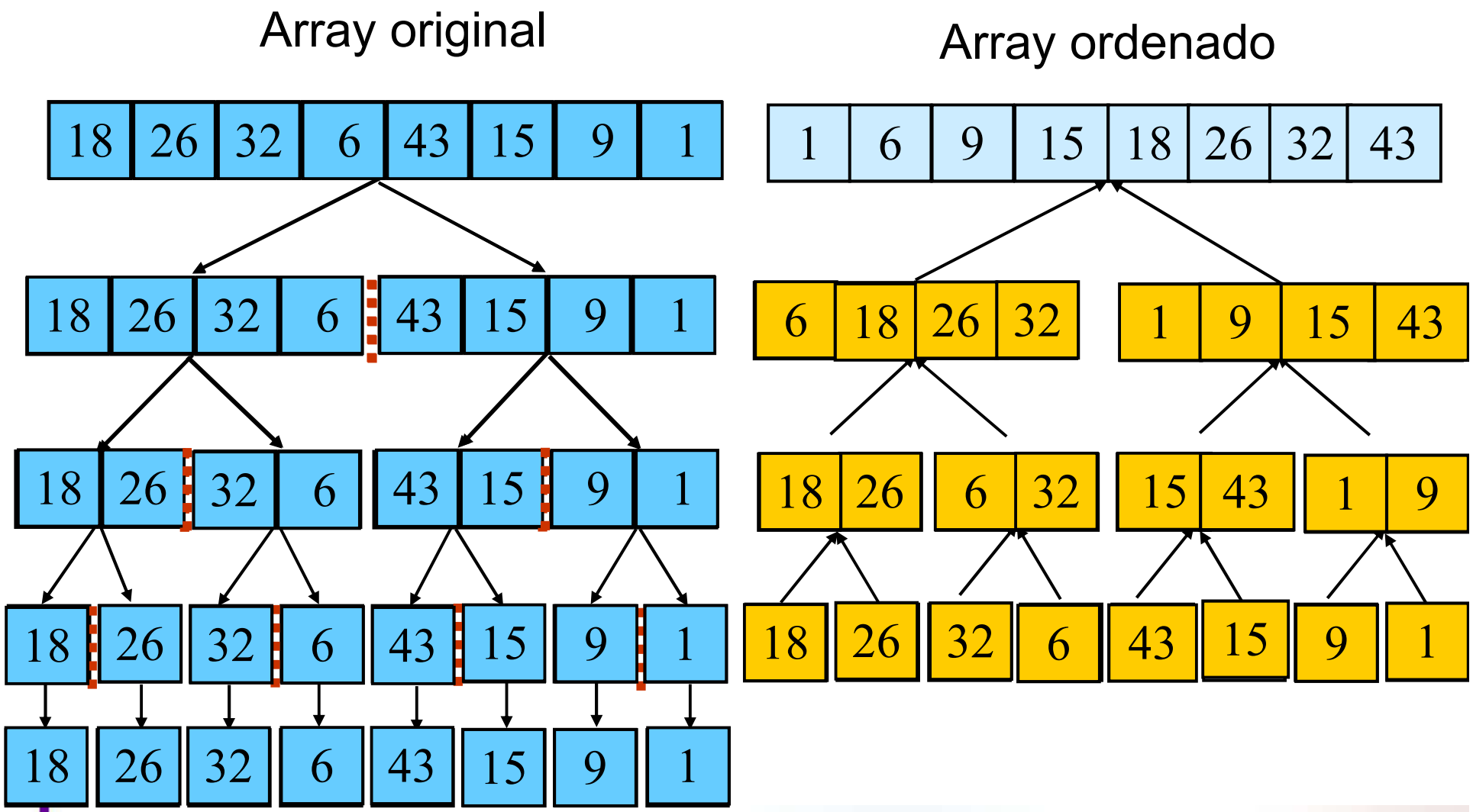
Merge Sort e Radix Sort

Outros Algoritmos de Ordenação: Merge Sort

Merge Sort: Divisão e Conquista

- *Mergesort* é um **algoritmo** para resolver o problema de **ordenação de arrays** e um exemplo clássico do **uso do paradigma de Divisão e Conquista** (*to merge* = intercalar)
  - Duas abordagens: **Top-Down (Recursiva)** e Bottom-Up (Iterativa)
- **Descrição** do ***Mergesort*** em alto nível (**Top-Down**):
  - **Divisão**: Divide a sequência de n elementos que deve ser ordenada em duas subsequências de n/2 elementos cada uma
  - **Conquista**: Ordena as duas subsequências recursivamente, utilizando a ordenação por intercalação (Algoritmo Mergesort)
  - **Combinação**: Intercala as duas subsequências ordenadas para produzir a resposta ordenada (Algoritmo Intercala)
  - **Condição de parada da Recursão**: quando for ordenar apenas um elemento, este caso será a sub-solução elementar

Merge Sort: Ilustração para um Array (n=8)



Merge Sort: Algoritmo

- ***Mergesort***: O objetivo é rearranjar um array A[p...r], com p ≤ *r, em* ordem crescente

```
MERGESORT(A, p, r)
1   se p < r                                   Divisão
2       então q ← ⌊(p + r)/2⌋
3           MERGESORT(A, p, q)                 Conquista
4           MERGESORT(A, q + 1, r)
5           INTERCALA(A, p, q, r)              Combinação
```

Merge Sort: Algoritmo Intercala()

- O que significa intercalar dois (sub)arrays ordenados?
- **Problema:** Dados A[p...q] e A[q+1...r] crescentes, rearranjar A[p...r] de modo que ele fique em ordem crescente

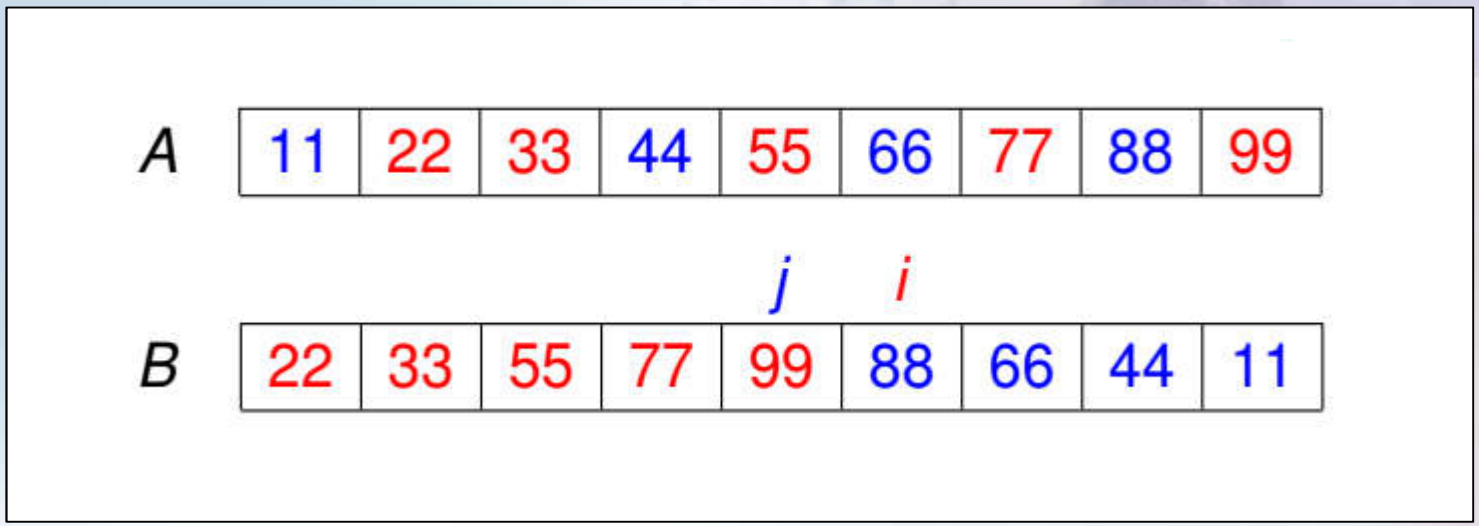
Entrada:

	p				*q*				*r*
A	22	33	55	77	99	11	44	66	88

Saída:

	p				*q*				*r*
A	11	22	33	44	55	66	77	88	99

Merge Sort: Algoritmo Intercala() – Sem Sentinela (1)

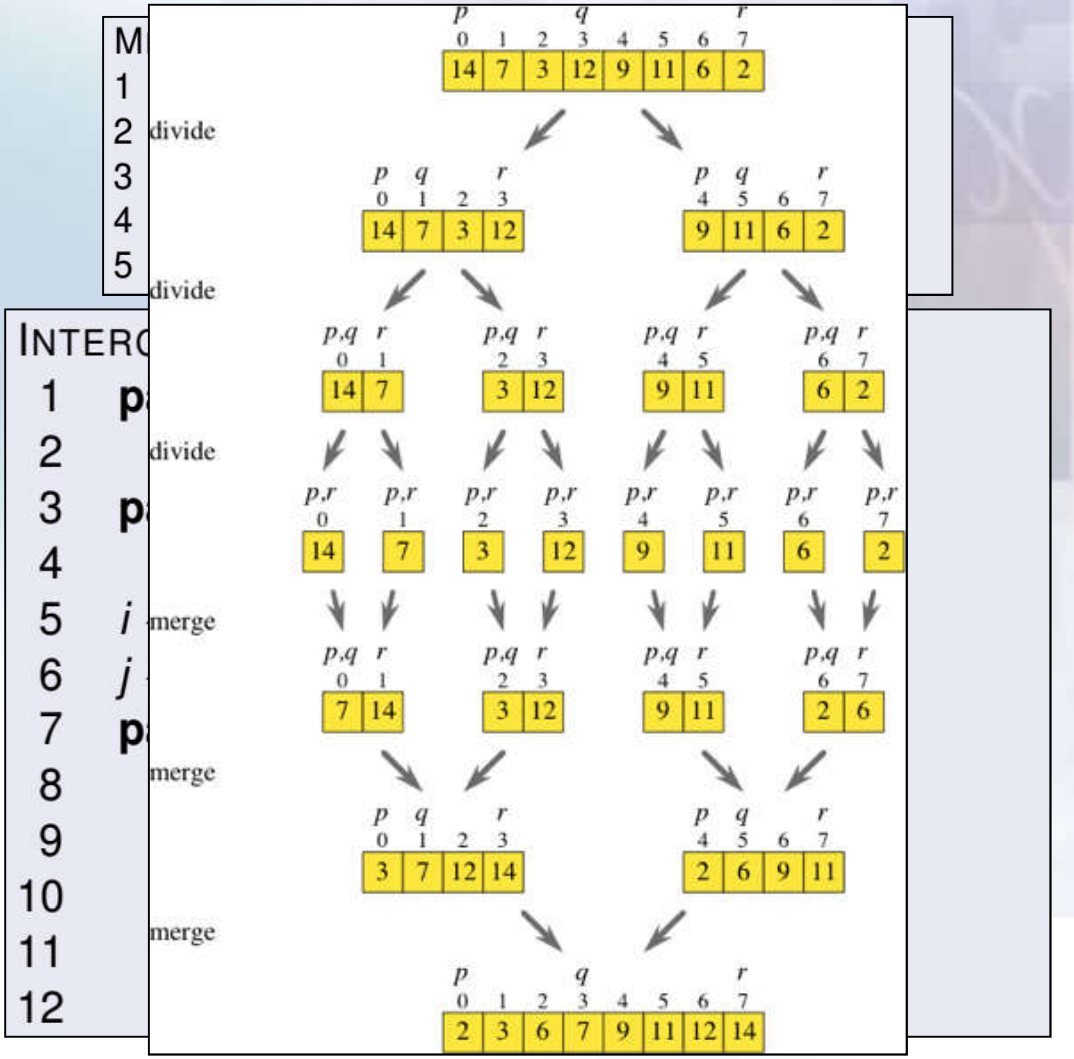


Merge Sort: Algoritmo Intercala() – Sem Sentinela (2)

```
INTERCALA(A, p, q, r)
 1  para i ← p até q faça
 2      B[i] ← A[i]
 3  para j ← q + 1 até r faça
 4      B[r + q + 1 − j] ← A[j]
 5  i ← p
 6  j ← r
 7  para k ← p até r faça
 8      se B[i] ≤ B[j]
 9          então A[k] ← B[i]
10                i ← i + 1
11          senão A[k] ← B[j]
12                j ← j − 1
```

Merge Sort: Ilustração do Algoritmo Exemplo Completo

- A[] = [14, 7, 3, 12, 9, 11, 6, 2]



Merge Sort: Análise da Complexidade Algoritmo Intercala

- **Pior Caso – T(n)**:

Entrada:

	p			q				r
A	22	33	55	77	99	11	44	66	88

Saída:

	p			q				r
A	11	22	33	44	55	66	77	88	99

```
INTERCALA(A, p, q, r)
 1  para i ← p até q faça
 2      B[i] ← A[i]
 3  para j ← q + 1 até r faça
 4      B[r + q + 1 − j] ← A[j]
 5  i ← p
 6  j ← r
 7  para k ← p até r faça
 8      se B[i] ≤ B[j]
 9          então A[k] ← B[i]
10                i ← i + 1
11          senão A[k] ← B[j]
12                j ← j − 1
```

- **Tamanho da Entrada:** n = r – p + 1
- **Complexidade T(n):** O(n) → Linear

Merge Sort: Análise da Complexidade Algoritmo Mergesort (1)

- **Pior Caso – T(n):** Assumir que o Tamanho do Array é Par

```
MERGESORT(A, p, r)
1   se p < r
2       então q ← ⌊(p + r)/2⌋
3             MERGESORT(A, p, q)
4             MERGESORT(A, q + 1, r)
5             INTERCALA(A, p, q, r)
```

Linha	Consumo de Tempo
1	1
2	1
3	T(n/2) $T(\lceil n/2 \rceil)$
4	T(n/2) $T(\lfloor n/2 \rfloor)$
5	n: Complexidade do Intercala
T(n)	**T(n/2) + T(n/2) + O(n) + 2**

Se não fosse par!

Merge Sort: Análise da Complexidade Algoritmo Mergesort (2)

- **Fórmula de Recorrência** (ou seja, uma fórmula definida em termos de si mesma)**:**
  - **T(1) = O(1)** se n=1
  - **T(n) = T(n/2)+ T(n/2)+ O(n)** se n>1

- Em geral, ao utilizar-se do **paradigma de Divisão e Conquista**, adota-se **recursividade** → **Complexidade** T(n) é uma **Fórmula de Recorrência**
  - Precisamos aprender a **resolver recorrência** → Encontrar uma **"Fórmula Fechada"** para T(n)

Merge Sort: Análise da Complexidade Algoritmo Mergesort (3)

- **Fórmula de Recorrência:**
  - **T(1) = c** se n=1
  - **T(n) = 2.T(n/2) + c.n** se n>1

  onde c é um a constante para o tempo exigido em resolver problemas de tamanho 1, bem como o tempo por elemento do (sub)array para as etapas de dividir e combinar

- Pelo teorema mestre, $n^{\log_a b} = n^{\log_2 2} = n$ → Mesma Complexidade de f(n) = c.n
  - Caso 2: **T(n) = n . lgn**

Outros Métodos: Merge Sort (3)

```
/*OrdenaMergeSort(): Função que ordena um vetor utilizando o MergeSort. */
void OrdenaMergeSort(int Vet[Max])
{
  //Variaveis Locais
  int tam=1;                    //Fusao de Arquivos de Tamanho 1
  int L1, R1;                   //Limite Inferior e Superior do Primeiro Arquivo
  int L2, R2;                   //Limite Inferior e Superior do Segundo Arquivo
  int i, j, k;                  //Indices Auxiliares
  int AuxFun[Max];    //Vetor Auxiliar da Funcao

  while(tam < Max)
  {
     L1 = 0;
    k = 0;
    while(L1+tam < Max)
    {
                  L2 = L1+tam;
      R1 = L2-1;
      if((L2+tam-1) < Max)
                  R2 = L2+tam-1;
      else
                  R2 = Max-1;

      //Trabalhando com os dois sub-Arquivos
      for(i=L1, j=L2; (i<=R1)&&(j<=R2); k++)
      {
                              //Insere os menores no vetor auxiliar
        if(Vet[i] <= Vet[j])
        {
          AuxFun[k] = Vet[i];
          i++;
        }
        else
        {
                AuxFun[k] = Vet[j];
          j++;
        }
      }
```

Entender o Funcionamento é Mais Importante que Decorar o Algoritmo!!!

Outros Métodos: Merge Sort (4)

```
    //Neste ponto um arquivo foi esgotado
    //Insere os elementos restantes no outro
    while(i <= R1)
    {
            AuxFun[k] = Vet[i];
      i++;
      k++;
    }
    while(j <= R2)
    {
            AuxFun[k] = Vet[j];
      j++;
      k++;
    }
    //Avanca L1 para o proximo par de arquivos
    L1 = R2+1;
  }
  //Copia todo o arquivo restante
  for(i=L1; k < Max; i++)
  {
            AuxFun[k] = Vet[i];
    k++;
  }
  //Copia AuxFun em Aux e ajusta tam
  for(i=0; i < Max; i++) Vet[i] = AuxFun[i];

  tam = tam*2;                    //Dobra o tamanho do segmento
 }
}
```

Entender o Funcionamento é Mais Importante que Decorar o Algoritmo!!!

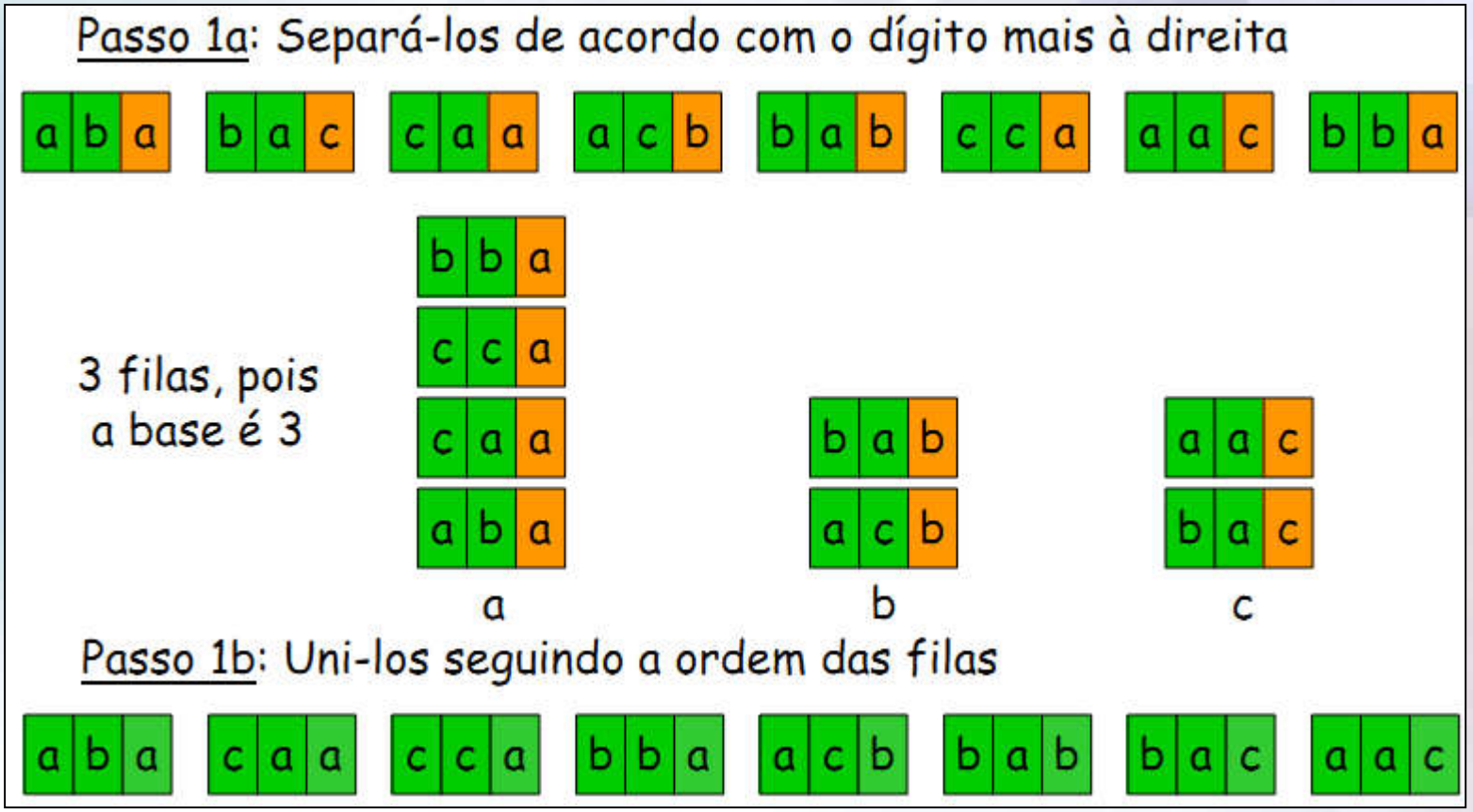
Outros Algoritmos de Ordenação: Radix Sort

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista

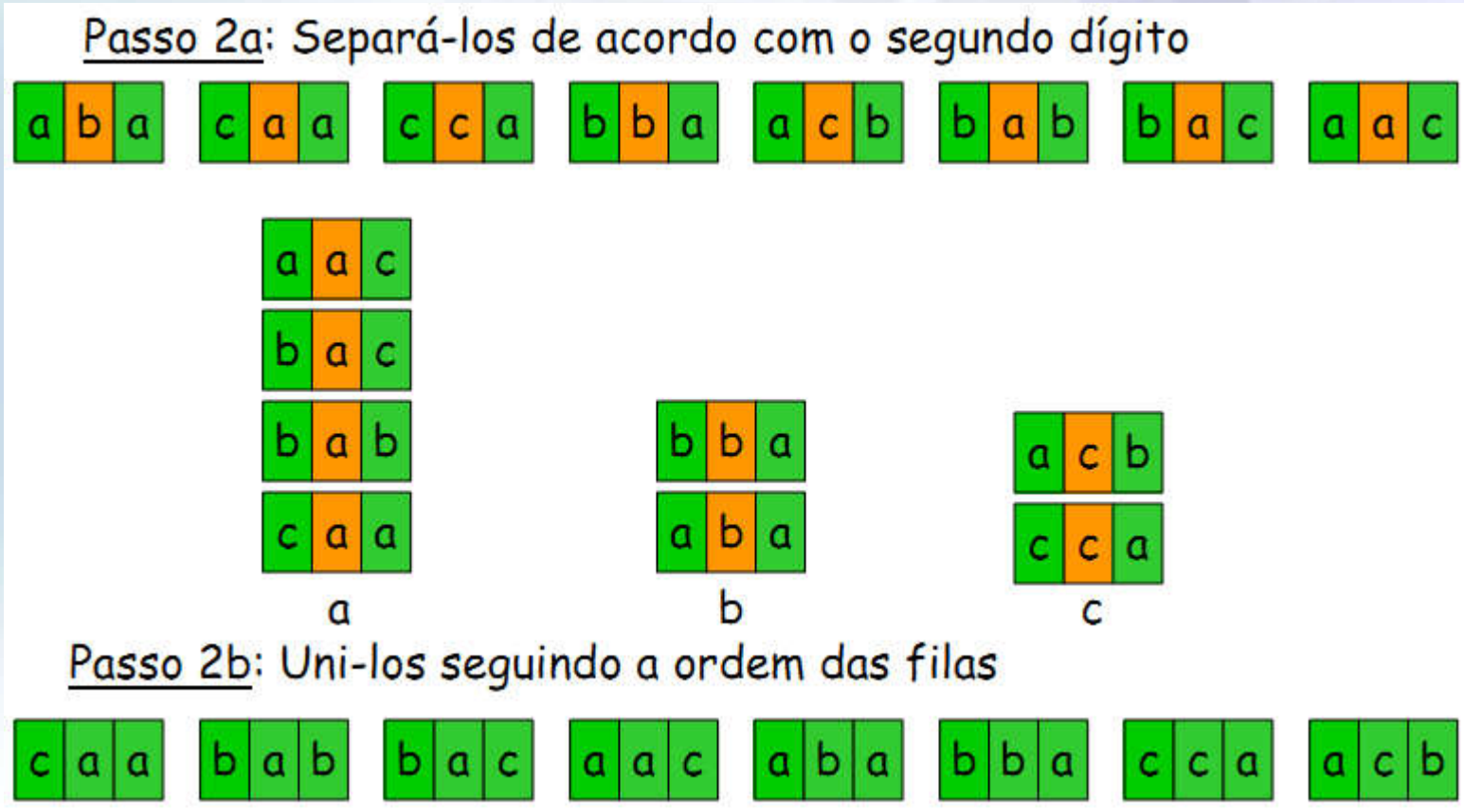
Outros Métodos: Radix Sort (1)

- Em determinadas condições especiais, é possível ordenar em tempo linear
- Por exemplo, isto ocorre quando:
  - os valores têm um comprimento limitado
  - a ordenação baseia-se em cálculos com esses valores
- Como funciona o RadixSort():
  - Os valores de entrada, escritos em alguma base numérica, têm exatamente d dígitos
  - A ordenação é realizada em d passos: um dígito por vez, começando a partir dos menos significativos

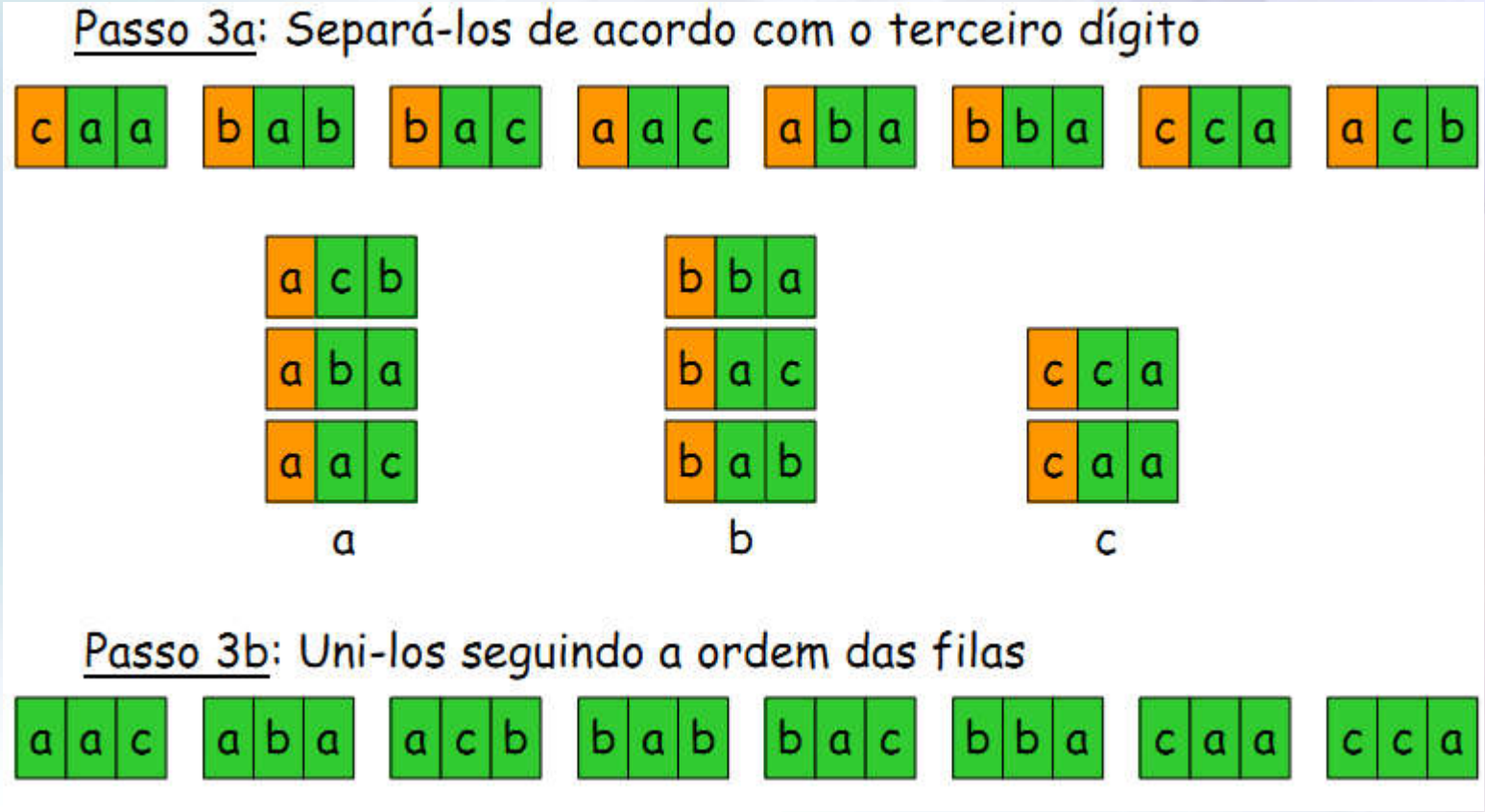
Outros Métodos: Radix Sort – Exemplo com 3 Dígitos (2)



Outros Métodos: Radix Sort – Exemplo com 3 Dígitos (3)



Outros Métodos: Radix Sort – Exemplo com 3 Dígitos (2)



Outros Exemplo de Radix Sort (1)

12	34	42	32	44	41	34	11	32	23	87	50	77	58	08

Para inteiros, o número de compartimentos é 10, de 0 a 9. A primeira passada distribui as chaves nos compartimentos com base no dígito menos significativo.

Primeira Interação

0	1	2	3	4	5	6	7	8
50	41	12	23	34			87	58
	11	42		44			77	08
		32		34				
		32						

Outros Exemplo de Radix Sort (2)

50	41	11	12	42	32	32	23	34	44	34	87	77	58	08

A distribuição agora é feita com base no segundo dígito menos significativo

Segunda Interação

0	1	2	3	4	5	6	7	8
08	11	23	32	41	50		77	87
	12		32	42	58			
			34	44				
			34					

Resultado após a recomposição dos conjuntos

08	11	12	23	32	32	34	34	41	42	44	50	58	77	87

Outros Métodos: Radix Sort (2)

```
/*OrdenaRadixSort(): Funcao que ordena o vetor pelo método radix sort. */
void OrdenaRadixSort(int Vet[])
{
    int i, b[Max], m=0, exp=1;
    for(i=0;i<Max;i++)
    {
                if(Vet[i]>m)
                    m=Vet[i];
    }

    while(m/exp>0)
    {
                int bucket[10]={0};
                for(i=0;i<Max;i++)
                            bucket[Vet[i]/exp%10]++;
                for(i=1;i<10;i++)
                            bucket[i]+=bucket[i-1];
                for(i=Max-1;i>=0;i--)
                            b[--bucket[Vet[i]/exp%10]]=Vet[i];
                for(i=0;i<Max;i++)
                            Vet[i]=b[i];
                exp*=10;
    }
}
```

Entender o Funcionamento é Mais Importante que Decorar o Algoritmo!!!

Ordenação por Radix Sort: Análise da Complexidade

- Principais pontos a se destacar:
  - Algoritmo de ordenação por distribuição que ordena com base nos dígitos de um número → prioriza os dígitos menos significativos

- Análise da Complexidade do Algoritmo:
  - Para todos os casos (Melhor, Pior e Caso Médio):
    - T(n) = O(n.d) → T(n) = O(n), pois d é uma constante
    - Justificativa: n é o número de elementos e d é a quantidade de dígitos

PANC: Projeto e Análise de Algoritmos

Aula 12: Algoritmos de Ordenação III - Troca:

Select Sort e Heap Sort

e outros Algoritmos:

Merge Sort e RadixSort

Breno Lisi Romano

Dúvidas???

http://sites.google.com/site/blromano

Instituto Federal de São Paulo – IFSP São João da Boa Vista
Bacharelado em Ciência da Computação – 3º Semestre

INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus São João da Boa Vista