Estrutura de dados
Métodos de ordenação

Andrei Hirata
Guilherme Diniz
Thiago Nakao

Câmpus Campo Mourão

UTFPR
UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS CAMPO MOURÃO

MINISTÉRIO DA EDUCAÇÃO

BRASIL
GOVERNO FEDERAL

Sumário

1. Introdução
2. Merge Sort
3. Quick Sort
4. Counting Sort
5. Bucket Sort
6. Radix Sort

Métodos de Ordenação

- Método de ordenação é um algoritmo que coloca os elementos de uma dada sequência em uma certa ordem.

- O objetivo é facilitar a recuperação dos dados de uma lista

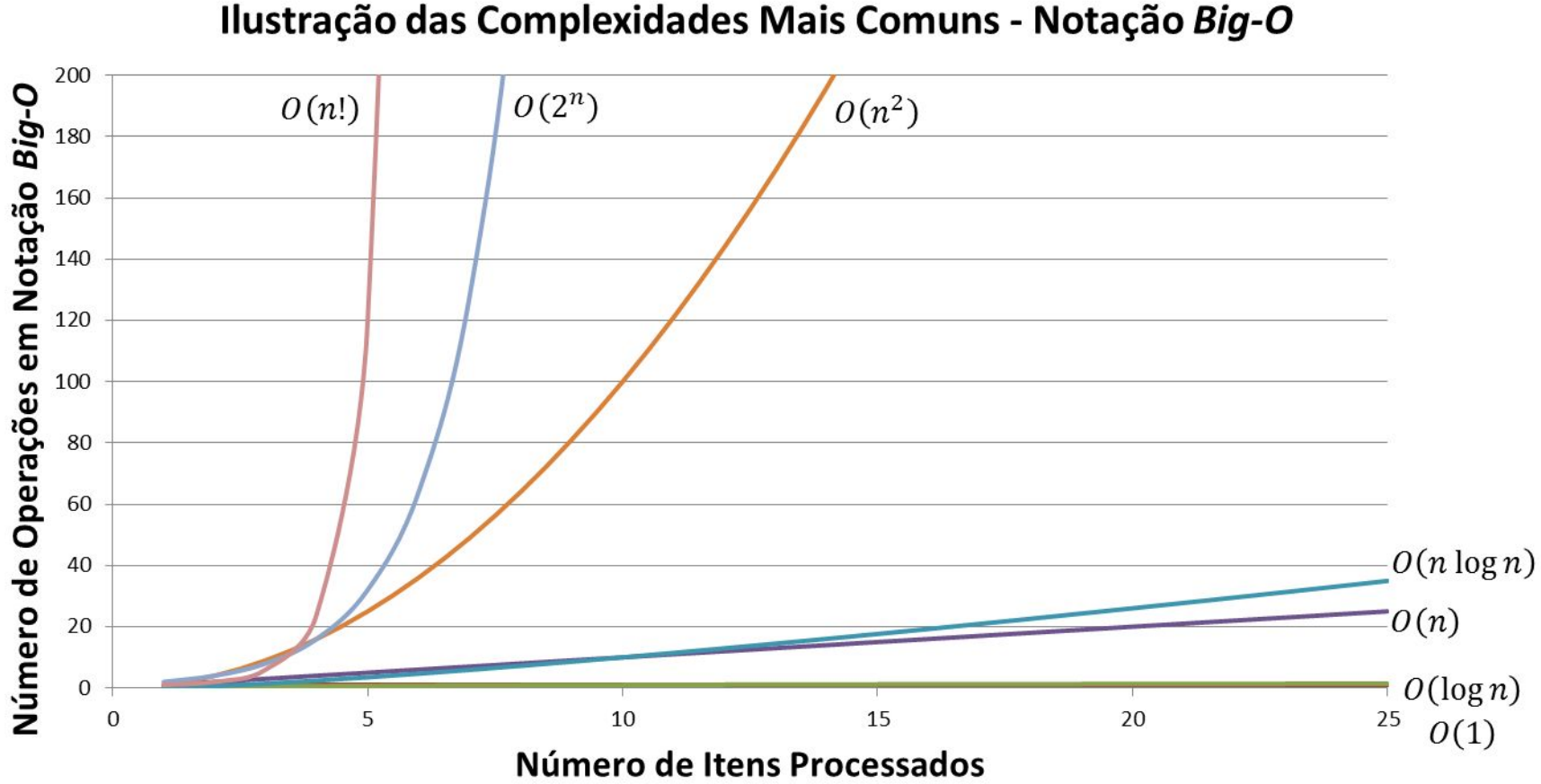
Métodos de Ordenação

Avaliando um algoritmo

- *Big-O notation* - complexidade
  - Definir um limite superior a uma função de acordo com a taxa de crescimento da função
  - Complexidade pior caso
  - Complexidade caso médio
  - Complexidade melhor caso

- Uso de memória

Métodos de Ordenação



Métodos de Ordenação

Métodos simples

- Inserction sort
- Selection sort
- Buble sort
- Bogo sort
- Comb sort

Métodos sofisticados

- Merge sort
- Quick sort
- Counting sort
- Radix sort
- Bucket sort
- Heap sort
- Gnome sort
- Tim Sort

Métodos de Ordenação

Métodos simples

- ~~Inserction sort~~
- ~~Selection sort~~
- ~~Buble sort~~
- ~~Bogo sort~~
- ~~Comb sort~~

Métodos sofisticados

- Merge sort
- Quick sort
- Counting sort
- Radix sort
- Bucket sort
- ~~Heap sort~~
- ~~Gnome sort~~
- ~~Tim Sort~~

Merge Sort

- Desenvolvido por John von Neumann.
- É considerado um dos primeiros métodos de ordenação inventados.
- A propriedade mais atrativa deste método de ordenação é que ele é capaz de ordenar um vetor qualquer de n de elementos em um tempo proporcional a O(N log N).

Merge Sort

- Espaço extra de memória proporcional a n.

- Assim, Merge Sort é ideal para aplicações que precisam de ordenação eficiente, que não toleram desempenho ruim no pior caso e que possuam espaço de memória extra disponível.

Merge Sort

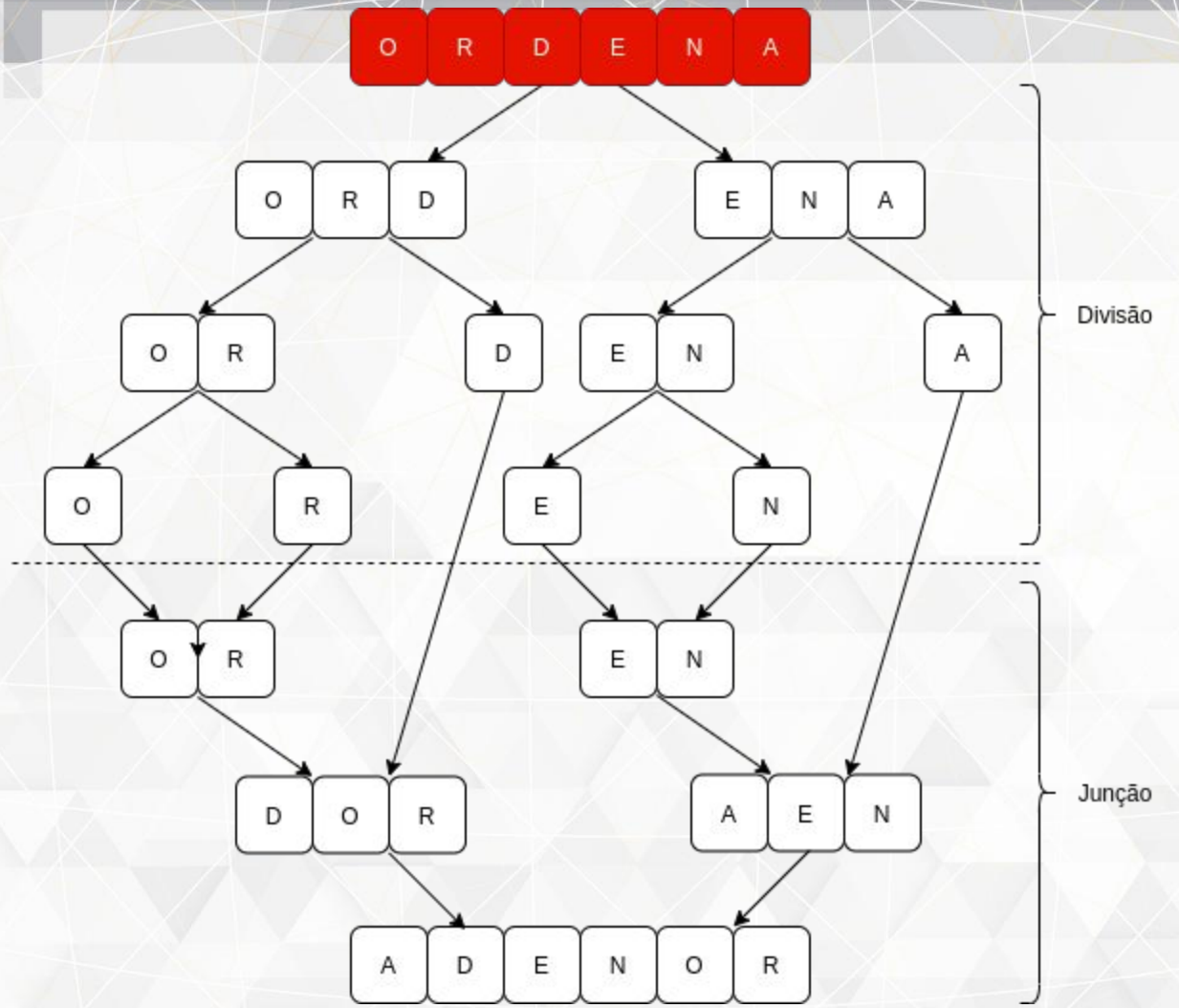
classe	Algoritmo de ordenação
estrutura de dados	Array, Listas ligadas
complexidade pior caso	$\Theta(n \log n)$
complexidade caso médio	$\Theta(n \log n)$
complexidade melhor caso	$\Theta(n \log n)$ típico, $\Theta(n)$ variante natural

Merge Sort

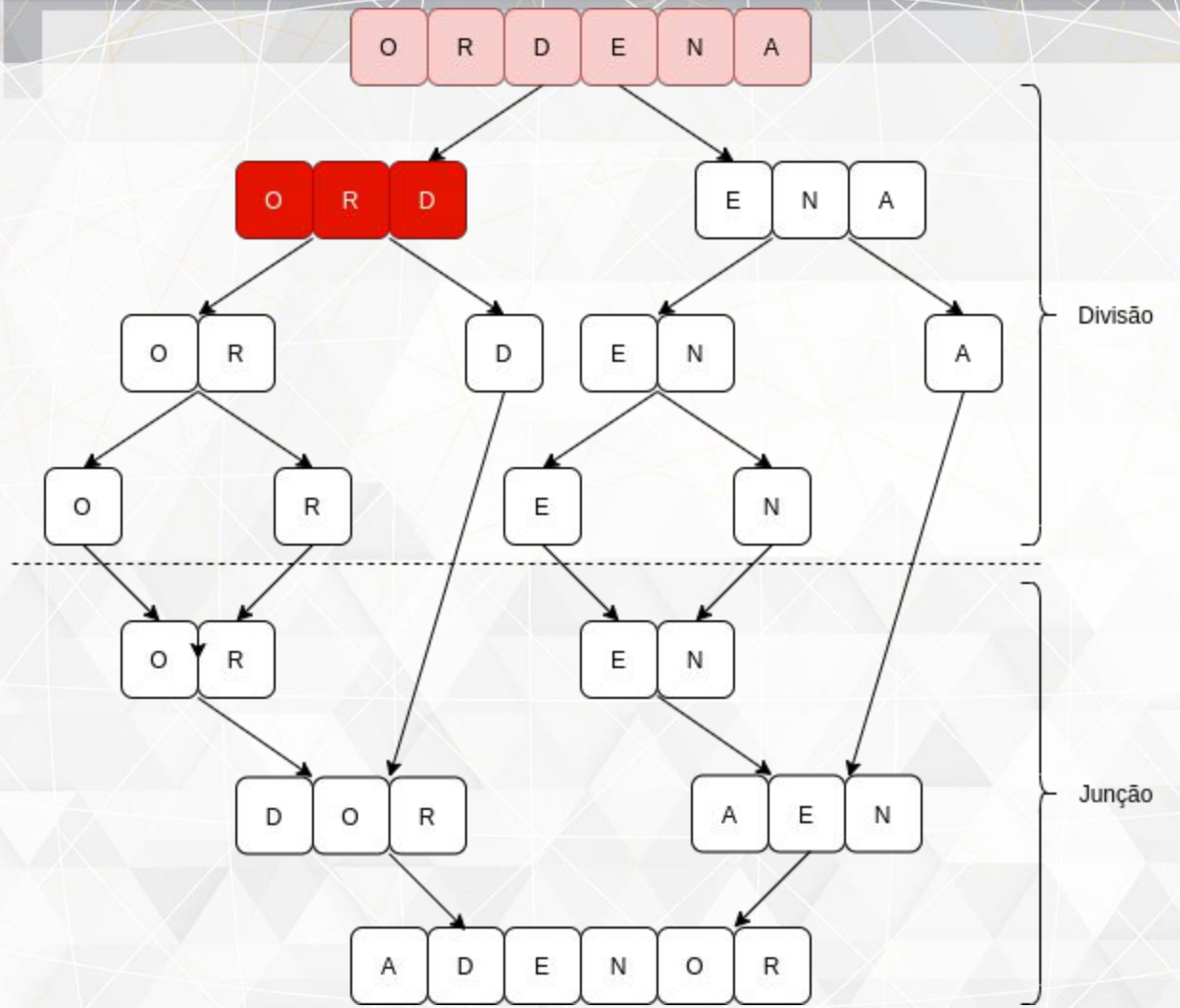
- O processo-chave do Merge Sort consiste em dividir o vetor original em subvetores cada vez menores até que se tenha pequenos subvetores com um elemento apenas.

- A partir daí, cada par de subvetores é fundido (merged) de forma intercalada até se obter um único vetor ordenado com todos os elementos.

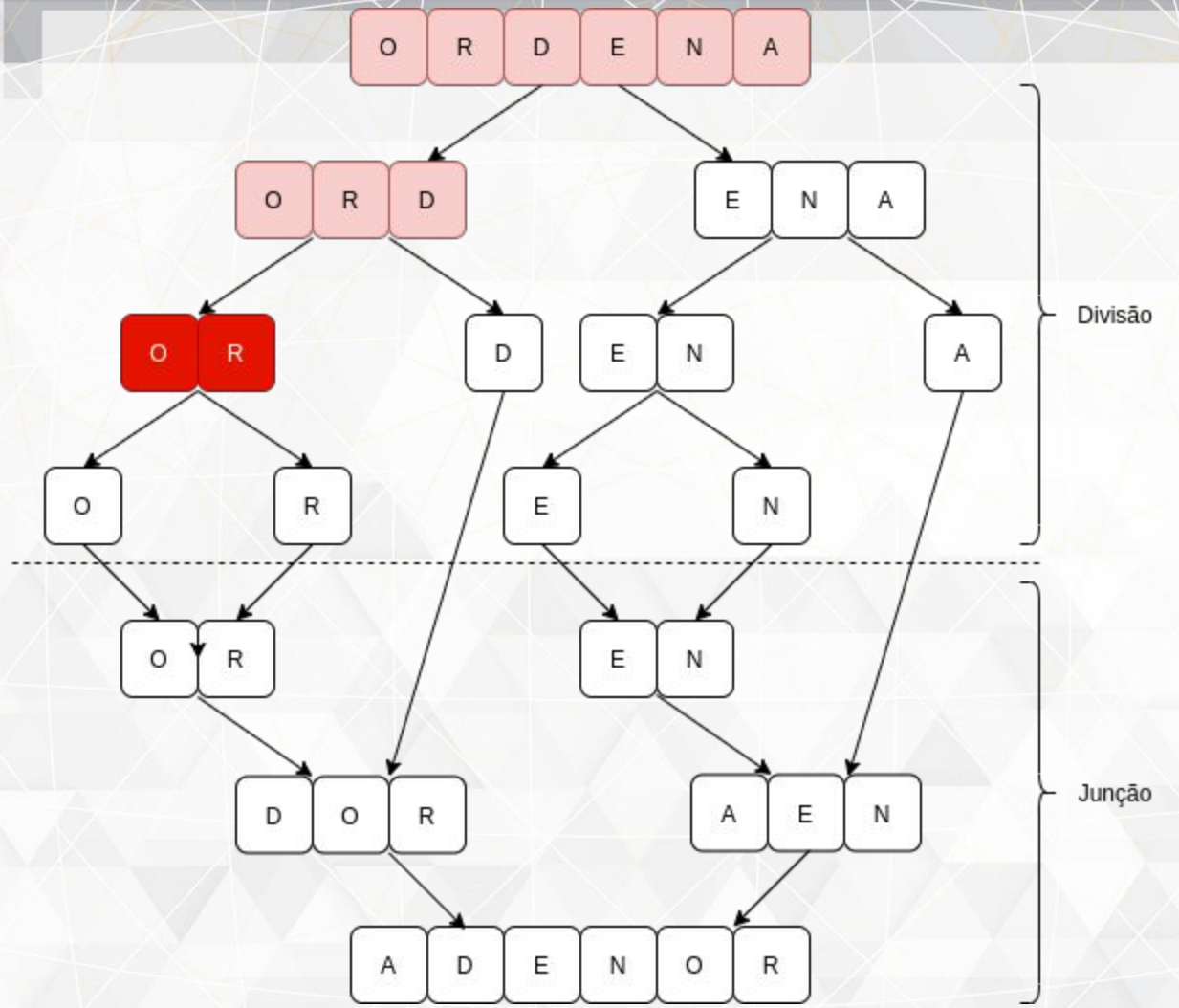
Merge Sort - Divisão esquerdo



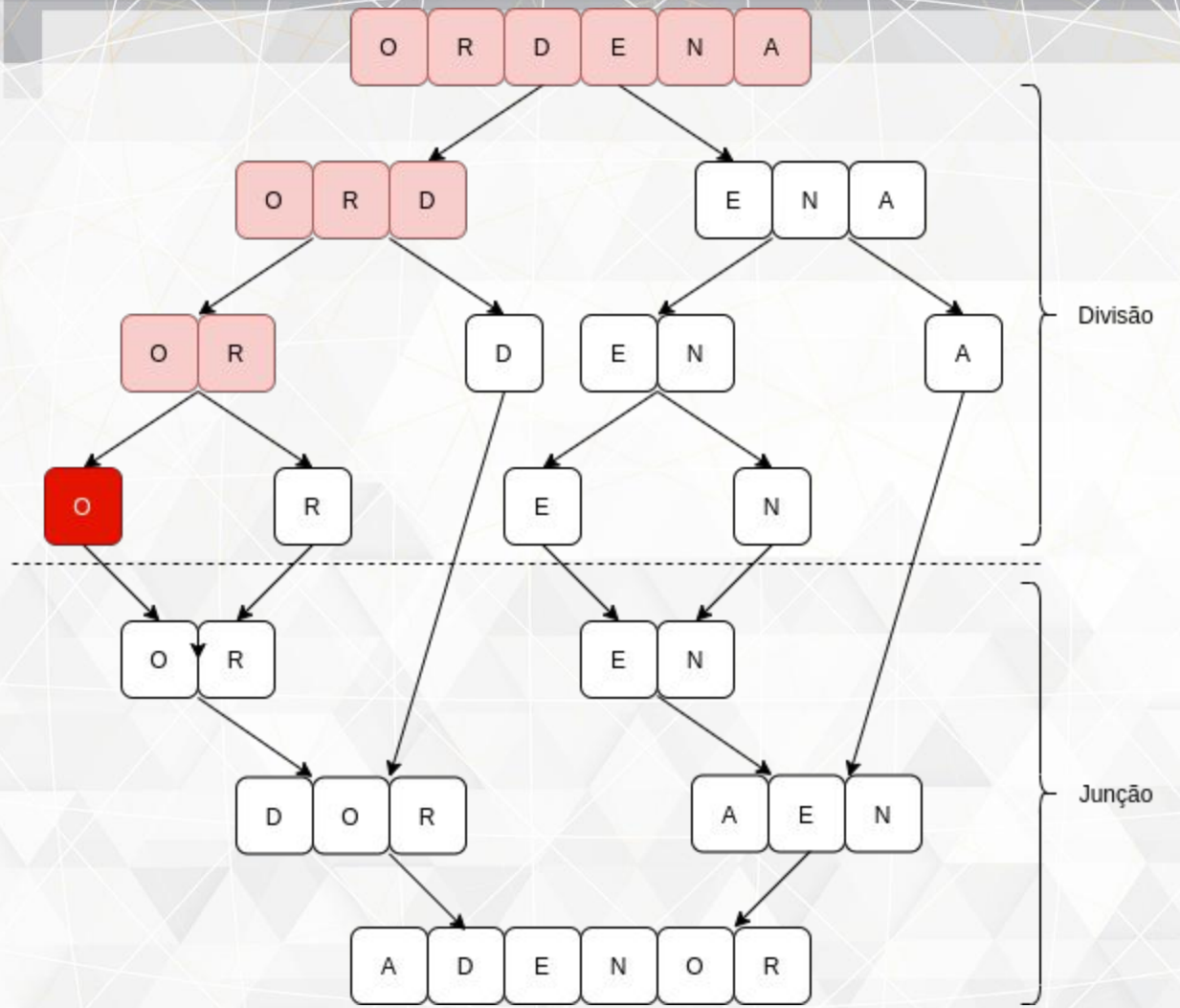
Merge Sort - Divisão esquerdo



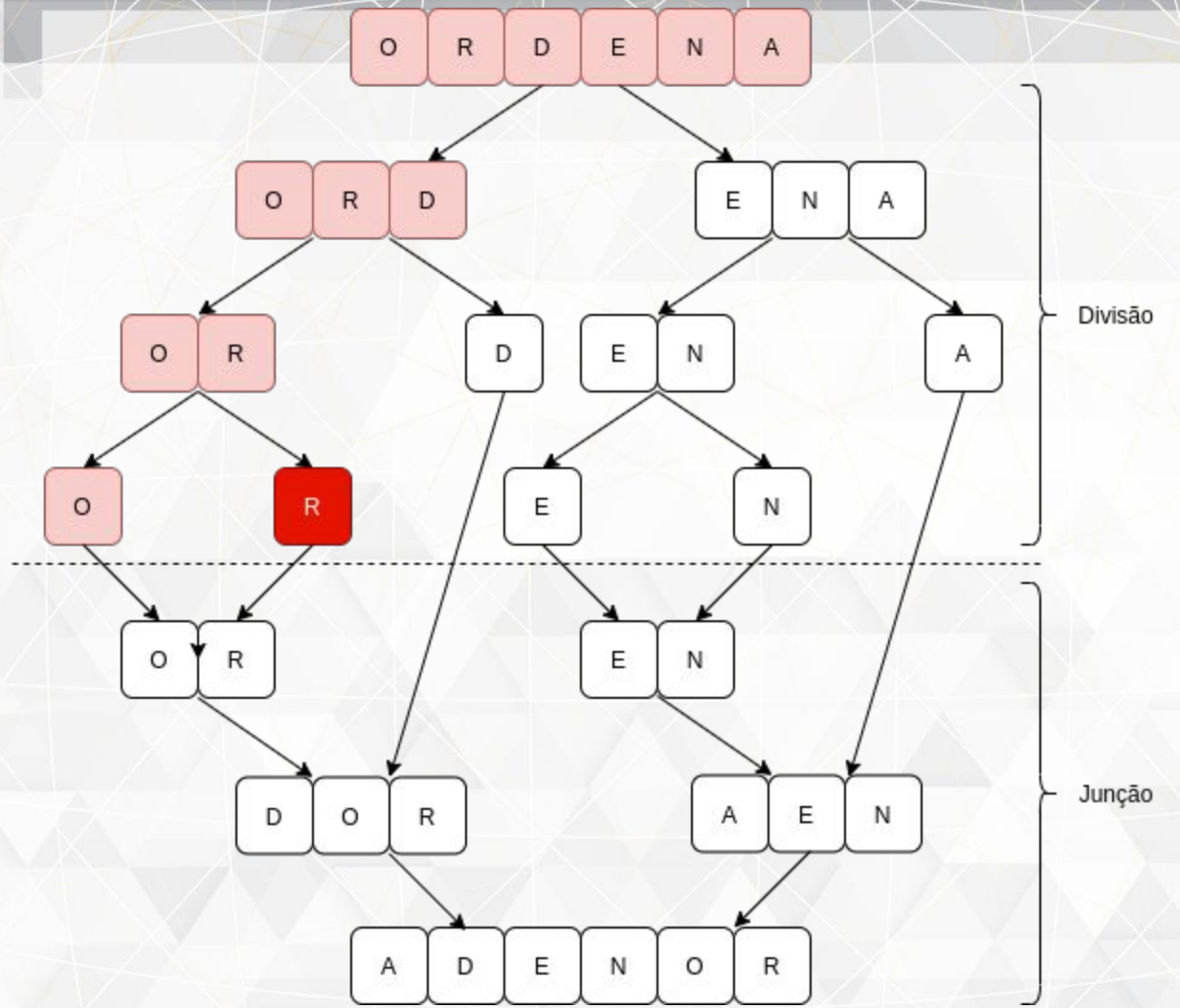
Merge Sort - Divisão esquerdo



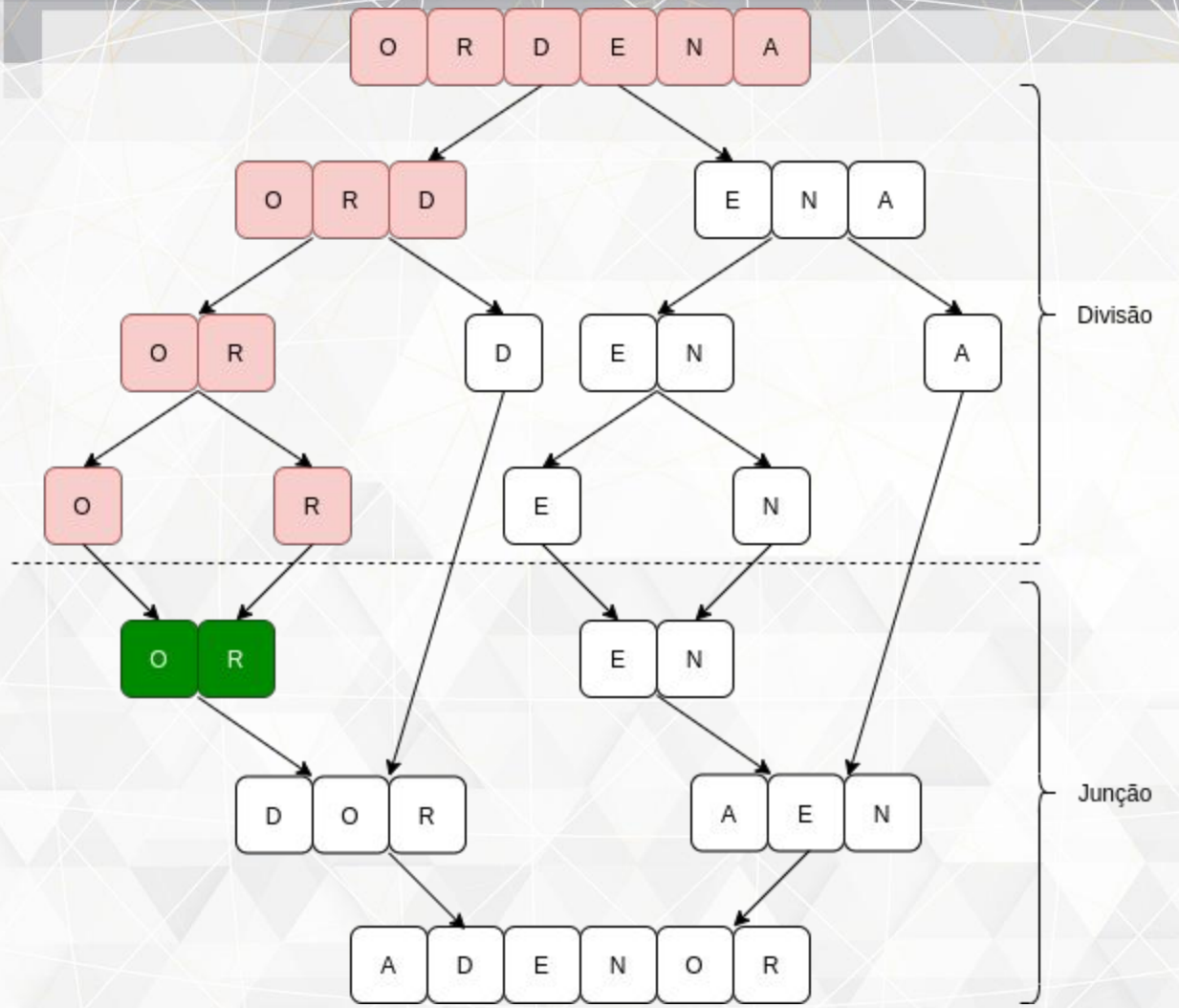
Merge Sort - Divisão esquerdo



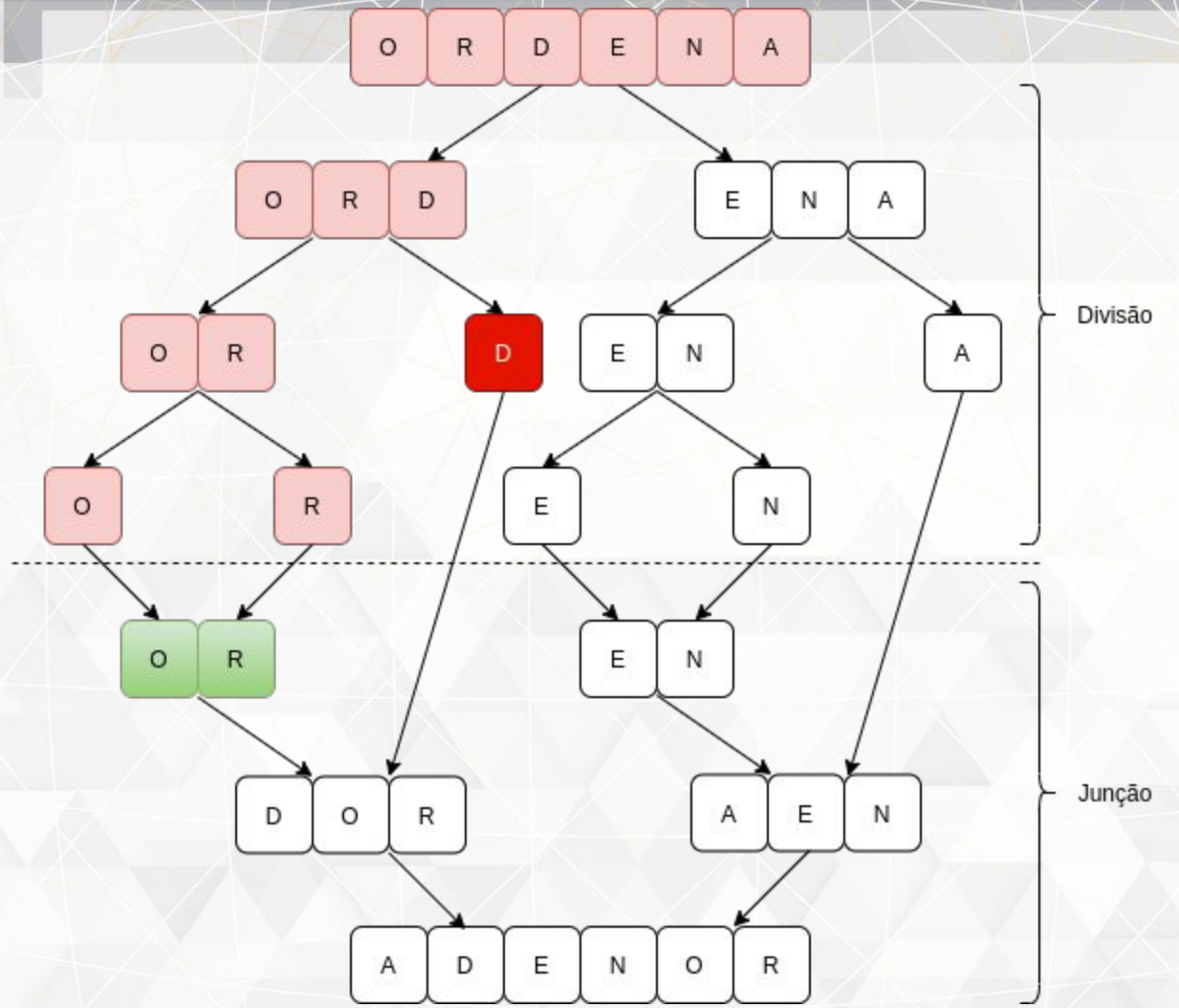
Merge Sort - Divisão esquerdo



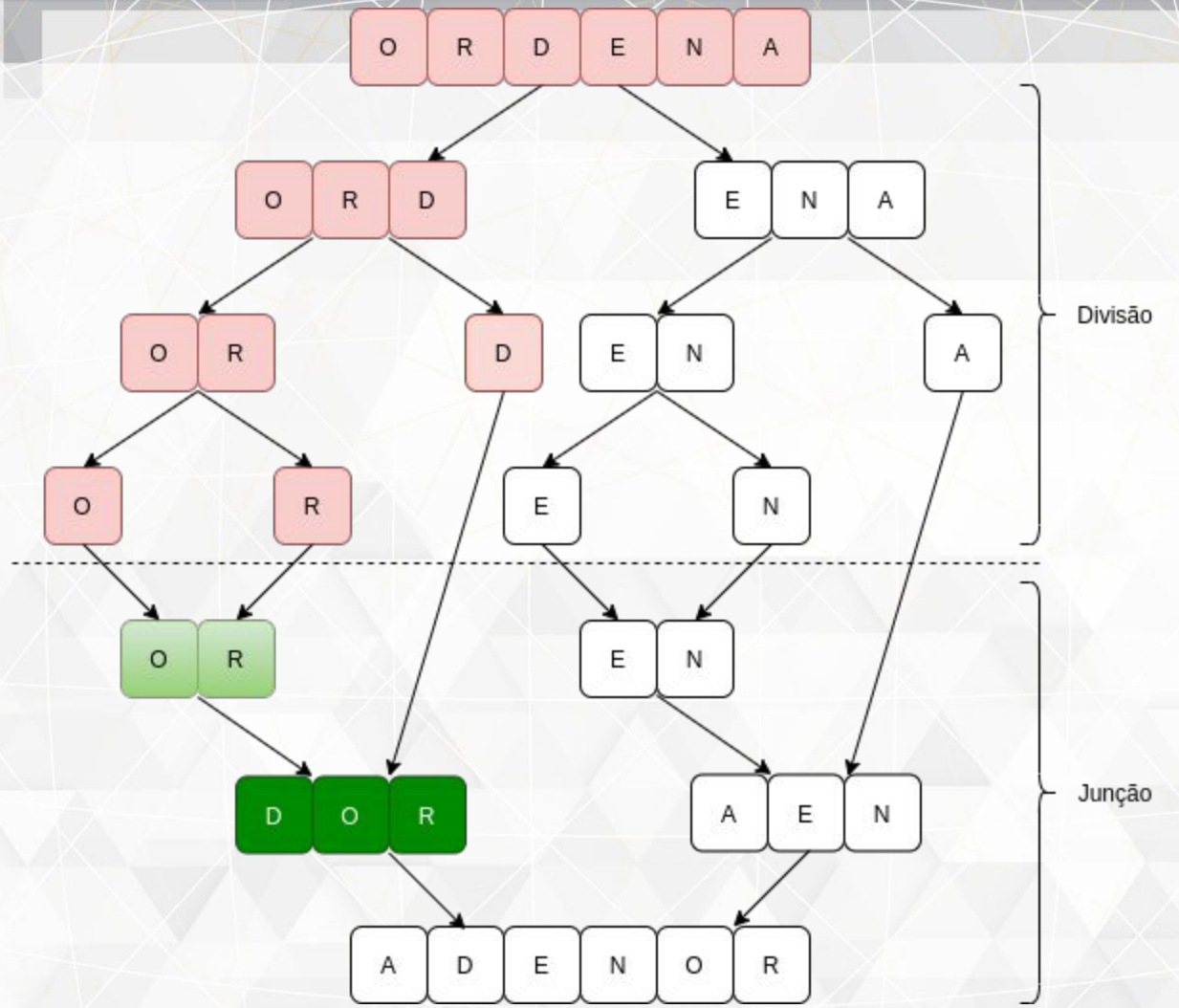
Merge Sort - Junção



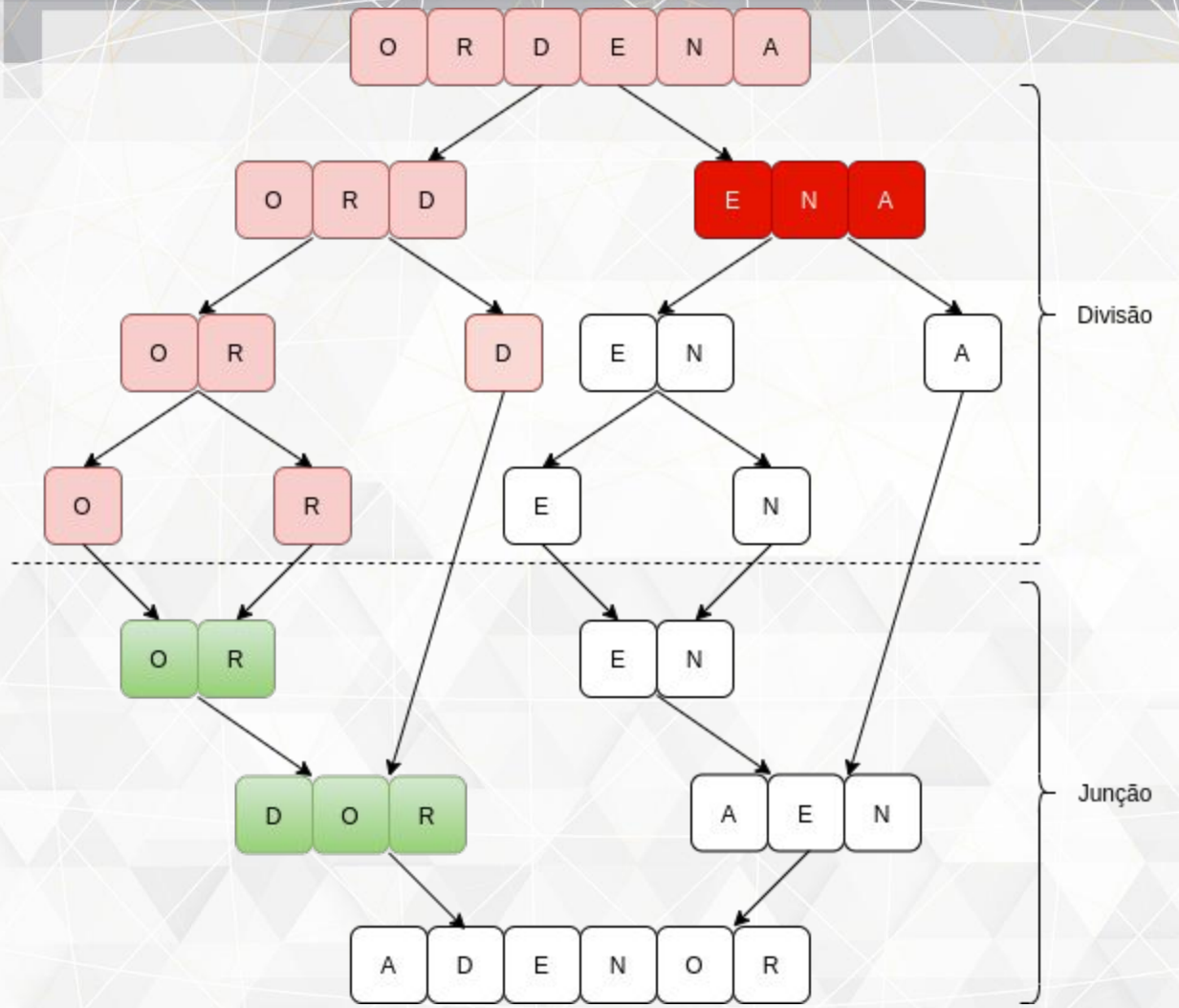
Merge Sort - Divisão



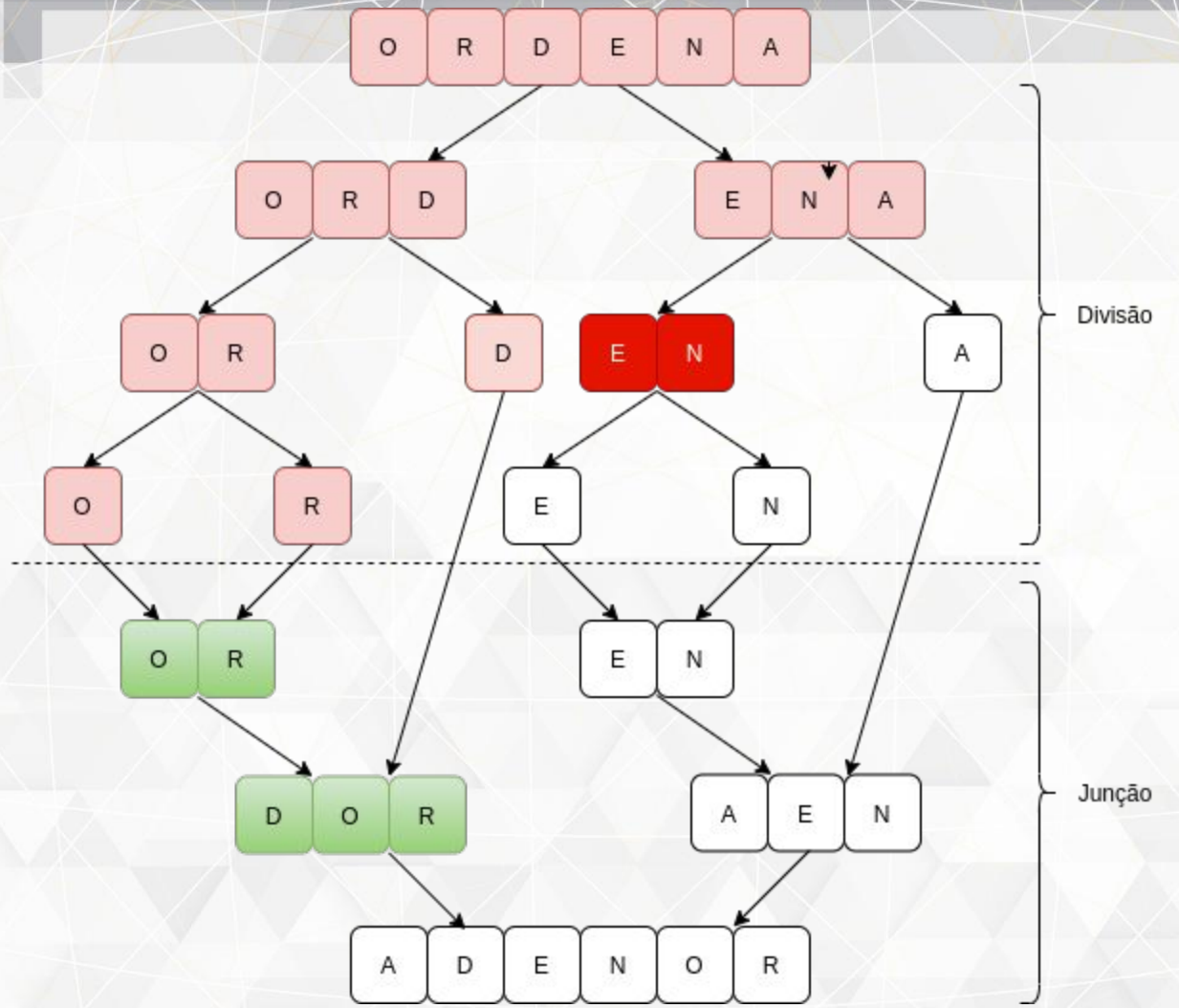
Merge Sort - Junção



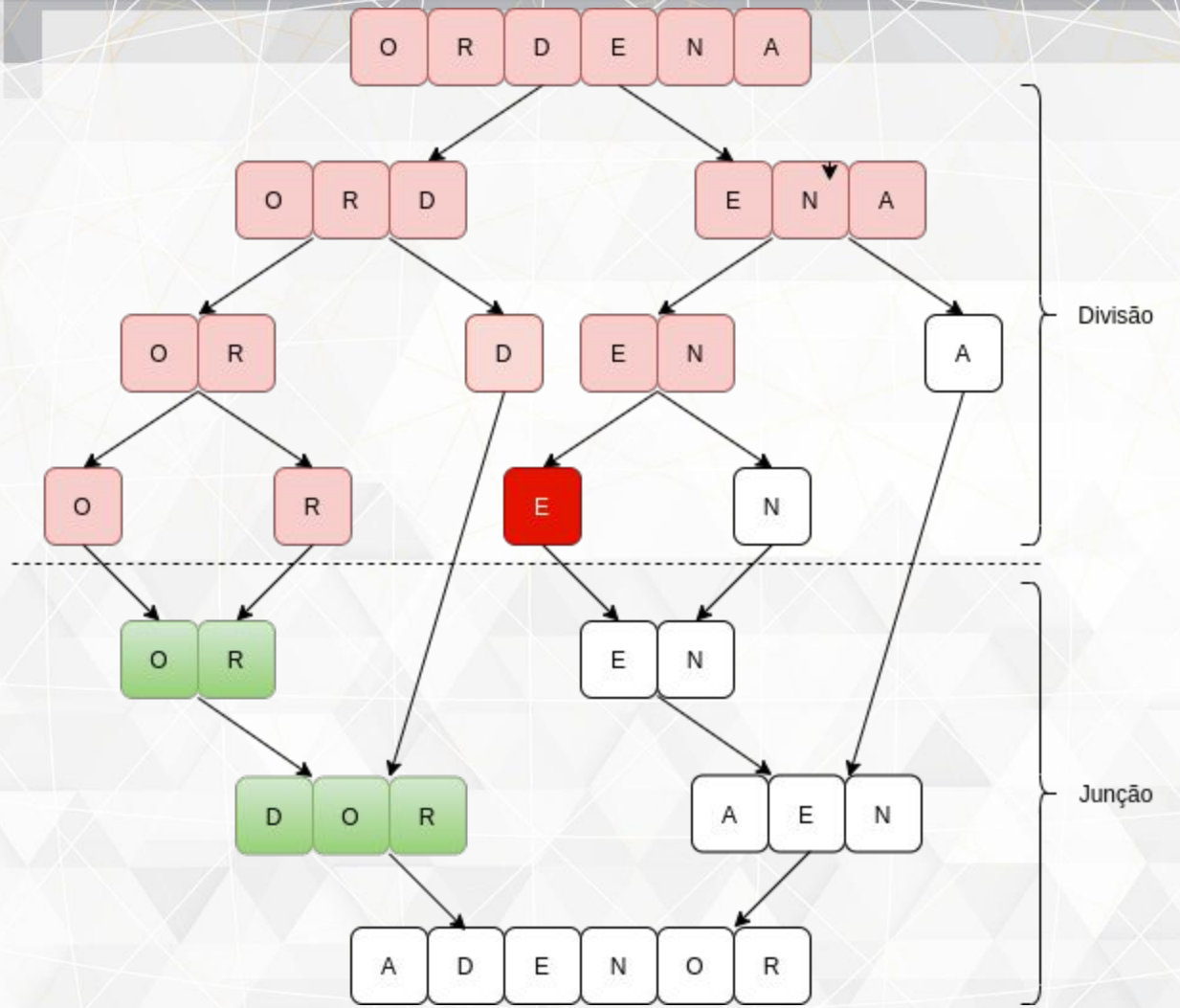
Merge Sort - Divisão



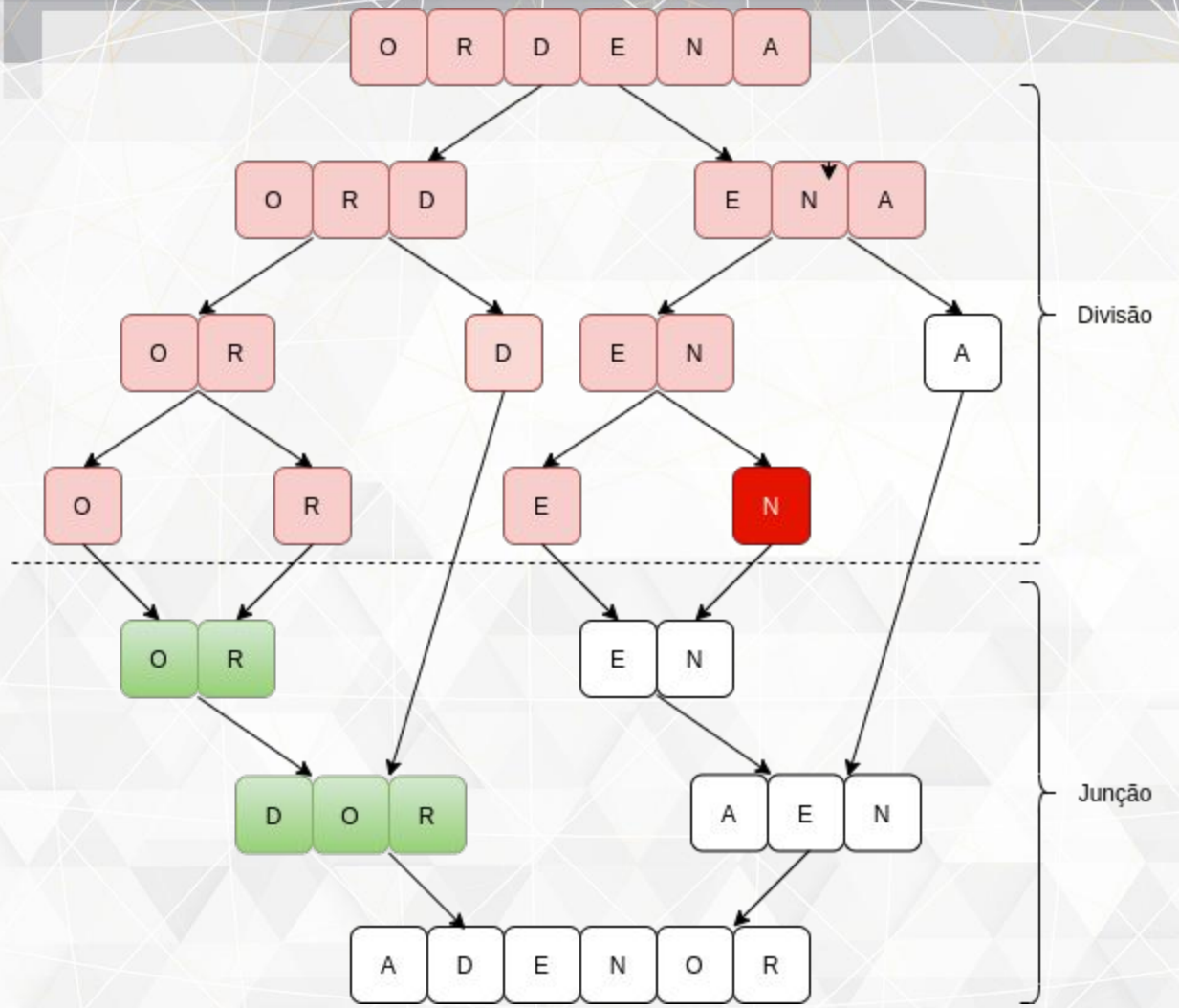
Merge Sort - Divisão



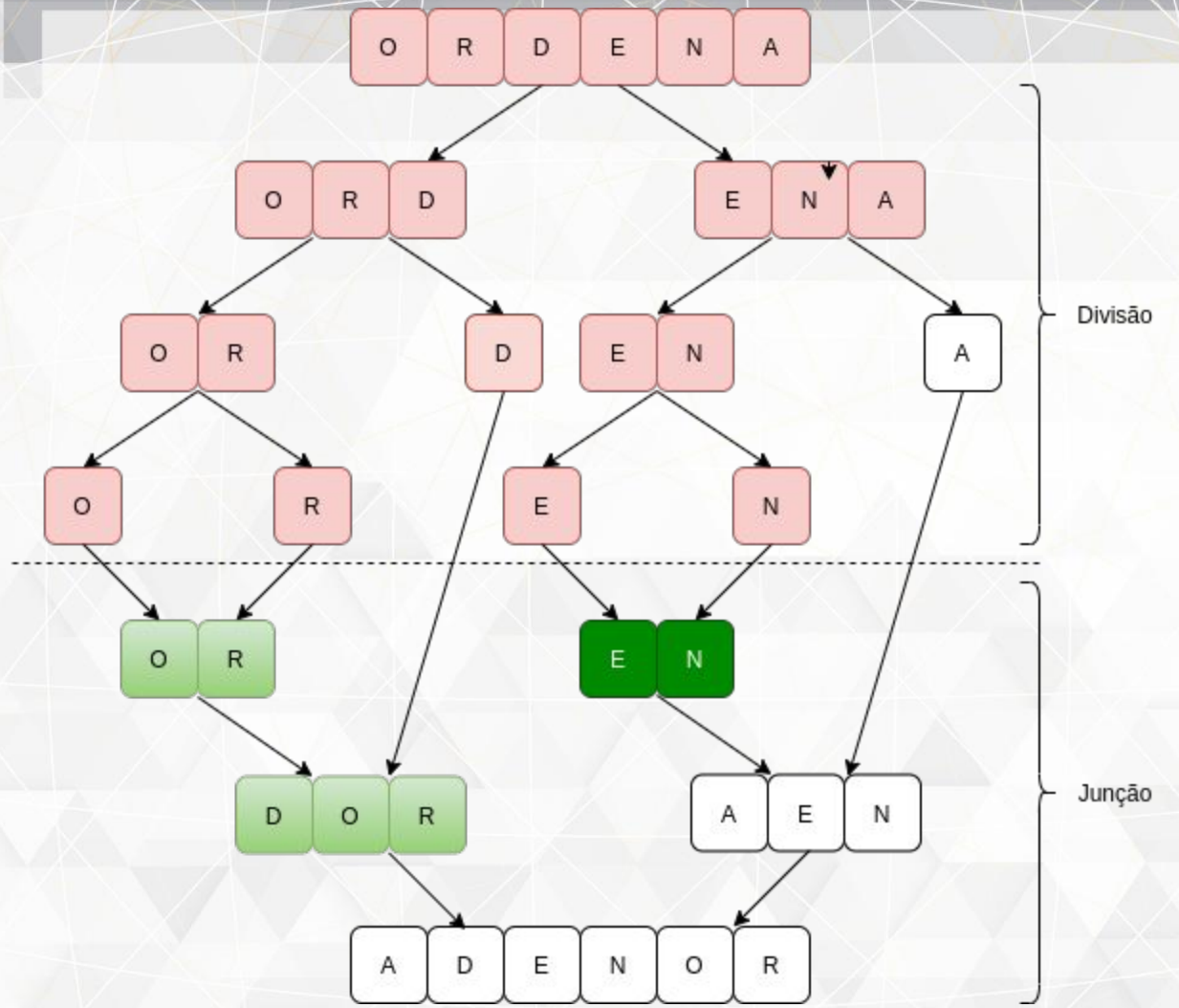
Merge Sort - Divisão



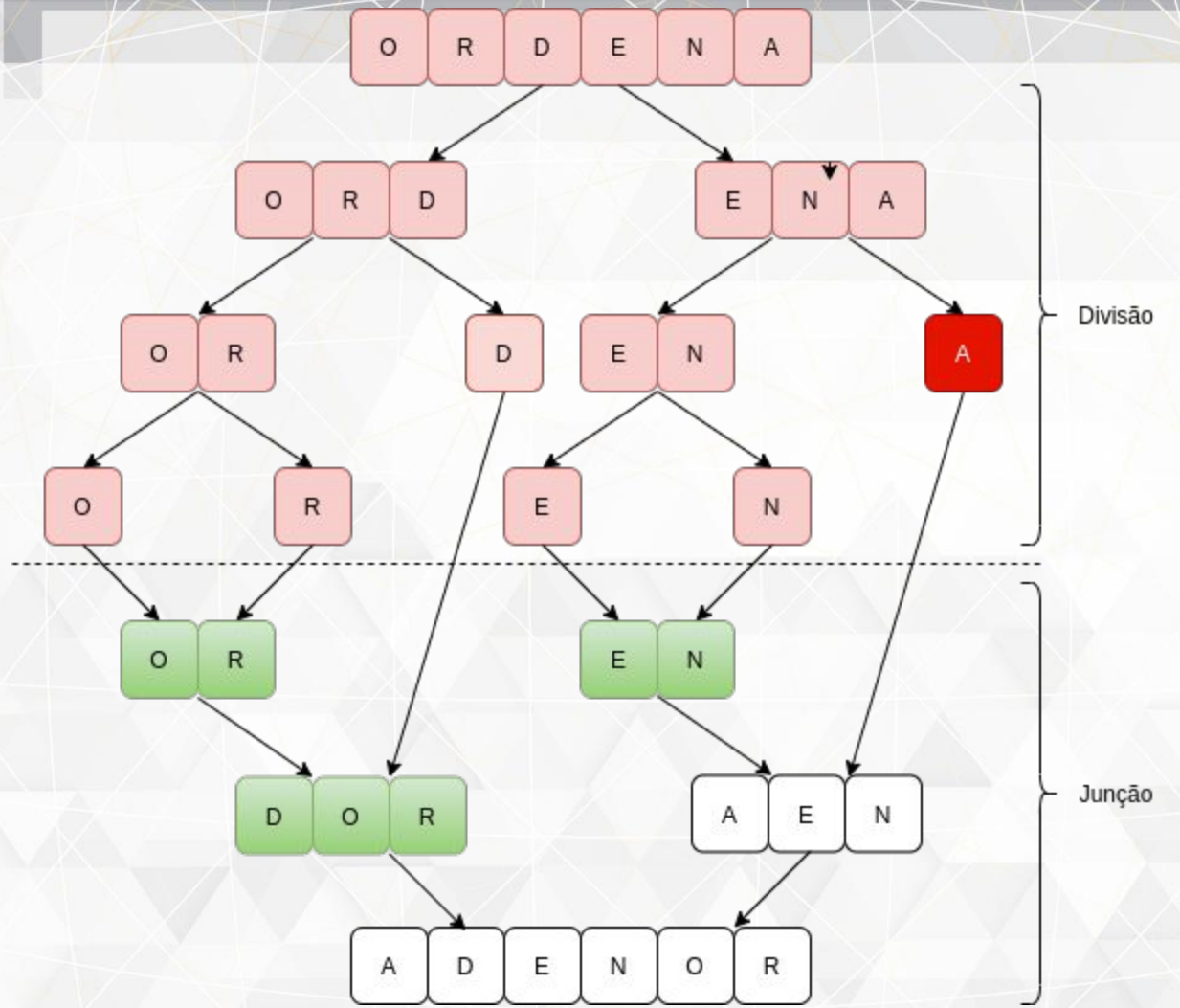
Merge Sort - Divisão



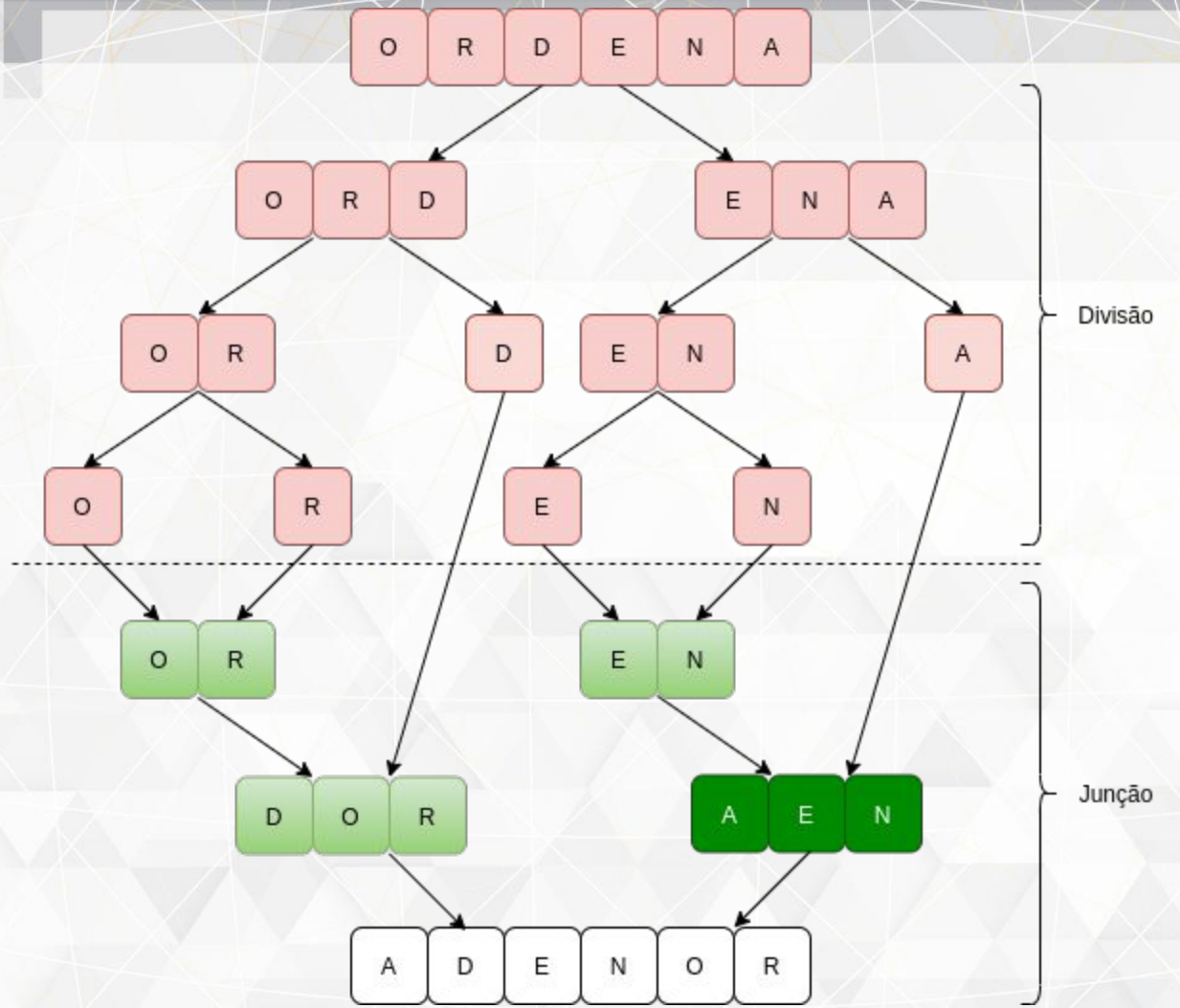
Merge Sort - Junção



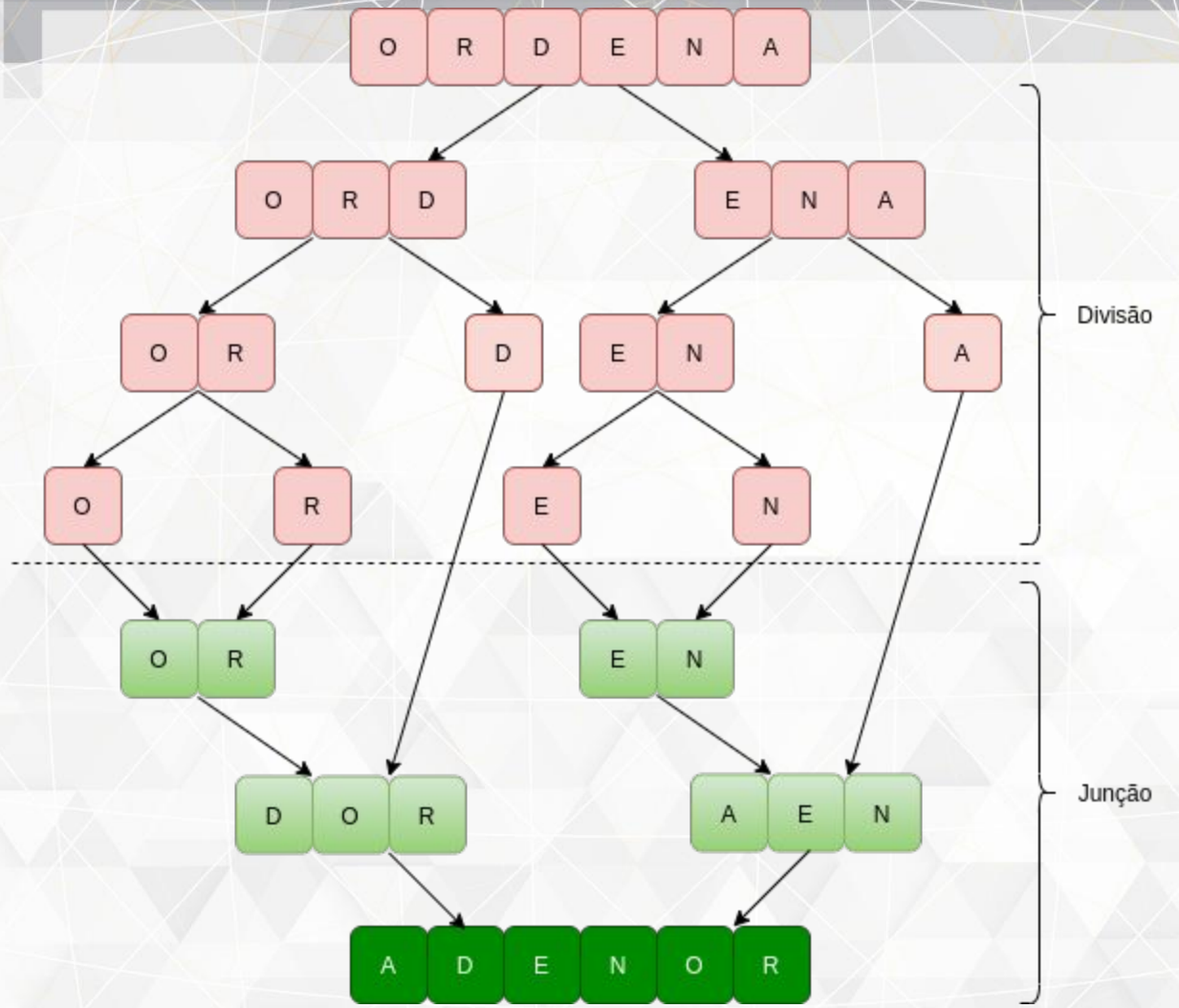
Merge Sort - Divisão



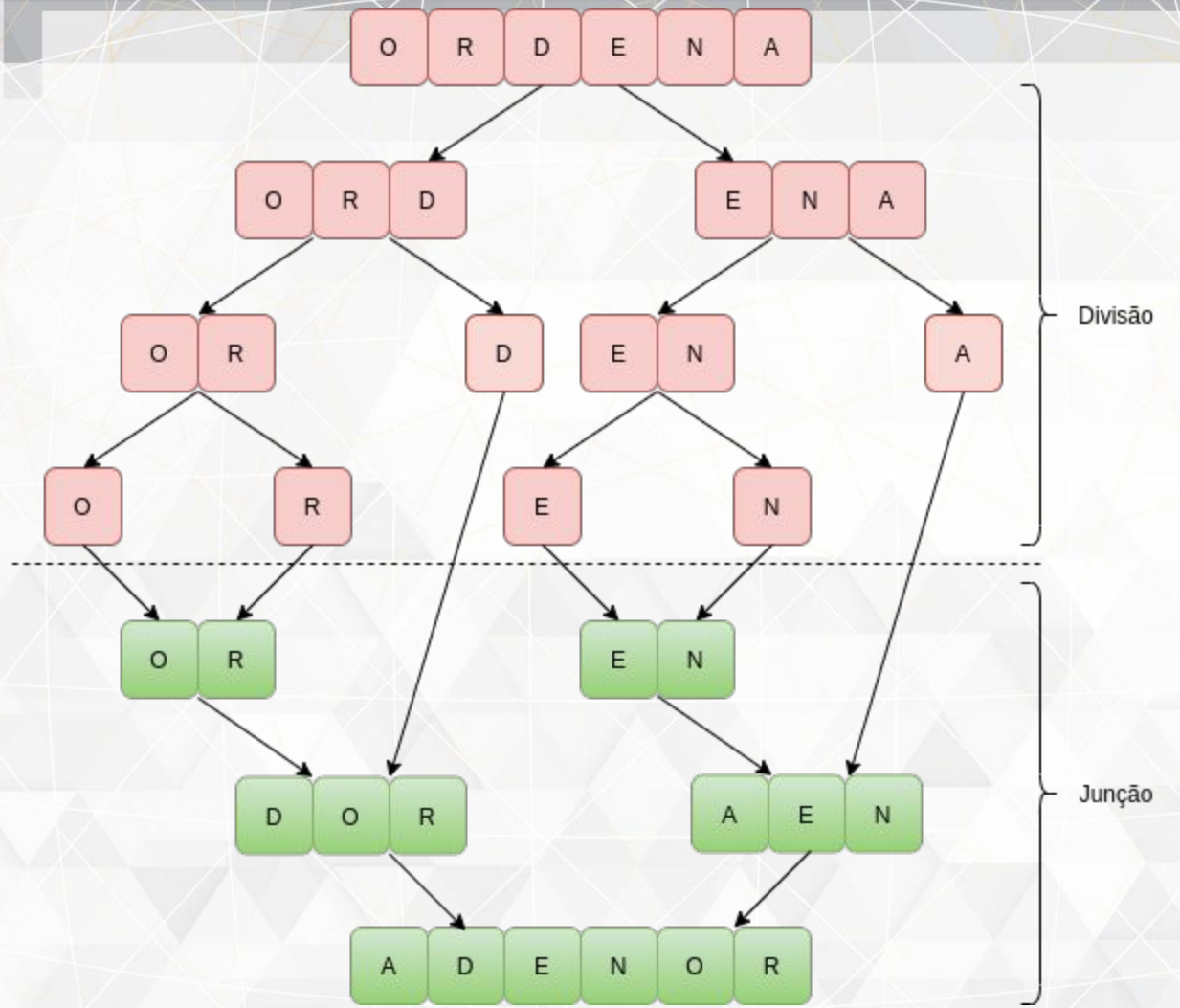
Merge Sort - Junção



Merge Sort - Junção



Merge Sort - Junção



Merge Sort - Divisão

- Vetor
  - v → 

- Tamanho
  - tam → 6

```
//ALGORITMO MERGE SORT EM C
void merge_sort(int * v, int tam){
   if(tam >= 2){
      int meio = tam / 2;
      merge_sort(v, meio);
      merge_sort(v + meio, tam - meio);
      merge(v, tam);
   }
}
```

Merge Sort - Junção

```c
void merge(int* v, int tamanho){
    int *novo = (int*) calloc(tamanho, sizeof(int));
    int meio = tamanho / 2;
    int i = 0, j = meio, k = 0;
    while((i < meio) && (j < tamanho)){//junto os vetores
        if(v[i] < v[j]){
            novo[k++] = v[i++];
        }
        else{
            novo[k++] = v[j++];
        }
    }
    if(i == meio){//Caso ainda haja elementos na primeira metade
        while(j < tamanho){
            novo[k++] = v[j++];
        }
    }
    else{
        while(i < meio){ //Caso ainda haja elementos na segunda metade
            novo[k++] = v[i++];
        }
    }
    for(i = 0; i < tamanho; i++){//Move os elementos de volta para o vetor original
        v[i] = novo[i];
    }
    free(novo);
}
```

BucketSort - O que é

- Bucket sort, ou bin sort, é um algoritmo de ordenação que funciona dividindo um vetor em um número finito de recipientes.
- Cada recipiente é então ordenado individualmente, seja usando um algoritmo de ordenação diferente, ou usando o algoritmo bucket sort recursivamente.

BucketSort - Complexidade

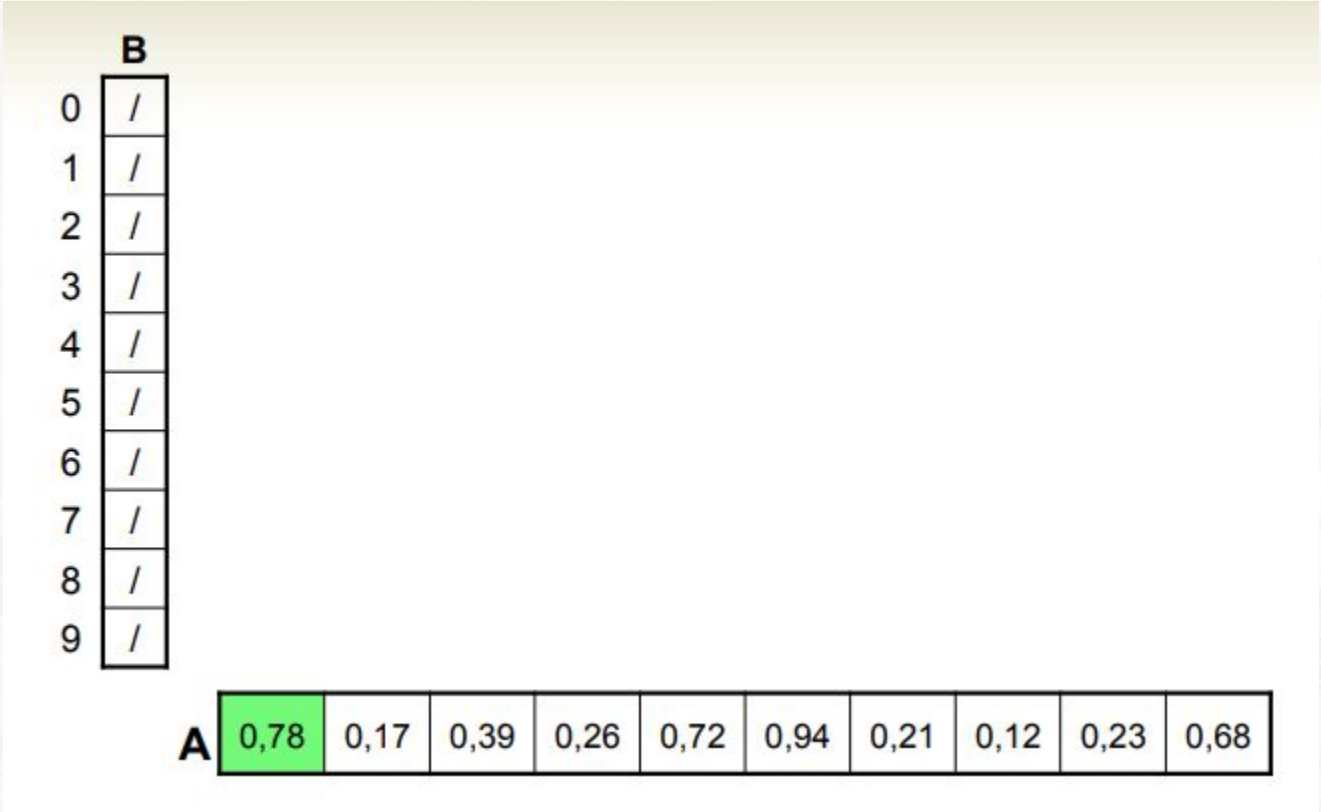
Bucket sort	
classe	Algoritmo de ordenação
estrutura de dados	Array, Listas ligadas
complexidade pior caso	$O(n^2)$
complexidade caso médio	$O(n+k)$
complexidade melhor caso	$O(n+k)$
Algoritmos	

Esta caixa: ver · discutir

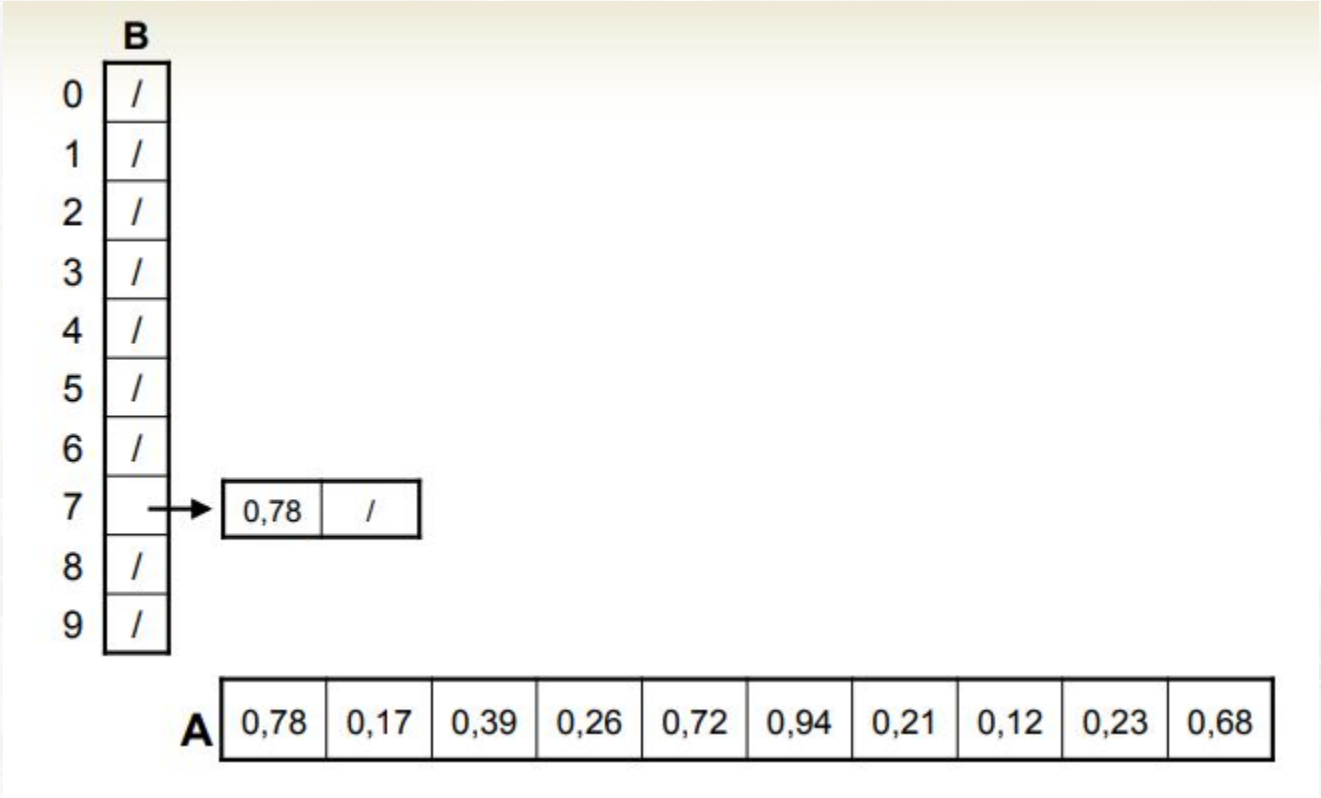
BucketSort - Funcionamento

1. Inicialize um vetor de "baldes", inicialmente vazios.
2. Vá para o vetor original, incluindo cada elemento em um balde.
3. Ordene todos os baldes não vazios.
4. Coloque os elementos dos baldes que não estão vazios no vetor original.

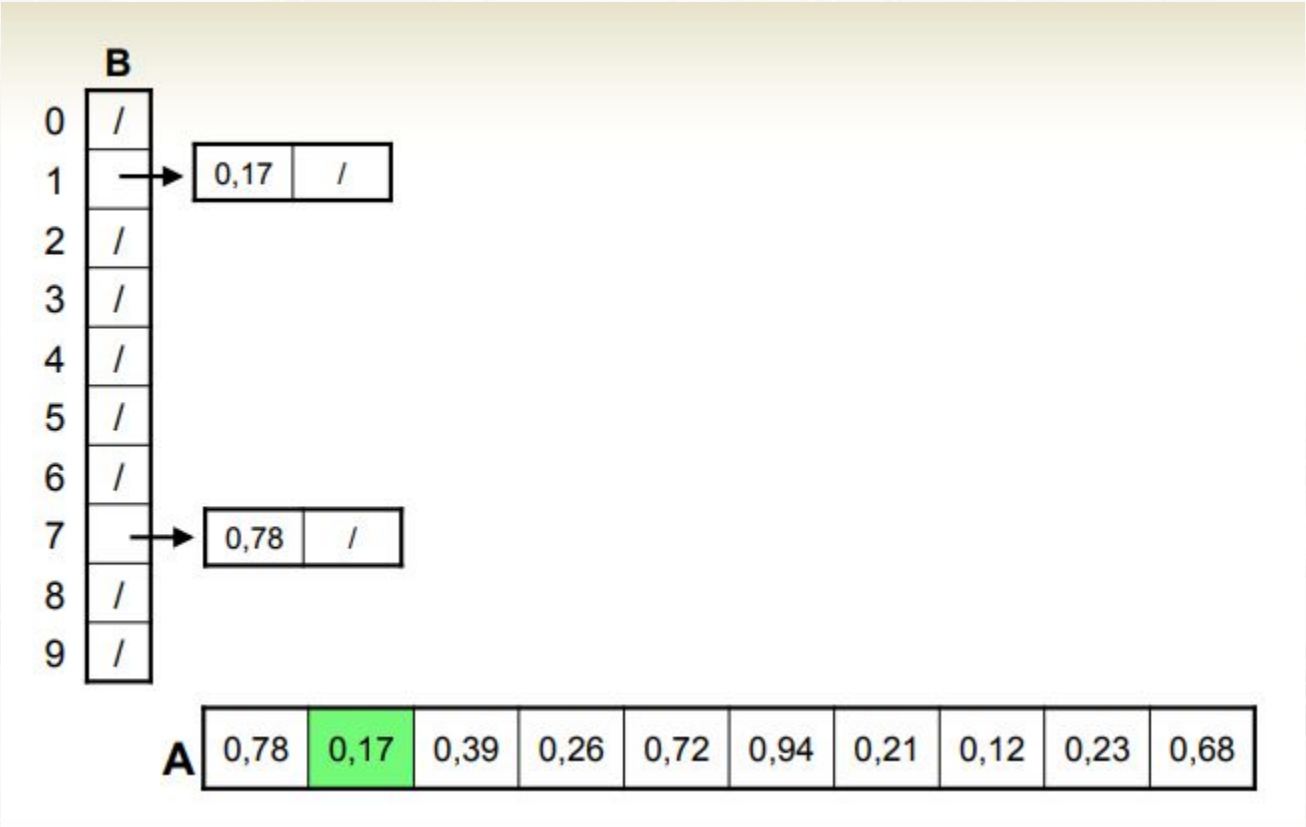
BucketSort - Passo a Passo



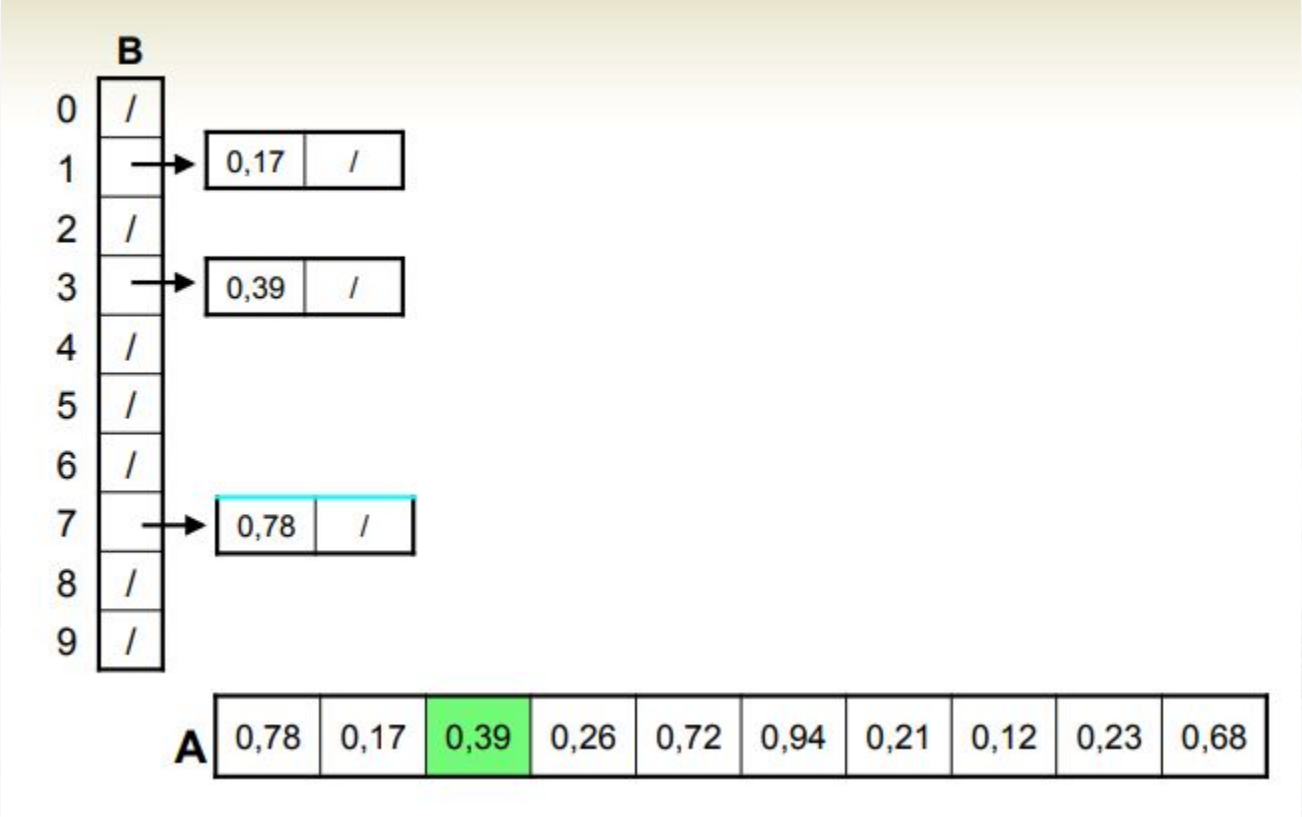
BucketSort - Passo a Passo



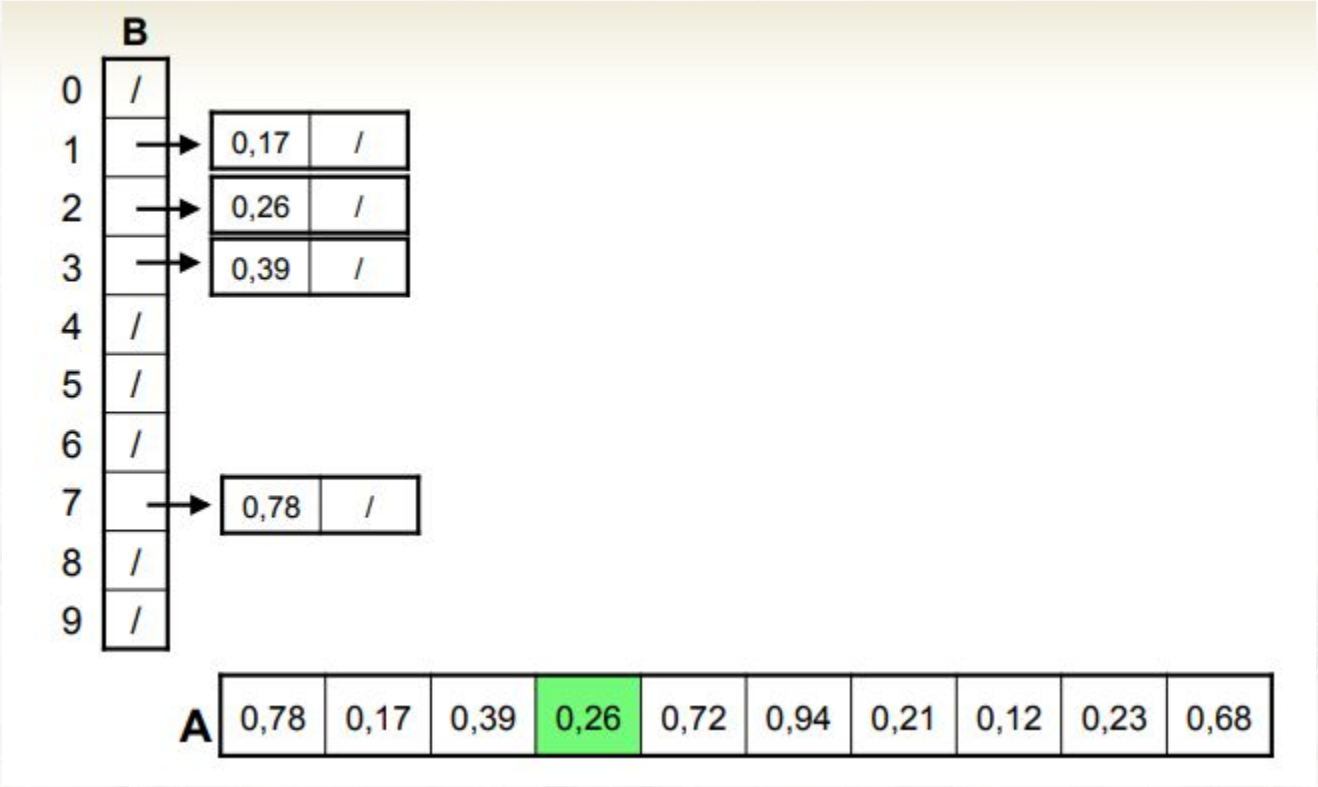
BucketSort - Passo a Passo



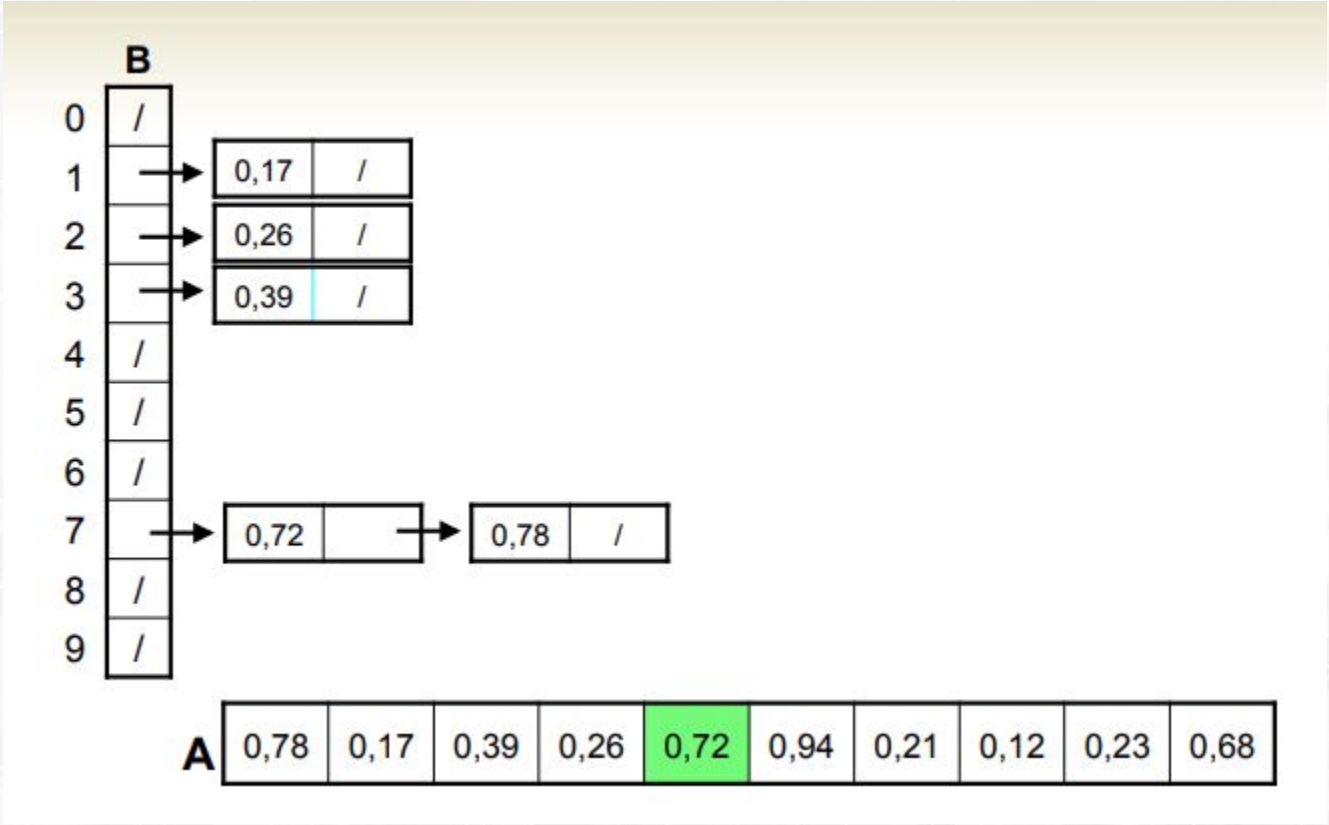
BucketSort - Passo a Passo



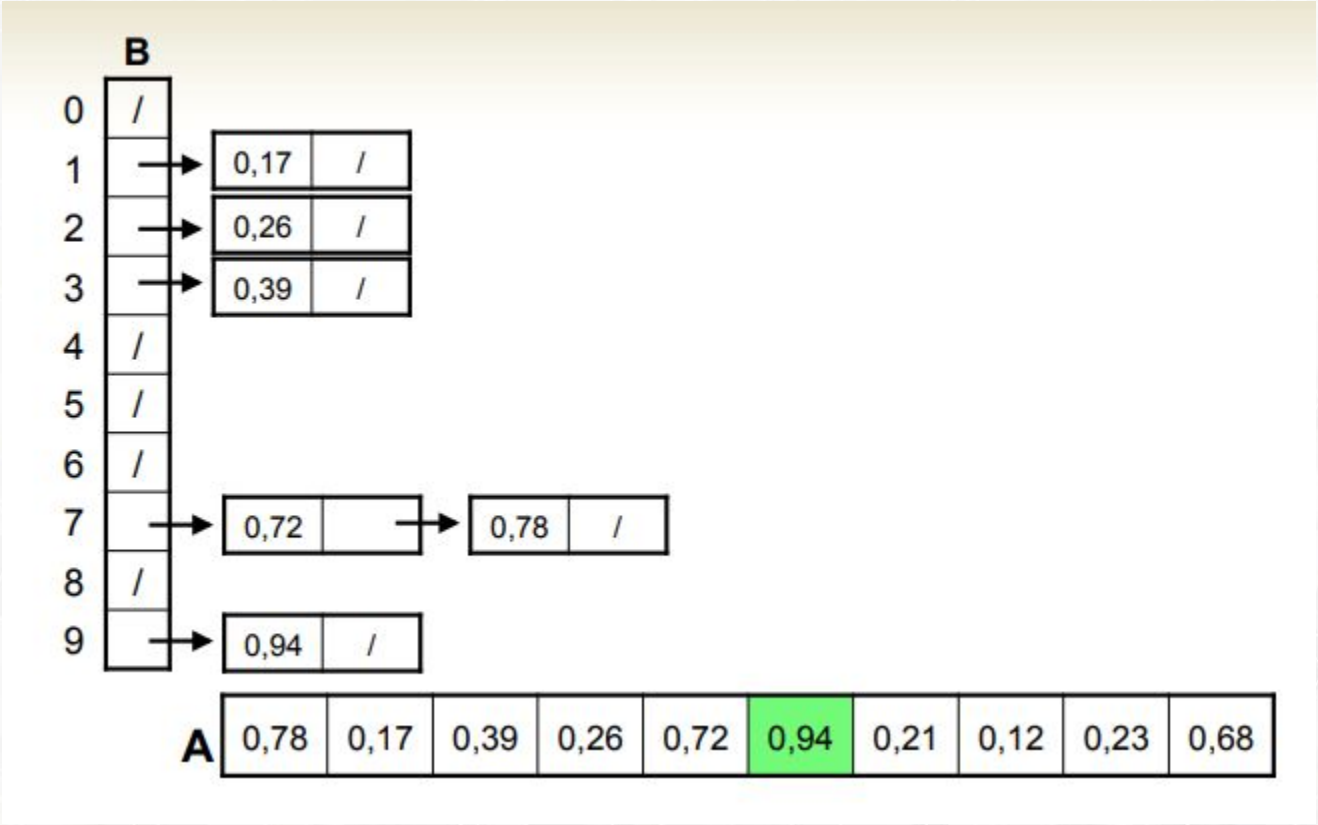
BucketSort - Passo a Passo



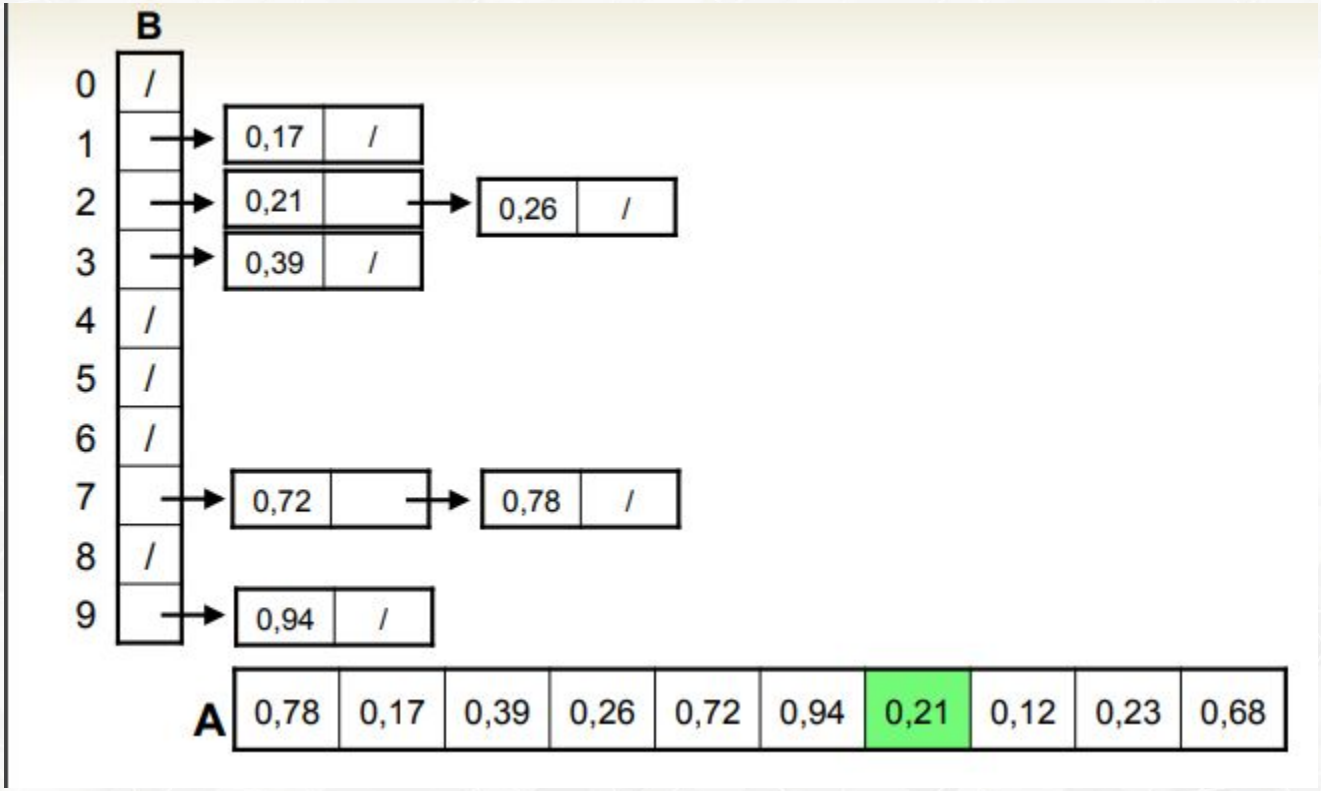
BucketSort - Passo a Passo



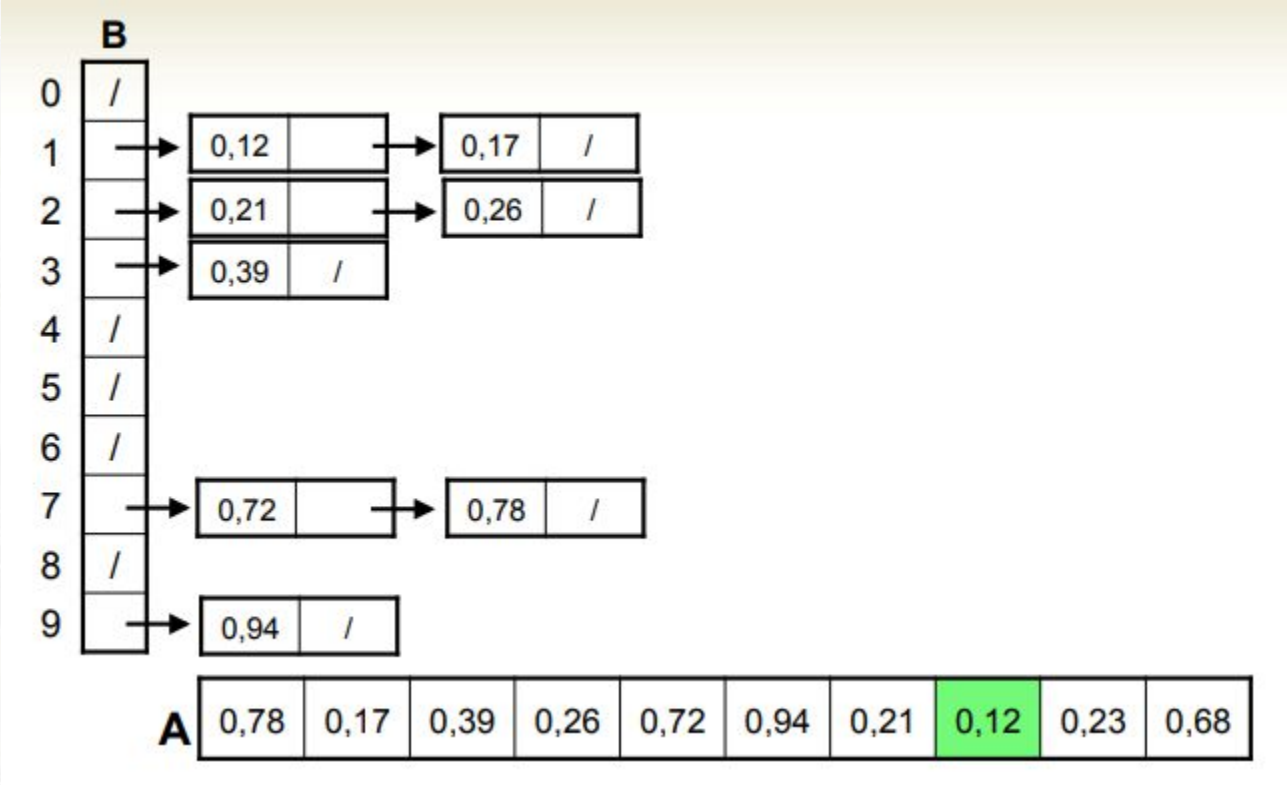
BucketSort - Passo a Passo



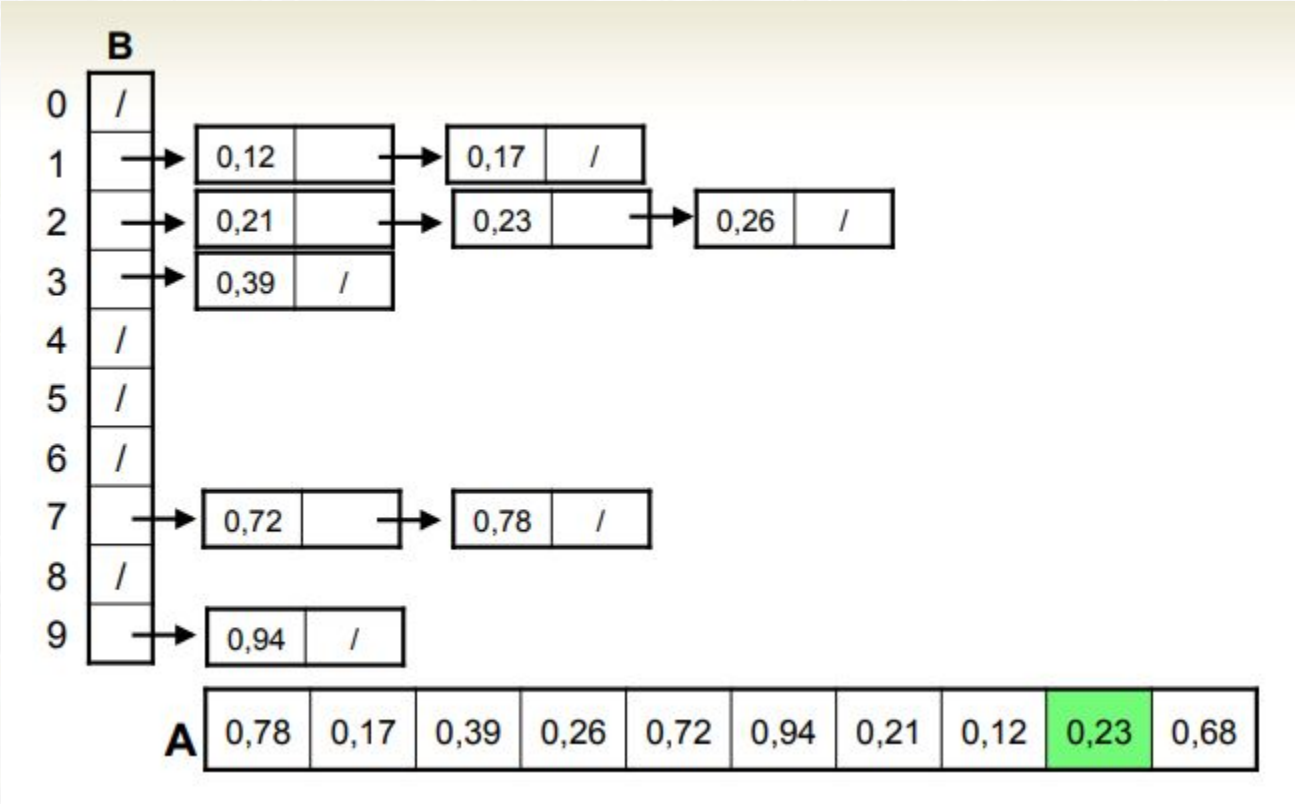
BucketSort - Passo a Passo



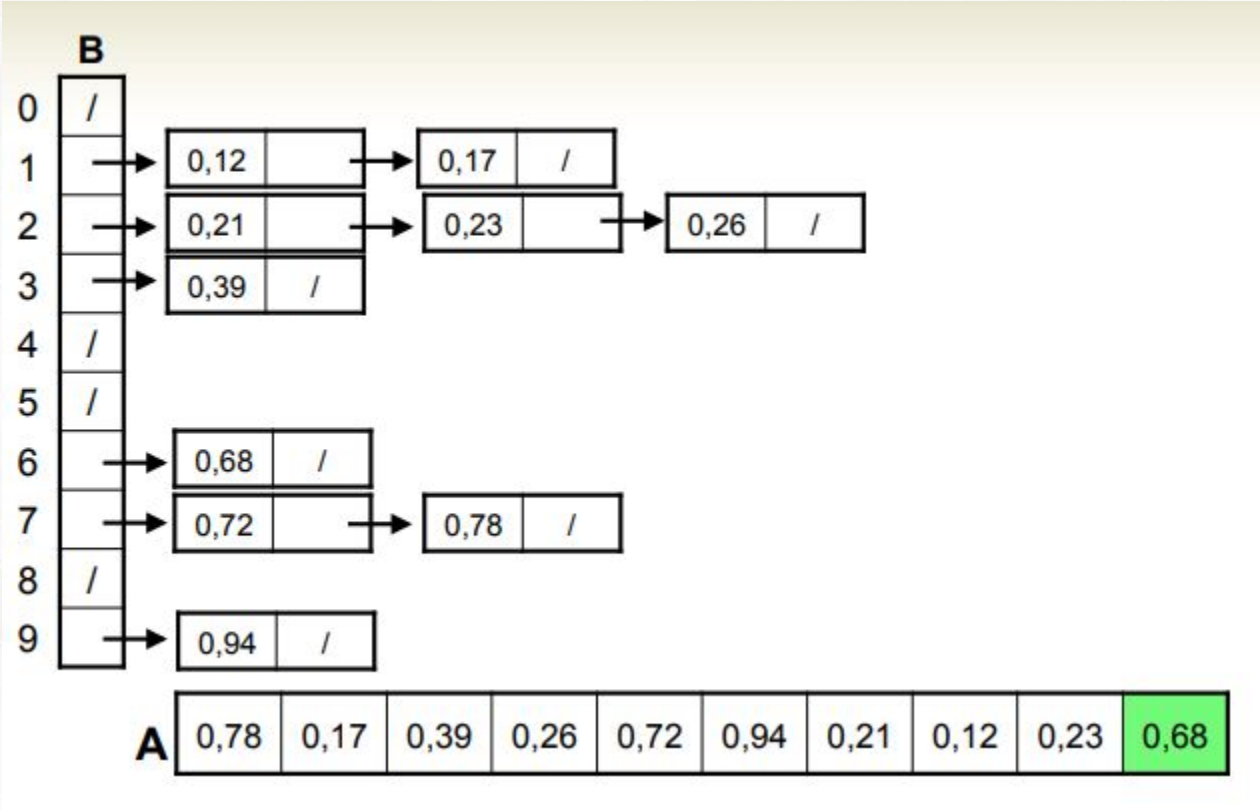
BucketSort - Passo a Passo



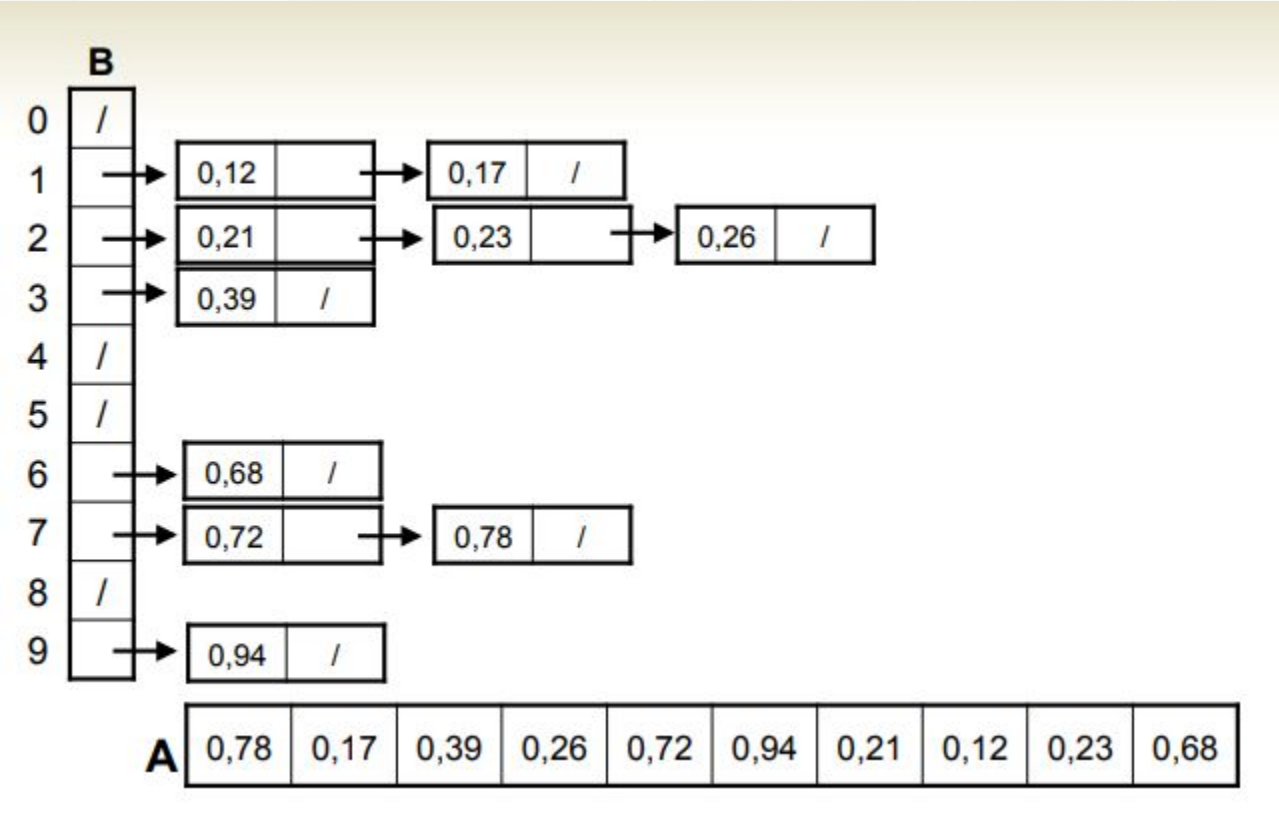
BucketSort - Passo a Passo



BucketSort - Passo a Passo



BucketSort - Passo a Passo



```
27  void bucketSort (int a[], int len){
28      int bucket[10][len], bucket_cnt[10];
29      int i, j, k, r, divisor = 10;
30      for (i = 0; i < 10; i++)
31        bucket_cnt[i] = 0;//contador de elementos salvos por slot(0 a 9) na matrix auxiliar
32      for (i = 0; i < len; i++){
33        r = a[i] / divisor; //Pega o digito correto do numero em a[i]
34        if (r<1)
35            r=0;
36        if (r>9)
37            r=9;
38        bucket[r][bucket_cnt[r]] = a[i];//Salva a[i] na posicao correta da matriz
39        bucket_cnt[r] += 1;//delimita quantos elementos existem por indice na matriz
40      }
41      i = 0;
42      for (k = 0; k < 10; k++){
43          if (bucket_cnt[k]==0) continue;
44          insertionSort(bucket[k],bucket_cnt[k]);
45          for (j = 0; j < bucket_cnt[k]; j++){
46              a[i] = bucket[k][j];//Armazena no array original
47              i++;
48          }
49      }
50  }
```

Quick Sort - O que é

- Método de ordenação muito rápido e eficiente, inventado por Charles Antony Richard Hoare em 1960, quando visitou a Universidade de Moscovo como estudante

- Foi criado ao tentar traduzir um dicionário de inglês para russo, ordenando as palavras, tendo como objetivo reduzir o problema original em subproblemas que possam ser resolvidos mais fácil e rápido.

Quick Sort - Complexidade

classe	Algoritmo de ordenação
estrutura de dados	Array, Listas ligadas
complexidade pior caso	$O(n^2)$
complexidade caso médio	$O(n \log n)$
complexidade melhor caso	$O(n \log n)$
complexidade de espaços pior caso	$O(n)$
otimo	Não
estabilidade	não-estável

Quick Sort - Funcionamento

- Adota a estratégia de divisão e conquista.
- Um elemento é escolhido como pivô.
- Os dados são rearranjados (valores menores do que o pivo são colocados antes dele e os maiores, depois)
- Recursivamente ordena as 2 partições

Quick Sort - Passos

Os passos são:

1.1. Escolha um elemento da lista, denominado pivô;
1.2. Particiona: rearrange a lista de forma que todos os elementos anteriores ao pivô sejam menores que ele, e todos os elementos posteriores ao pivô sejam maiores que ele. Ao fim do processo o pivô estará em sua posição final e haverá duas sub listas não ordenadas. Essa operação é denominada partição;
1.3. Recursivamente ordene a sub lista dos elementos menores e a sublista dos elementos maiores;

Quick Sort (Código)

```
void QuickSort(int* V, int inicio, int fim){
        int pivo;
        if (fim > inicio) {
                pivo = particiona(V, inicio,
fim);
                quickSort(V, inicio, pivo-1);
                quickSort(V, pivo+1, fim);
        }
}
```

separa os dados em 2 partições

chama a função para as 2 metades

```
int particiona (int *V, int inicio, int
final){
        int esq, dir, pivo, aux
        esq = inicio;
        dir = final;
        pivo = V[inicio];
        while(esq<dir){
          while(v[esq] <=pivo)
                esq++;
          while(v[dir] >pivo)
                dir--;
          if (esq<dir){
                aux = V [esq];
                V [esq] =  v[dir];
                V[dir] = aux;
          }
        }
        V[inicio] = V[dir];
        V[dir] = pivo;
        return dir;
```

Avança posição da esquerda

Avança posição da esquerda

Troca esq e dir

Quick Sort - Passo a Passo

Pivo Esq Dir

0	1	2	3	4
25	57	86	48	37

	0	1	2	3
25	57	37	48	86

0	1	2	3	4
25	57	86	48	37

25	48	37	57	86

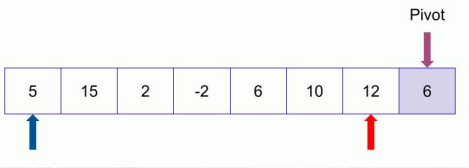
0	1	2	3	4
25	57	86	48	37

	0	1		
25	48	37	57	86

	0	1	2	3
25	57	37	48	86

25	37	48	57	86

```
while(esq<dir){
    while(v[esq] <=pivo)
            esq++;
    while(v[dir] >pivo)
            dir--;
    if (esq<dir){
            aux = V [esq];
            V [esq] =  v[dir];
            V[dir] = aux;
    }
}
```



Radix Sort

462	273	1465	722	383

0		
1		
2	462	722
3	273	383
4		
5	1465	
6		
7		
8		
9		

- Faz uso de uma estrutura auxiliar (tipicamente uma matriz) para realizar a ordenação.
- Operações de cópia de elementos realizadas a cada iteração.

Radix Sort

462	722	273	383	1465

0		
1		
2	722	
3		
4		
5		
6	462	1465
7	273	
8	383	
9		

Radix Sort

722	462	1465	273	383

0		
1		
2	273	
3	383	
4	462	1465
5		
6		
7	722	
8		
9		

Radix Sort

273	383	462	1465	722

0	273	383	462	722
1	1465			
2				
3				
4				
5				
6				
7				
8				
9				

Finalmente, os elementos que já tiverem todas as dezenas “visitadas”, irão ser alocadas para o 0 na mesma sequência

Radix Sort

273	383	462	722	1465

- A complexidade do Radix Sort é baseada no tamanho dos números no array a ser ordenado, nesse caso seria 4N.
- Em teoria, deveria ser um método de ordenação muito eficiente, afinal sua complexidade é linear.
- Na prática ele não é tão excelente, devido a necessidade de realização de operações de cópia entre o array e a estrutura auxiliar para realizar a ordenação.

```
void radix_sort (int a[], int len){
   int bucket[10][10], bucket_cnt[10];
   int i, j, k, r, numberOperations = 0, divisor = 1, maxElement, pass;
   maxElement = get_max (a, len);
   while (maxElement > 0){//Conta numero de operações
      numberOperations++;
      maxElement /= 10;
   }
   for (pass = 0; pass < numberOperations; pass++){
      for (i = 0; i < 10; i++)
         bucket_cnt[i] = 0;//contador de elementos salvos por slot(0 a 9) na matrix auxiliar
      for (i = 0; i < len; i++){
         r = (a[i] / divisor) % 10; //Pega o digito correto do numero em a[i]
         bucket[r][bucket_cnt[r]] = a[i];//Salva a[i] na posicao correta da matriz
         bucket_cnt[r] += 1;//delimita quantos elementos existem por indice na matriz
      }
      i = 0;
      for (k = 0; k < 10; k++){
         for (j = 0; j < bucket_cnt[k]; j++){
            a[i] = bucket[k][j];//Realoca vetor para proxima iteração
            i++;
         }
      }
      divisor *= 10;
   }
}
```

Counting Sort

- Ordenação por contagem

Algo que chama a atenção em primeiro momento é:

Como é possível ordenar elementos sem utilizar comparação?

Counting Sort

estrutura de dados	Array, Listas ligadas
complexidade pior caso	$O(n+k)$
complexidade caso médio	$O(n+k)$
complexidade melhor caso	$O(n+k)$

Counting Sort

- Os valores a serem ordenados devem ser números inteiros positivos.

- A ideia geral é podermos mapear o valor presente em uma sequência para a posição de mesmo valor em um array auxiliar (array[i] = i)

- Saber o maior valor valor do array, o qual chamamos de K.

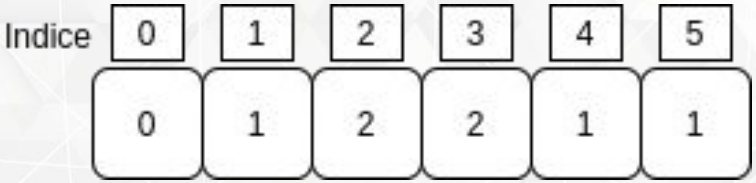
Counting Sort

Entrada A ->

Passo 1 - encontrar o maior valor -> k =

Passo 2 - Registrar a frequência dos elementos de A em um vetor auxiliar de tamanho K

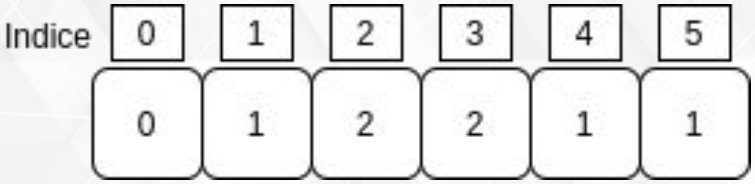
B ->



Counting Sort

Passo 3 - Calcular a soma cumulativa de B. Esse passo registra, para cada elemento *x* da entrada, o número de elementos menores ou iguais a *x*;

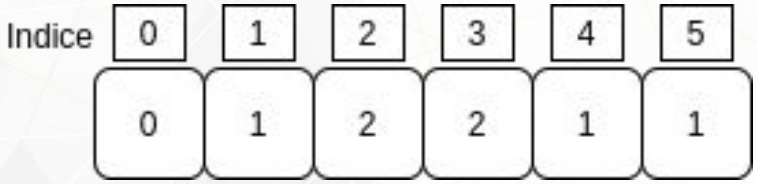
Vetor B



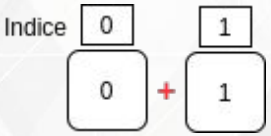
- Como a posição 0 não possuir um valor inferior a ela, deve-se começar a soma a partir da posição 1

Counting Sort

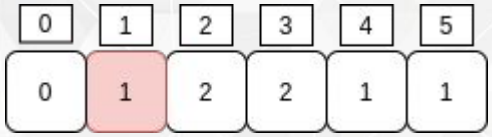
Vetor B



Soma dos elementos 0 e 1 ->

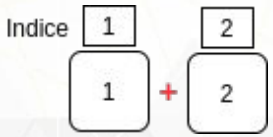


Novo vetor B



Counting Sort

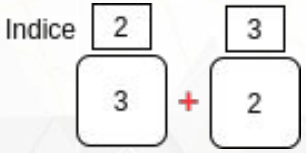
Soma dos elementos 1 e 2 ->



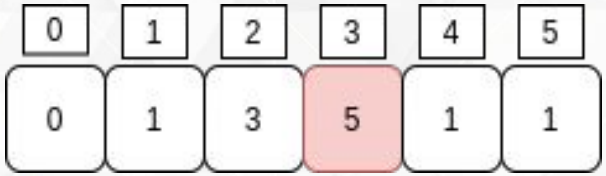
Novo vetor B

Counting Sort

Soma dos elementos 2 e 3 ->

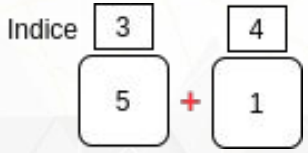


Novo vetor B



Counting Sort

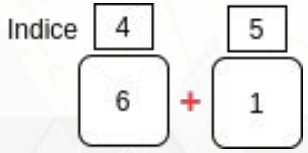
Soma dos elementos 3 e 4 ->



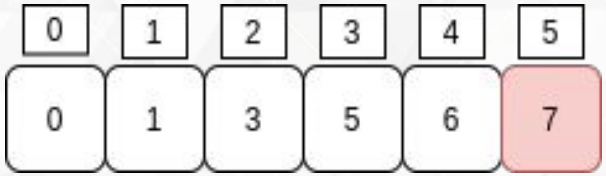
Novo vetor B

Counting Sort

Soma dos elementos 4 e 5 ->



Novo vetor B



Counting Sort

Passo 4

Iterar sobre *A* do fim ao início registrando em *R* o valor de *A* com as seguintes instruções:

```
B[A[i]]--;
R[B[A[i]]] = A[i];
```

Não se assuste. Essa sequência de decrementos em 1 é devido ao fato de começarmos os índices de um array a partir do zero em B;

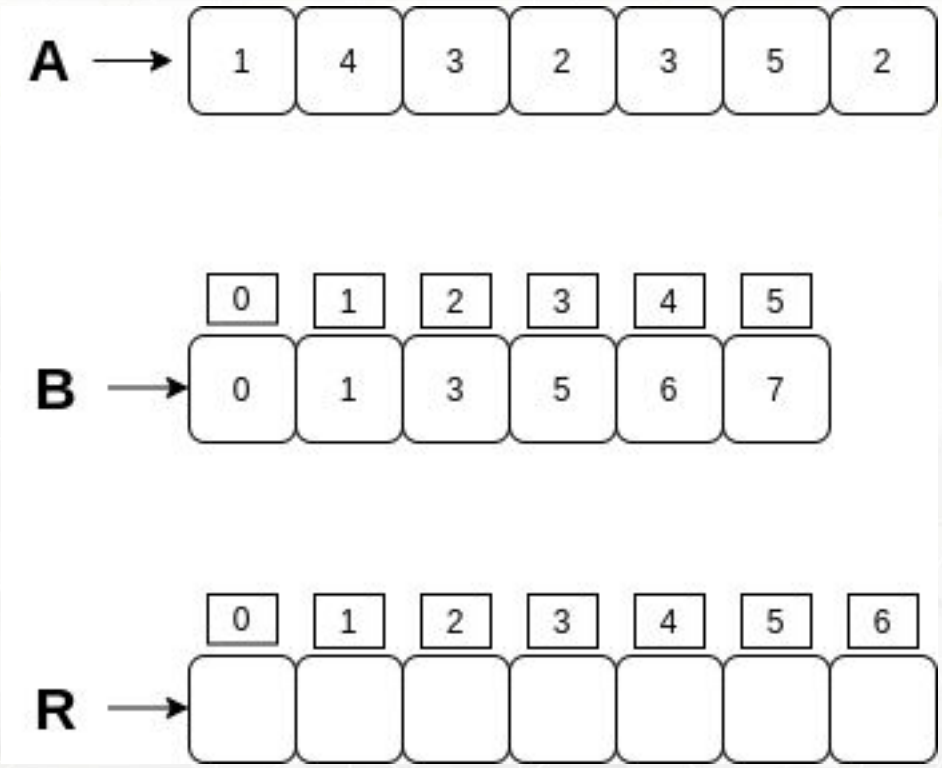
Counting Sort

Seguindo a instrução:

→ i = tamA - 1

B[A[i]]--;

R[B[A[i]]] = A[i];



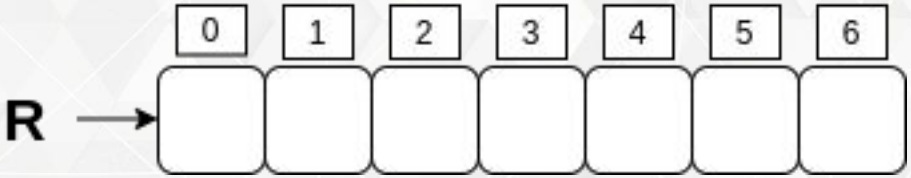
Counting Sort

Seguindo as instruções:

→ i = 6

B[A[i]]--;

R[B[A[i]]] = A[i];



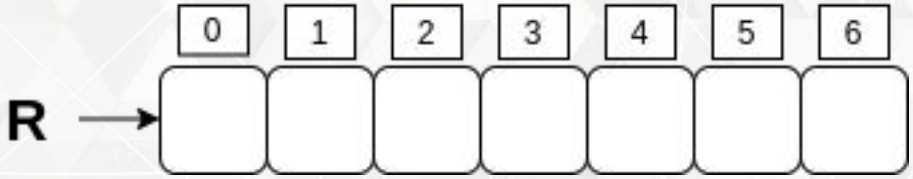
Counting Sort

Seguindo as instruções:

→ i = 6

B[2]--;

R[B[2]] =2;



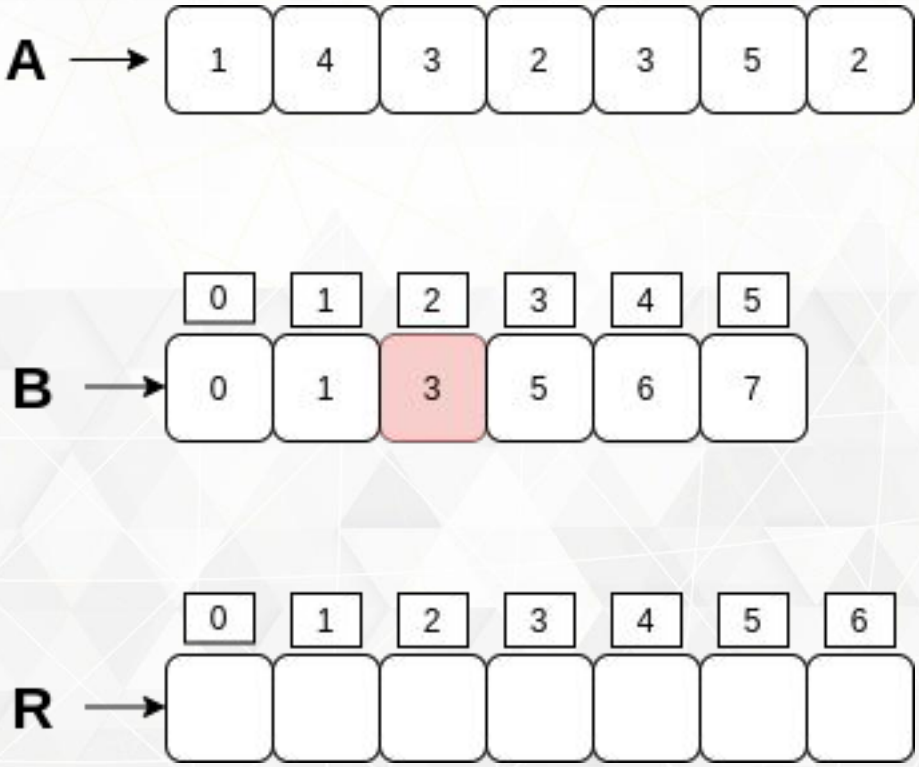
Counting Sort

Seguindo a instrução:

i = 6

→ B[2]--;

R[B[2]] =2;

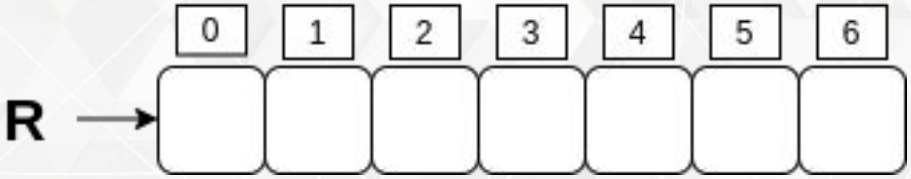


Counting Sort

Seguindo a instrução:

i = 6

→ R[B[2]] =2;

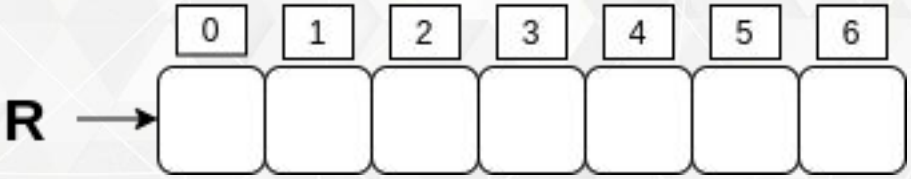


Counting Sort

Seguindo a instrução:

i = 6

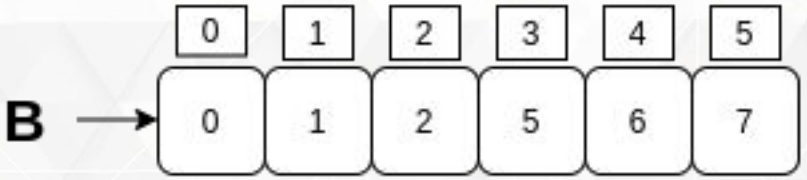
→ R[2] =2;



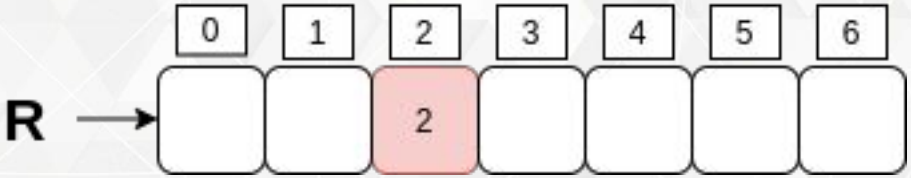
Counting Sort

Seguindo a instrução:

i = 6



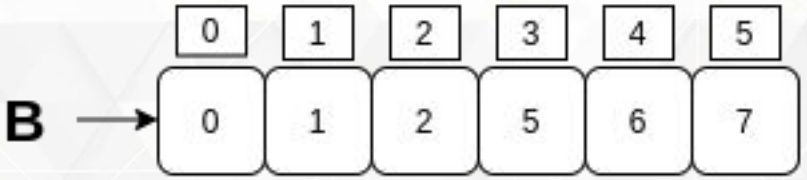
→ R[2] =2;



Counting Sort

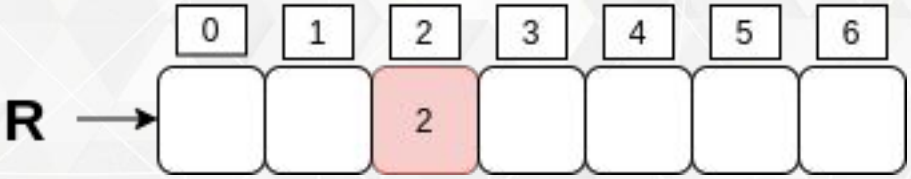
Seguindo a instrução:

→ i = 5



B[A[i]]--;

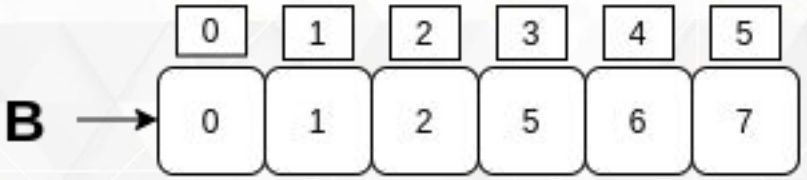
R[B[A[i]]] = A[i];



Counting Sort

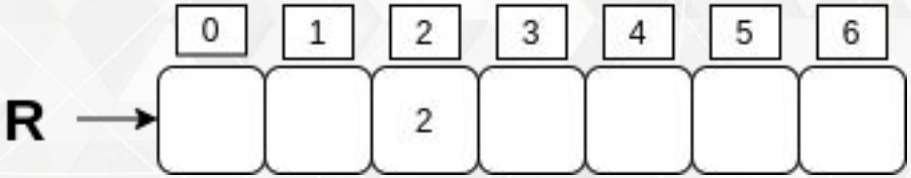
Seguindo as instruções:

→ i = 5



B[A[i]]--;

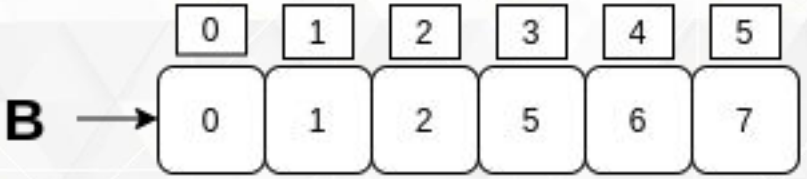
R[B[A[i]]] = A[i];



Counting Sort

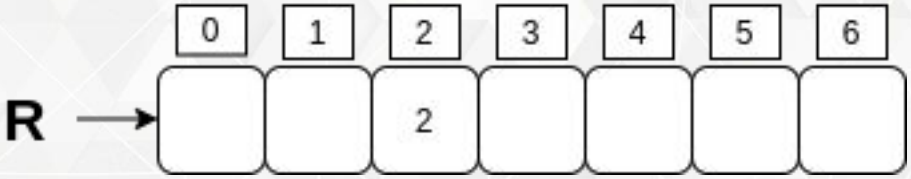
Seguindo as instruções:

→ i = 5



B[5]--;

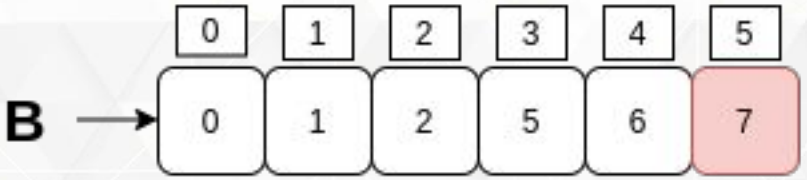
R[B[5]] = 5;



Counting Sort

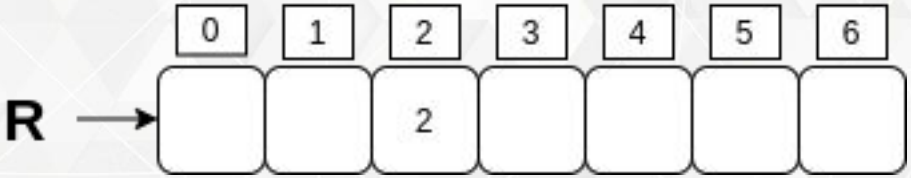
Seguindo a instrução:

i = 5



→ B[5]--;

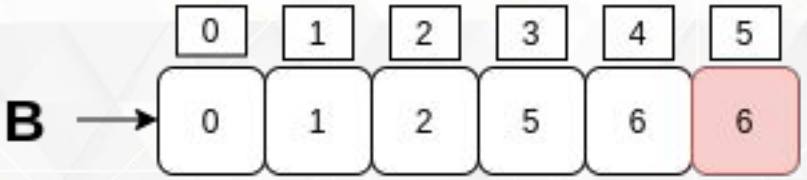
R[B[5]] = 5;



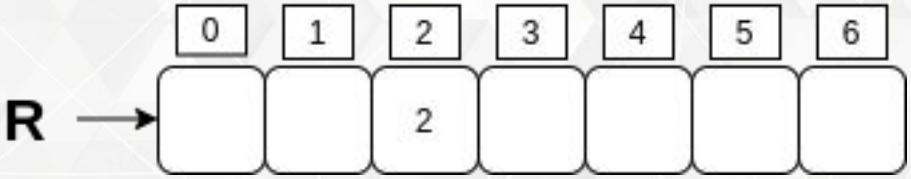
Counting Sort

Seguindo a instrução:

i = 5



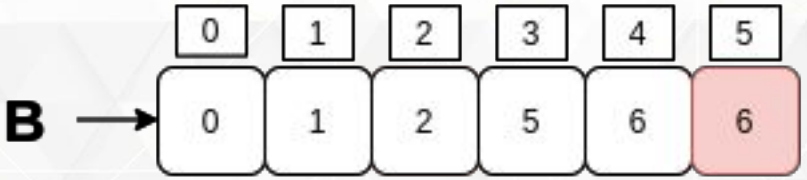
→ R[B[5]] = 5;



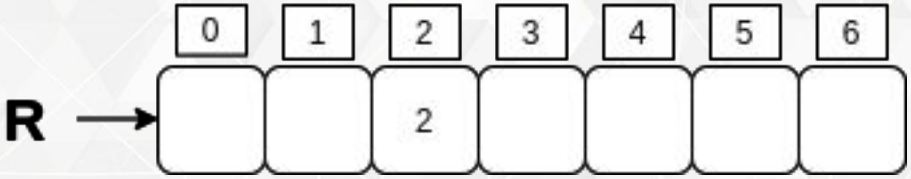
Counting Sort

Seguindo a instrução:

i = 5



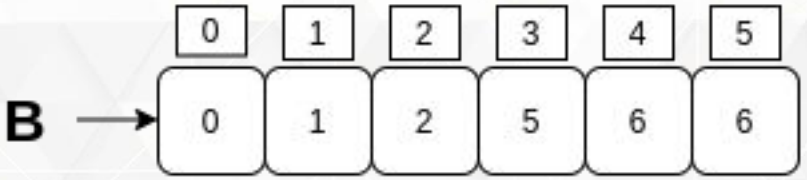
→ R[6] = 5;



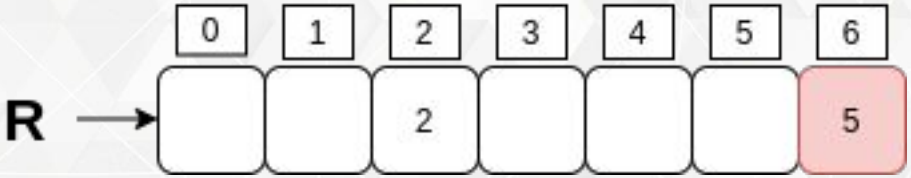
Counting Sort

Seguindo a instrução:

i = 5



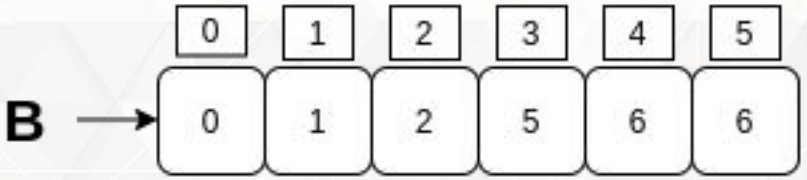
→ R[6] = 5;



Counting Sort

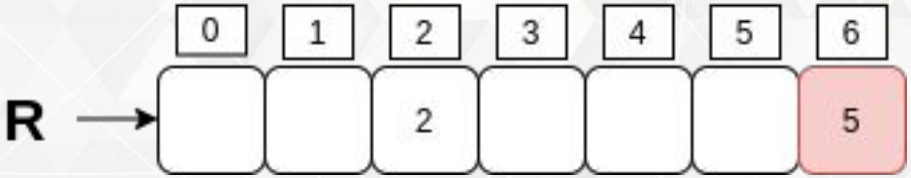
Seguindo a instrução:

→ i = 4



B[A[i]]--;

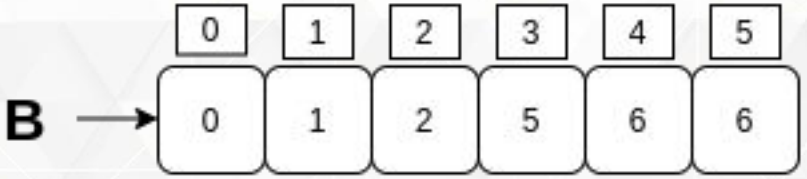
R[B[A[i]]] = A[i];



Counting Sort

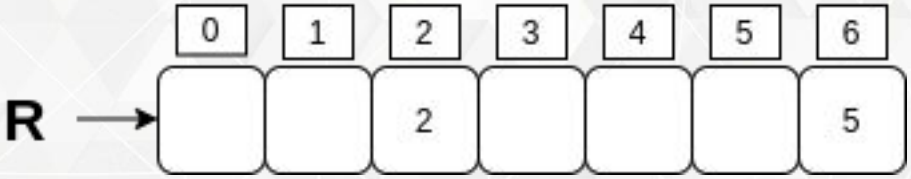
Seguindo as instruções:

→ i = 4



B[A[i]]--;

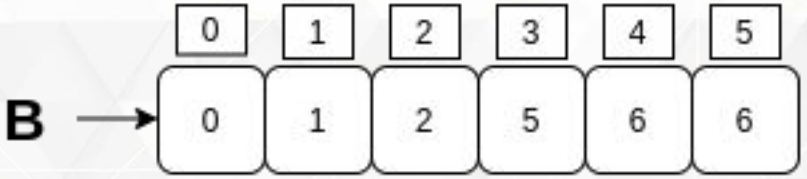
R[B[A[i]]] = A[i];



Counting Sort

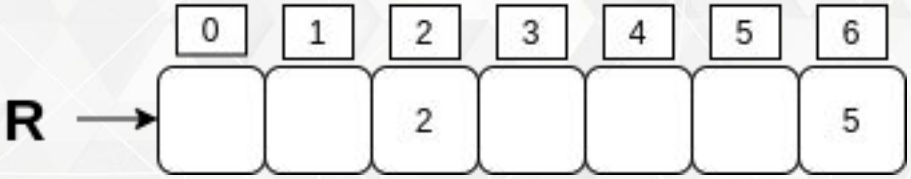
Seguindo as instruções:

→ i = 4



B[3]--;

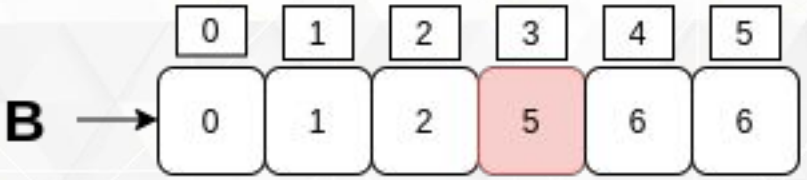
R[B[3]] = 3;



Counting Sort

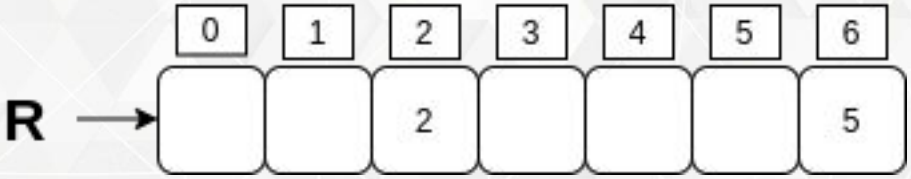
Seguindo a instrução:

i = 4



→ B[3]--;

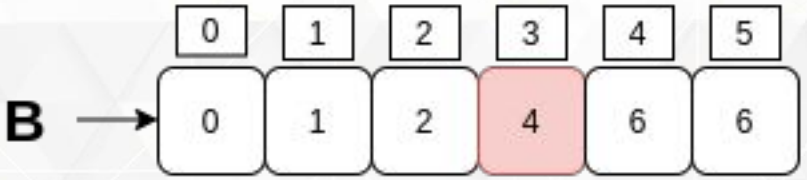
R[B[3]] = 3;



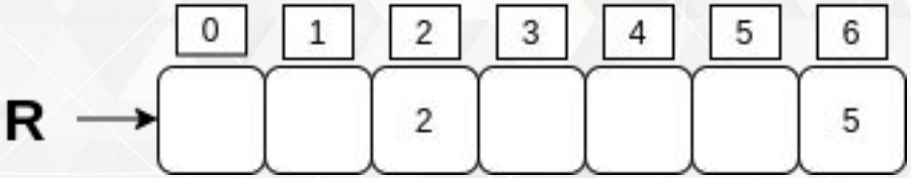
Counting Sort

Seguindo a instrução:

i = 4

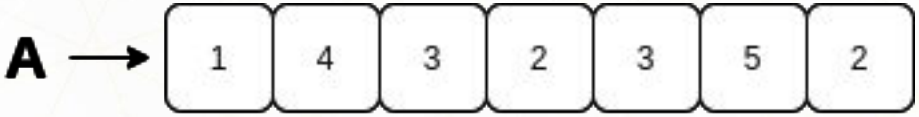


→ R[B[3]] = 3;

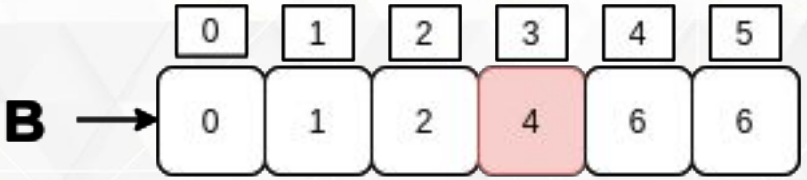


Counting Sort

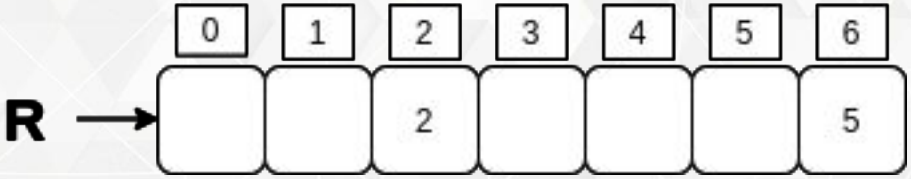
Seguindo a instrução:



i = 4



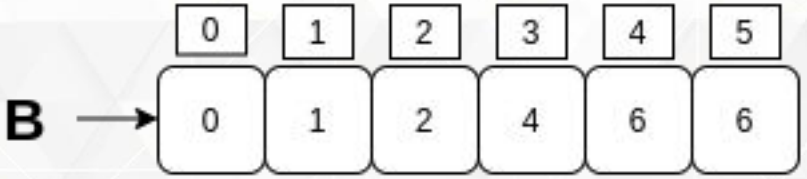
→ R[4] = 3;



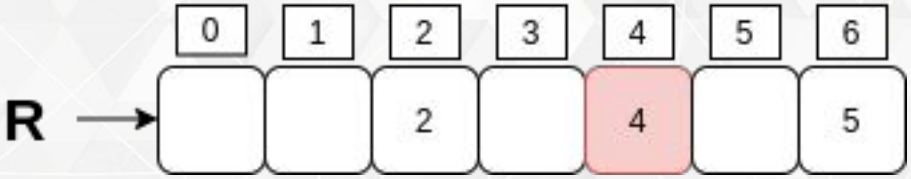
Counting Sort

Seguindo a instrução:

i = 4



→ R[4] = 3;



Counting Sort

```
void counting_sort(int* vet, int tam){
    int maior = maximo(vet, tam);
    int vetorB[maior + 1];
    int vetorC[tam];
    int i;
    for(i = 0; i < maior + 1; i++){
        vetorB[i] = 0;
    }
    for(i = 0; i < tam; i++){
        vetorB[vet[i]]++;
        vetorC[i] = 0;
    }
    for(i = 1; i < maior + 1; i++){
        vetorB[i] = vetorB[i - 1] + vetorB[i];
    }
    for(i = tam - 1; i >= 0; i--){
        vetorB[vet[i]]--;
        vetorC[vetorB[vet[i]]] = vet[i];
    }
    for(i = 0; i < tam; i++){
        vet[i] = vetorC[i];
    }
}
```

Exercícios

Executem casos de testes com entradas de 10.000 elementos. Os casos de teste deverão ter um vetor ordenado crescente, decrescente e aleatório.

Apresente uma comparação dos casos de testes, analisando o consumo de memória e tempo de execução. Utilizar dois (2) gráficos, um para cada análise.

Dúvidas?