



**ALGORITMOS E
ESTRUTURAS DE DADOS
CES-11**
Prof. Paulo André Castro
pauloac@ita.br
Sala 110 - Prédio da Computação
www.comp.ita.br/~pauloac
IECE - ITA

MÉTODOS MAIS EFICIENTES QUE $O(N^2)$

- Método Quicksort
- Método Heap-Sort
- Método MergeSoft

2

O MÉTODO QUICK-SORT

Generalidades e fundamentos

- **Quick-Sort** é provavelmente o algoritmo de ordenação **mais usado** por programadores em geral. Foi criado por C.Hoare em 1960.
- Durante muito tempo, pesquisadores de métodos de ordenação se dedicaram mais a **aperfeiçoar o Quick-Sort** do que a procurar **novos métodos**.

3

- No caso médio, Quick-Sort é $O(n \log_2 n)$ e no pior caso é $O(n^2)$.
- Quick-Sort é **recursivo**, mas pode ser transformado em **não recursivo**, mediante o uso de **pilha**.
- Quick-Sort aplica a técnica denominada **divisão e conquista**, que reparte o vetor a ser ordenado em dois sub-vetores e os ordena independentemente.

4

O método consiste em:

1. Escolher, mediante um critério razoável, um **pivô** entre todos os elementos do vetor a ser ordenado

Bom é quando tal **pivô** é a **mediana** de todos os elementos, ou seja, quando ele consegue dividir o vetor ao meio

Exemplo: seja o vetor

V

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

Seja o maior entre os dois elementos distintos mais à esquerda escolhido como **pivô** (**pivô = 5**)

5

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes

left ↓

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

 ↓ right

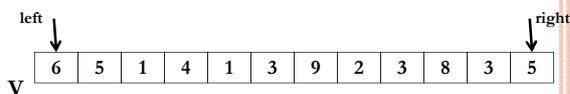
pivô = 5

Incondicionalmente, trocar **V[left]** com **V[right]**

(para facilitar a elaboração do algoritmo)

6

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes



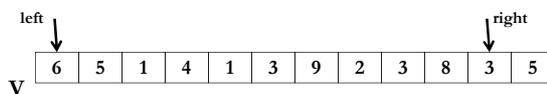
pivô = 5

Mover **left** para a direita até $V[\text{left}] \geq \text{pivô}$

Mover **right** para a esquerda até $V[\text{right}] < \text{pivô}$

7

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes

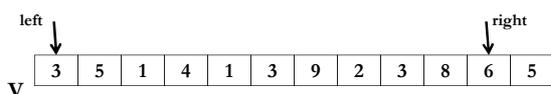


pivô = 5

Trocar $V[\text{left}]$ com $V[\text{right}]$

8

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes



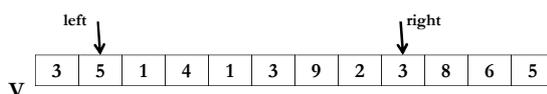
pivô = 5

Mover **left** para a direita até $V[\text{left}] \geq \text{pivô}$

Mover **right** para a esquerda até $V[\text{right}] < \text{pivô}$

9

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes

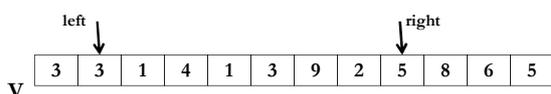


pivô = 5

Trocar $V[\text{left}]$ com $V[\text{right}]$

10

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes



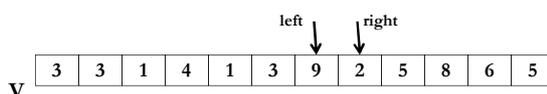
pivô = 5

Mover **left** para a direita até $V[\text{left}] \geq \text{pivô}$

Mover **right** para a esquerda até $V[\text{right}] < \text{pivô}$

11

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes



pivô = 5

Trocar $V[\text{left}]$ com $V[\text{right}]$

12

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes

pivô = 5

Mover **left** para a direita até $V[\text{left}] \geq \text{pivô}$

Mover **right** para a esquerda até $V[\text{right}] < \text{pivô}$

13

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes

pivô = 5

A divisão está completa pois $\text{left} > \text{right}$

14

2. Dividir o vetor em 2 partes: **sub-vetor baixo** e **sub-vetor alto**, colocando no primeiro, todos os elementos **menores** que o pivô, e no segundo, os elementos restantes

pivô = 5

Pior caso: um dos sub-vetores com um só elemento e o outro com mais de um

15

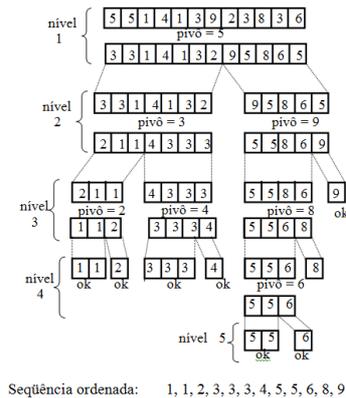
Aplicar **recursivamente** os passos anteriores nos 2 sub-vetores formados

Encerrar a recursividade quando, num dado sub-vetor, todos os seus elementos forem iguais

Outros critérios para a escolha do pivô:

- Um dos elementos extremos do vetor
- Média de três elementos aleatórios

16



5.4.2 - Programação

```

void QuickSort
    (int i, int j, vetor V) {
    int pivot, pivind, k;
    pivind = Pivo(i, j, V);
    if (pivind != -1) {
        pivot = V[pivind];
        k = Particao(i, j, pivot, V);
        QuickSort(i, k-1, V);
        QuickSort(k, j, V);
    }
}
    
```

QuickSort atua entre os índices i e j do vetor V

Pivo retorna o índice do pivô no intervalo $[i, j]$ do vetor V ; retorna -1, caso o intervalo não tenha pivô (sem elementos distintos)

Particao divide o intervalo $[i, j]$ do vetor V nos sub-vetores baixo e alto, retornando o índice do 1º elemento no sub-vetor alto

17

Função Pivô: retorna -1, se o vetor no trecho considerado não possui elementos distintos; caso contrário retorna o índice do maior entre os dois elementos distintos mais à esquerda.

```

int Pivo (int i, int j, vetor V) {
    int prim, k, indpiv; int achou;
    prim = V[i]; achou = FALSE;
    k = i+1; indpiv = -1;
    while (! achou && k <= j)
        if (V[k] == prim) k++;
        else {
            achou = TRUE;
            if (V[k] > prim) indpiv = k;
            else indpiv = i;
        }
    return indpiv;
}
    
```

18

Função Partição: faz a permutação dos elementos do vetor e acha o ponto de partição nos dois subvetores:

```

int Particao (int i, int j, int piv; vetor V) {
  int left, right, aux;
  left = i; right = j;
  do {
    aux = V[left]; V[left] = V[right];
    V[right] = aux;
    while (V[left] < piv) left++;
    while (V[right] >= piv) right--;
  } while (left <= right);
  return left;
}
    
```

19

O MÉTODO HEAP-SORT

Definição de heap

- **Árvore binária completa:** tem **todos** os nós possíveis em cada nível; exemplos:

nível 0: 1 nó
 nível 1: 2 nós
 nível 2: 4 nós
 nível 3: 8 nós
 :
 nível i: 2ⁱ nós

Nº total de nós de uma árvore completa:

$$\sum_{i=0}^h 2^i \text{ nos} = 2^{h+1} - 1$$
 h: altura da árvore

Heap: árvore binária completa ou a árvore binária:

- Que é completa pelos menos até o seu penúltimo nível e
- Na qual as folhas do último nível estão o **mais à esquerda** possível.

Exemplos:

é heap não é heap é heap não é heap

- A estrutura de um heap é **única** para um número fixo de nós
- **Exemplo:** o heap abaixo com 18 nós é único

- Estrutura de dados para heap's: vetor de **n** inteiros
- **Declarações:**

```

typedef int vetor[200];
struct heap {vet elem; int n;};

heap H1, H2;
    
```

- **Observação:** os nós são os índices do vetor; exemplo a seguir

23

Pais e filhos dentro do vetor

Propriedades dessa representação:

nó i: i-ésima posição do vetor

nó i tem como filhos $\begin{cases} 2i + 1 \\ 2i + 2 \end{cases}$

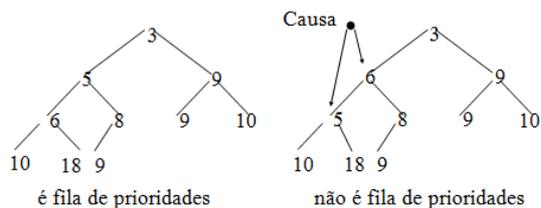
pai: $\lfloor (i-1) / 2 \rfloor$

altura de um heap de n nós: $\lfloor \log_2 n \rfloor$

5.5.2 - Fila de prioridades

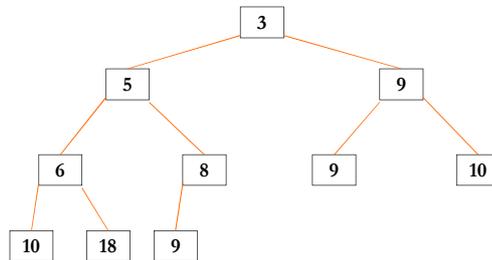
a) **Definição:** é um heap em que o valor de um nó é sempre menor ou igual aos de seus eventuais filhos

Exemplos:



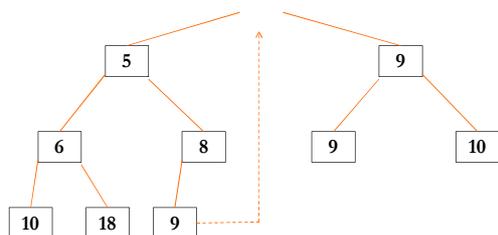
b) Operação de **deletar o elemento mínimo** numa fila de prioridades:

Inicialmente:



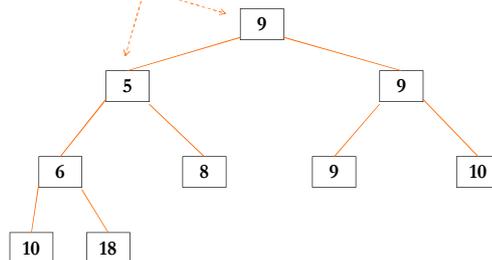
26

Retira-se a raiz e coloca-se ali o último elemento



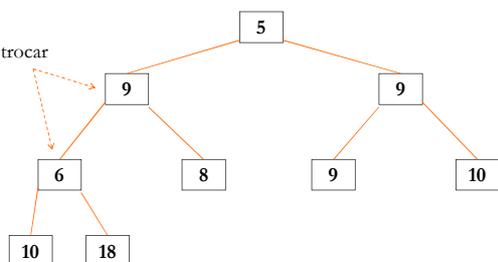
27

trocar



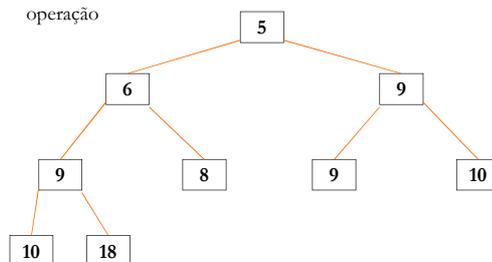
28

trocar

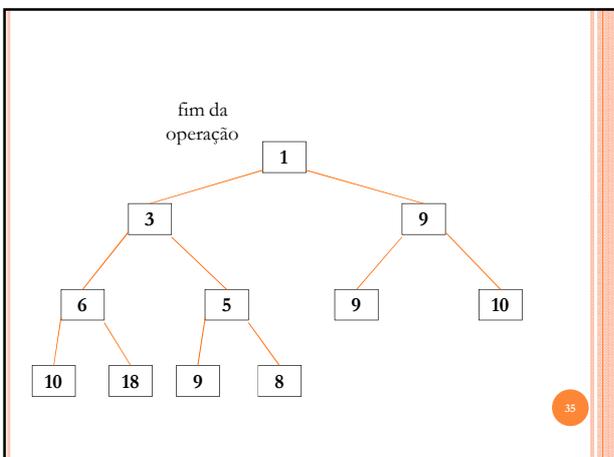
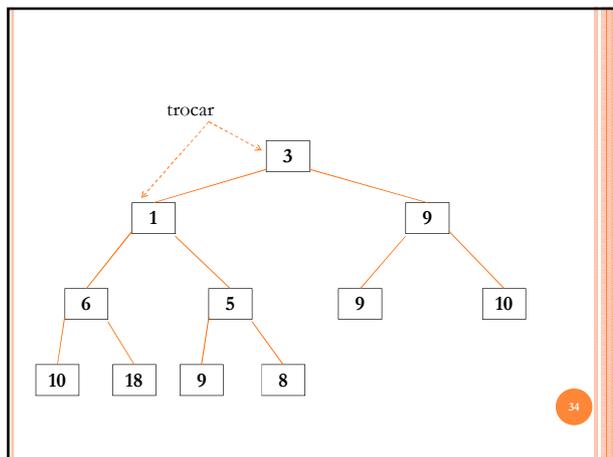
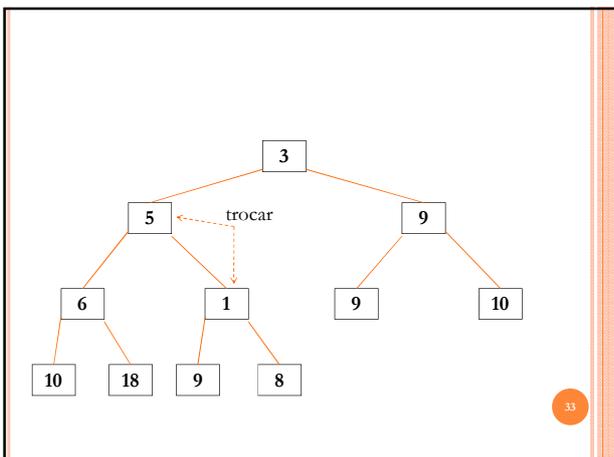
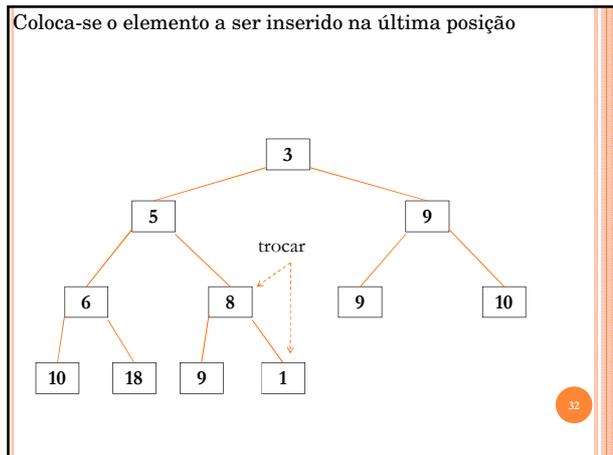
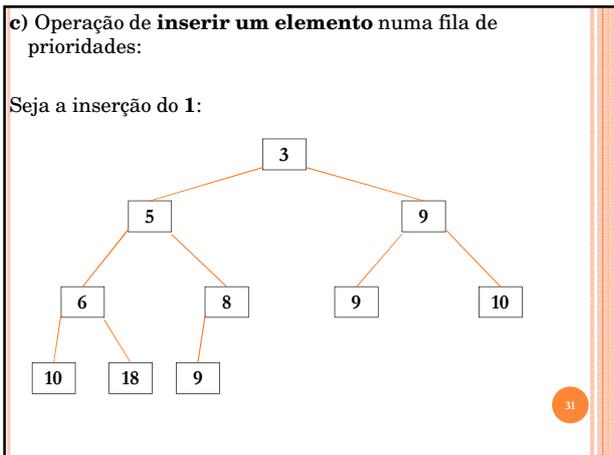


29

fim da operação



30



5.5.3 - Programação do Heap-Sort

- o O método consiste em duas partes principais:
 1. Inserir elemento por elemento do vetor numa fila de prioridades
 2. Retirar pela raiz todos os elementos da fila de prioridades, retornando-os ao vetor

36

```

void HeapSort (int n, vetor V) {
    int i; heap H;

    iniciar(&H);
    for (i = 0; i < n; i++)
        Inserir (V[i], &H);
    for (i = 0; i < n; i++)
        V[i] = DeletaMin (&H);
}

void iniciar (heap *H) {
    H->n = 0;
}

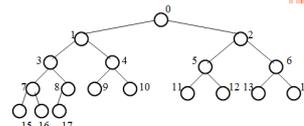
```

37

```

void Inserir (int x; heap *H) {
    int i, temp;
    if (H->n >= 200)
        printf("Insercao em heap cheio");
    else {
        (H->n)++; i = H->n - 1; H->elem[i] = x;
        while (i > 0 && H->elem[i] <
                H->elem[(i-1)/2]) {
            temp = H->elem[i];
            H->elem[i] = H->elem[(i-1)/2];
            H->elem[(i-1)/2] = temp; i = (i-1)/2;
        }
    }
}

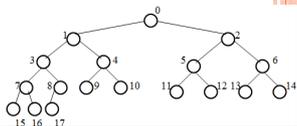
```



```

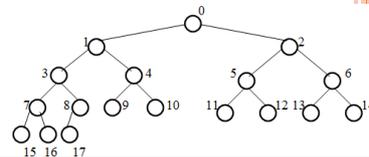
int DeletaMin (heap *H) {
    int i, j, temp, ret; int parou;
    if (H->n == 0) {
        printf ("Delecao em heap vazio"); return -1;
    }
    else {
        ret = H->elem[0]; H->elem[0] = H->elem[H->n - 1];
        (H->n)--; parou = FALSE; i = 0;
        while (! parou && 2*i+1 <= H->n - 1) {
            if (2*i+1 == H->n - 1 ||
                H->elem[2*i+1] < H->elem[2*i+2])
                j = 2*i+1;
            else j = 2*i+2;
            if (H->elem[i] > H->elem[j]) {
                temp = H->elem[i]; H->elem[i] = H->elem[j];
                H->elem[j] = temp; i = j;
            }
            else parou = TRUE;
        }
        return ret;
    }
}

```



Observações:

- O método Heap-Sort é $O(n \log n)$ no caso médio e no pior caso
 - Altura de um heap de n nós: $\lceil \log_2 n \rceil$
- No caso médio, Quick-Sort leva alguma vantagem sobre Heap-Sort
- Há um gasto maior de memória devido à variável H



O MÉTODO MERGE-SORT

- **Merge:** fundir num só dois vetores ordenados, mantendo a ordem.
- Mas, se inicialmente o vetor está **desordenado**, como fazer fusão?

41

Idéia recursiva: Caso o vetor tenha dois ou mais elementos:

- **Dividir** o vetor ao meio, obtendo a metade da esquerda e a metade da direita;
- **Ordenar** cada uma dessas metades pelo Merge-Sort;
- **Fundir** as duas metades ordenadas.

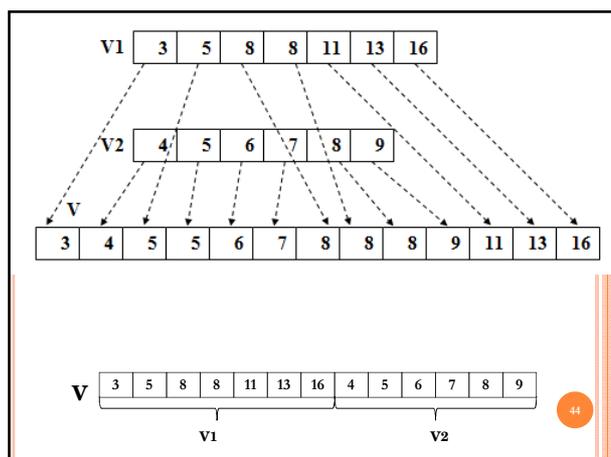
42

○ **Fundir dois vetores ordenados:** percorrê-los com **cursores**, comparando os elementos da posição dos mesmos; o menor é copiado num vetor temporário.

○ **Exemplo:** Fusão dos seguintes vetores:

V1: 3, 5, 8, 8, 11, 13, 16 e V2: 4, 5, 6, 7, 8, 9

43



44

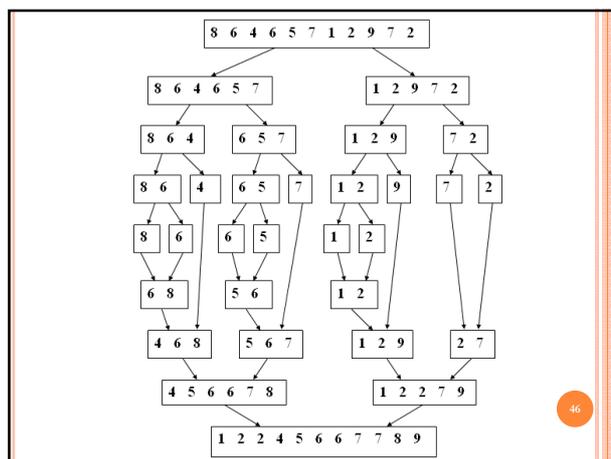
```
void MergeSort (int first,int last, vetor V) {
    int med;

    if (first < last) {
        med = (first + last)/2;
        MergeSort (first, med, V);
        MergeSort (med+1, last, V);
        Merge (first, last, V);
    }
}
```

○ A seguir, ilustração do método Merge-Sort para o vetor:

8, 6, 4, 6, 5, 7, 1, 2, 9, 7, 2

45



46

```
void Merge (int first, last; vetor V) {
    int med, i, j, k; vetor T;
    med = (first + last)/2;
    i = 0; j = first; k = med + 1;
    while (j <= med && k <= last) {
        if (V[j] < V[k]) {T[i] = V[j]; j++;}
        else {T[i] = V[k]; k++;}
        i++;
    }
    while (j <= med) {T[i] = V[j]; j++; i++;}
    while (k <= last) {T[i] = V[k]; k++; i++;}
    for (j = first, i = 0; j <= last; j++, i++)
        V[j] = T[i];
}
```

Merge é $O(n)$ e MergeSort é $O(n \log n)$

47