

# Classificação e Pesquisa de Dados

Aulas 08 e 09

Classificação de dados por Seleção: Seleção Direta e Heapsort

UFRGS

INF01124

1

## Classificação por Seleção

Caracteriza-se por identificar, a cada iteração, a chave de menor (maior) valor na porção do vetor ainda não ordenada e colocá-la em sua posição definitiva

Principais Algoritmos

- Seleção Direta
- Heapsort

2

## Seleção Direta

### Princípio de classificação

- ◆ A seleção da menor chave é feita por pesquisa seqüencial
- ◆ A menor chave encontrada é permutada com a que ocupa a posição inicial do vetor, que fica reduzido de um elemento
- ◆ O processo de seleção é repetido para o restante do vetor, até que todas as chaves alcancem suas posições definitivas

3

## Exercício

Suponha que se deseja classificar o seguinte vetor utilizando o método da seleção direta:

9 25 10 18 5 7 15 3

Simule as iterações necessárias para a classificação.

4

## Classificação por Seleção Direta Exemplo

Iteração	Vetor	Chave Selecionada	Permutação	Vetor ordenado até a posição
1	9 25 10 18 5 7 15 3	3	9 e 3	
2	3 25 10 18 5 7 15 9	5	25 e 5	1
3	3 5 10 18 25 7 15 9	7	10 e 7	2
4	3 5 7 18 25 10 15 9	9	18 e 9	3
5	3 5 7 9 25 10 15 18	10	25 e 10	4
6	3 5 7 9 10 25 15 18	15	25 e 15	5
7	3 5 7 9 10 15 25 18	18	25 e 18	6
8	3 5 7 9 10 15 18 25			8

5

## Classificação por Seleção Direta Procedimento

```
Proc seleção direta ( c, n );
begin
  for i ← 1 to n - 1 do
    begin
      min ← i; /* posição do mínimo inicial */
      for j ← i + 1 to n do
        if c[j] < c[min]
          then min ← j; /* posição do novo mínimo */
      /* troca */
      ch ← c[i];
      c[i] ← c[min];
      c[min] ← ch
    end
  end
end
```

6

## Classificação por Seleção Direta Análise de Desempenho

### ◆ Número de comparações efetuadas

1ª iteração: compara o 1º elemento com os n-1 demais: n-1

2ª iteração: compara o 2º elemento com os n-2 demais: n-2

3ª iteração: compara o 3º elemento com os n-3 demais: n-3

...

(n-1)ª iteração: compara o (n-1)º elemento com o último: 1

$$\begin{aligned}\text{Total de comparações} &= (n-1) + (n-2) + \dots + 1 \\ &= (n^2 - n) / 2 \\ &= O(n^2)\end{aligned}$$

7

## Heapsort

◆ Utiliza uma estrutura de dados (heap) para organizar informação durante a execução do algoritmo

◆ **Heap**: vetor que pode ser visto como uma árvore binária totalmente preenchida em todos os níveis exceto, possivelmente, o último

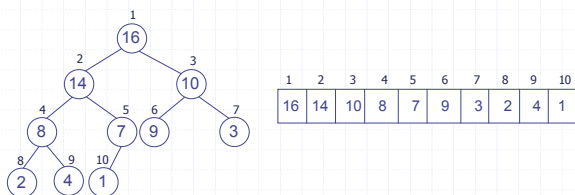
◆ O último nível é preenchido da esquerda para a direita até um certo ponto

◆ O vetor  $A$  que representa um heap possui dois atributos: tamanho ( $\text{length}[A]$ ) e número de elementos no vetor ( $\text{heap\_size}[A]$ )

8

## Heap

◆ Um heap visto como uma árvore binária ou como um vetor



◆ Operações de consulta sobre um nó  $i$

Pai( $i$ )  
**return**( $\lfloor i/2 \rfloor$ )

Esquerda( $i$ )  
**return**( $2*i$ )

Direita( $i$ )  
**return**( $2*i + 1$ )

9

## Propriedade do Heap

◆ O valor de um nó é sempre menor ou igual ao valor de seu nó pai

$$A[\text{Pai}(i)] \geq A[i], \quad \forall i \leq \text{heap\_size}[A]$$

■ O elemento de maior valor encontra-se armazenado na raiz da árvore

10

## Definições

◆ A altura de um nó em uma árvore corresponde ao número de arestas no caminho descendente mais longo daquele nó até um nó folha

◆ A altura de um heap de  $n$  elementos é  $\Theta(\log_2 n)$  – baseado em uma árvore binária completa

◆ As operações básicas sobre heaps executam em tempo no máximo proporcional a altura da árvore e, portanto,  $O(\log_2 n)$

■ Porque a notação  $O$ , ao invés de  $\Theta$ , foi utilizada na expressão acima?

11

## Procedimentos sobre Heaps

◆ **Heapify**

■ Garante a manutenção da propriedade do Heap. Executa em  $O(\log_2 n)$

◆ **Build-Heap**

■ Produz um heap a partir de um vetor não ordenado. Executa em  $O(n)$

◆ **Heapsort**

■ Procedimento de ordenação local. Executa em  $O(n \log_2 n)$

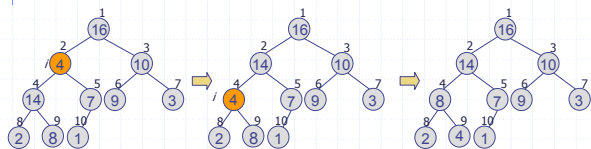
◆ **Extract-Max e Insert**

■ Permitem utilizar um heap para implementar uma fila de prioridades. Ambos executam em  $O(\log_2 n)$

12

## Procedimento Heapify

- ◆ Reorganiza heaps
- ◆ Assume que as árvores binárias correspondentes a *Esquerda(i)* e *Direita(i)* são heaps, mas  $A[i]$  pode ser menor que seus filhos
- ◆ Exemplo:



13

## Procedimento Heapify

```

Proc heapify ( A, i )
begin
  e ← Esquerda(i);
  d ← Direita(i);
  maior ← i;
  if (e ≤ heap_size[A] and A[e] > A[maior]) then
    maior ← e; /* filho da esquerda é maior */
  if (d ≤ heap_size[A] and A[d] > A[maior]) then
    maior ← d; /* filho da direita é maior */
  if (maior ≠ i) then
    begin
      exchange(A[i] ↔ A[maior]);
      heapify(A, maior);
    end
  end
end

```

Custo:  $O(\log_2 n)$  – cada troca tem custo  $\Theta(1)$  e ocorrem no máximo  $\log_2 n$  trocas

14

## Procedimento Build-Heap

- ◆ Utiliza o procedimento Heapify de forma *bottom-up* para transformar um vetor  $A[1..n]$  em um heap com  $n$  elementos
- ◆  $A[(\lfloor n/2 \rfloor + 1)]$  a  $A[n]$  correspondem às folhas da árvore e portanto são heaps de um elemento
- ◆ Basta chamar Heapify para os demais elementos do vetor  $A$

```

Proc build-heap ( A )
begin
  heap_size[A] ← length[A];
  for i ← ⌊length[A]/2⌋ downto 1 do
    heapify(A, i);
  end
end

```

15

## Exercícios

1. Utilize o procedimento Build-Heap para construir um heap a partir do vetor  
4 1 3 2 16 9 10 14 8 7
2. Por que no laço do procedimento Build-Heap fazemos o valor da variável "i" variar de  $\lfloor \text{length}[A]/2 \rfloor$  até 1 ao invés de de 1 até  $\lfloor \text{length}[A]/2 \rfloor$ ?

16

## Procedimento Heapsort

- ◆ Constrói um heap a partir de um vetor de entrada
- ◆ Como o maior elemento está localizado na raiz ( $A[1]$ ), este pode ser colocado em sua posição final, trocando-o pelo elemento  $A[n]$
- ◆ Reduz o tamanho do heap de uma unidade, chama Heapify( $A, 1$ ) e repete o passo anterior até que o heap tenha tamanho = 2

17

## Procedimento Heapsort

```

Proc heapsort (A)
begin
  build_heap(A);
  for i ← length[A] downto 2 do
    begin
      exchange(A[i] ↔ A[1]);
      heap_size[A] ← heap_size[A] - 1;
      heapify(A, 1);
    end
  end
end

```

Custo:  $O(n \log_2 n)$  – build\_heap custa  $O(n)$  e cada uma das  $n-1$  chamadas a heapify custa  $O(\log_2 n)$

18

## Uso de Heap

◆ **Fila de Prioridades** – estrutura de dados para manutenção de um conjunto com  $S$  elementos, cada um valor de chave, e que suporta as seguintes operações:

- **Inserere\_Heap(S,x)**: Insere o elemento  $x$  no conjunto  $S$
- **Máximo(S)**: Retorna o elemento de  $S$  com maior valor de chave
- **Extrai\_Max(S)**: Remove de  $S$  e retorna o elemento com o maior valor de chave

19

## Procedimento Inserere\_Heap

```
Proc inserere_heap (A, chave)
begin
  heap_size[A] ← heap_size[A] + 1;
  i ← heap_size[A];
  while (i > 1 and A[Pai(i)] < chave) do
    begin
      A[i] ← A[Pai(i)];
      i ← Pai(i);
    end
  end
  A[i] ← chave;
end
```

Custo:  $O(\log_2 n)$

20

## Procedimento Extrai\_Max

```
Proc extrai_max (A)
begin
  if (heap_size[A] < 1) then
    error ("heap underflow");
  max ← A[1];
  A[1] ← A[heap_size[A]];
  heap_size[A] ← heap_size[A] - 1;
  heapify(A, 1);
  return (max);
end
```

Custo:  $O(\log_2 n)$

21