CI1056: Algoritmos e Estruturas de Dados II

Prof. Dr. Marcos Castilho

Departamento de Informática/UFPR

4 de dezembro de 2020

Resumo

O algoritmo de ordenação *heapsort*

Objetivos da aula

- Apresentar o algoritmo de ordenação por *heap*: o *heapsort*
- Discutir a complexidade do número de comparações

Motivação

- Qual o motivo de apresentarmos um outro algoritmo de ordenação?
- O *mergesort* e o *quicksort* são extremamente eficientes
- O problema de ordenação é extremamente importante em Computação e é estudado até hoje!
- O *heapsort* também é extremamente eficiente e é também elegante!

A ideia do *heapsort*

- Sabemos que construir um *heap* de máximo tem custo linear
- Sabemos que em um *heap* de máximo o maior elemento do vetor está na primeira posição
- A ideia é trocar os elementos da primeira e da última posições
- Isto resultará em um vetor que não é um *heap* de máximo
- Basta considerar as primeiras $n - 1$ posições e reconstruir o *heap*, usando para isto o *max_heapify*, a custo logaritmico!

heapsort: ordenação por *heap*

- Instância: (v, n), onde v é um vetor qualquer e n é o tamanho do vetor $v[1..n]$
- Resposta: retorna (v, n) de forma que v é um vetor ordenado em ordem não decrescente, isto é: $v[i] \leq v[i+1]$ para $i \in [1..(n-1)]$

O algoritmo *heapsort*

```
heapsort (v, n)
    build_max_heap (v, n)
    para i de n ate 2, regressivamente faca
        trocar v[1] com v[i]
        n = n - 1
        max_heapify (v, 1)
```

Exemplo

- Considere o seguinte vetor como exemplo de entrada:

	1	2	3	4	5	6	7	8	9	10
v	15	35	40	10	20	5	45	70	25	90

- Com um custo linear no número de comparações obtemos um *heap* de máximo:

	1	2	3	4	5	6	7	8	9	10
v	90	70	45	25	35	5	40	10	15	20

Exemplo

- Neste *heap* de máximo, o primeiro elemento é o maior

	1	2	3	4	5	6	7	8	9	10
v	90	70	45	25	35	5	40	10	15	20

- Trocamos o primeiro com o último
- Diminuimos o tamanho do vetor (`n = n - 1`)

	1	2	3	4	5	6	7	8	9	10
v	20	70	45	25	35	5	40	10	15	90

Exemplo

- Este *heap* reduzido não é de máximo

	1	2	3	4	5	6	7	8	9	10
v	20	70	45	25	35	5	40	10	15	90

- Aplicamos `max_heapify (v, 1)`:

	1	2	3	4	5	6	7	8	9	10
v	70	35	45	25	20	5	40	10	15	90

Exemplo

- Agora vamos trocar o primeiro com o penúltimo e diminuir o vetor:

	1	2	3	4	5	6	7	8	9	10
v	15	35	45	25	20	5	40	10	70	90

- Aplicamos `max_heapify (v, 1)`:

	1	2	3	4	5	6	7	8	9	10
v	45	35	40	25	20	5	15	10	70	90

Exemplo

- Agora vamos trocar o primeiro com o antepenúltimo e diminuir o vetor:

	1	2	3	4	5	6	7	8	9	10
v	10	35	40	25	20	5	15	45	70	90

- Aplicamos `max_heapify (v, 1)`:

	1	2	3	4	5	6	7	8	9	10
v	40	35	15	25	20	5	10	45	70	90

Exemplo

- Trocamos 1 com 7 e diminuimos o vetor:

	1	2	3	4	5	6	7	8	9	10
v	10	35	15	25	20	5	40	45	70	90

- Aplicamos `max_heapify (v, 1)`:

	1	2	3	4	5	6	7	8	9	10
v	35	25	15	10	20	5	40	45	70	90

Exemplo

- Trocamos 1 com 6 e diminuimos o vetor:

	1	2	3	4	5	6	7	8	9	10
v	5	25	15	10	20	35	40	45	70	90

- Aplicamos `max_heapify (v, 1)`:

	1	2	3	4	5	6	7	8	9	10
v	25	20	15	10	5	35	40	45	70	90

Exemplo

- Trocamos 1 com 5 e diminuimos o vetor:

	1	2	3	4	5	6	7	8	9	10
v	5	20	15	10	25	35	40	45	70	90

- Aplicamos `max_heapify (v, 1)`:

	1	2	3	4	5	6	7	8	9	10
v	20	10	15	5	25	35	40	45	70	90

Exemplo

- Trocamos 1 com 4 e diminuimos o vetor:

	1	2	3	4	5	6	7	8	9	10
v	5	10	15	20	25	35	40	45	70	90

- Aplicamos `max_heapify (v, 1)`:

	1	2	3	4	5	6	7	8	9	10
v	15	10	5	20	25	35	40	45	70	90

Exemplo

- Trocamos 1 com 3 e diminuimos o vetor:

	1	2	3	4	5	6	7	8	9	10
v	5	10	15	20	25	35	40	45	70	90

- Aplicamos `max_heapify (v, 1)`:

	1	2	3	4	5	6	7	8	9	10
v	10	5	15	20	25	35	40	45	70	90

Exemplo

- Na última etapa trocamos 1 com 2 e diminuimos o vetor:

	1	2	3	4	5	6	7	8	9	10
v	5	10	15	20	25	35	40	45	70	90

- Aplicamos `max_heapify (v, 1)` e o vetor estará ordenado:

	1	2	3	4	5	6	7	8	9	10
v	5	10	15	20	25	35	40	45	70	90

Análise da complexidade

- O número de comparações entre os elementos do vetor pode ser assim discutido (sem entrar muito em detalhes, ignorando os pisos e tetos)
  - O laço executa $(n-1)$ vezes
  - A cada iteração o `max_heapify` é chamado a um custo $log_2(i)$, *no pior caso*
    - Na primeira iteração: $log_2(n-1)$
    - Na segunda iteração: $log_2(n-2)$
    - Na terceira iteração: $log_2(n-3)$
    - assim por diante...
    - Na última iteração: $log_2(2)$
- Isto resulta em um custo aproximado de pior caso de $n.log_2(n)$
- Em Análise de Algoritmos este estudo pode ser melhor visto

Fim do tópico

- O conteúdo desta aula está no livro Cormen, Leiserson, Rivest e Stein, no capítulo 6, seção 6.4

Licença

- Slides feitos em LaTeX usando beamer
- Licença

  Creative Commons Atribuição-Uso Não-Comercial-Vedada a Criação de Obras Derivadas 2.5 Brasil License.`http://creativecommons.org/licenses/by-nc-nd/2.5/br/`

  Creative Commons Atribuição-Uso Não-Comercial-Vedada a Criação de Obras Derivadas 2.5 Brasil License.`http://creativecommons.org/licenses/by-nc-nd/2.5/br/`