HEAPS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

Agenda

1 Introdução

2 Heapsort

3 Bibliografia

Introdução

Motivação: filas de prioridade

- Escalonamento de processos
- Gerenciamento de tráfego em redes
- Algoritmos como Prim, Dijkstra, etc.
- Ordenação (heapsort)

ADT: fila de prioridade

- E $find\_max(heap\ h)$
- E $remove\_max(heap\ h)$
- $void\ add(heap\ h, Key\ K, Element\ E)$

Introdução

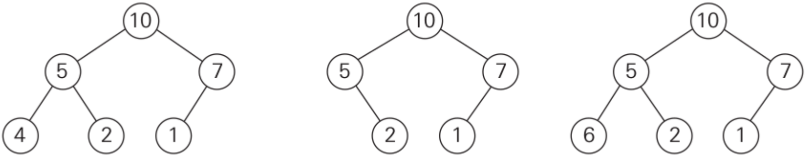
Implementações:

- Fila: busca $\Theta(n)$
- Lista (ordenada ou não-ordenada): $\Theta(n)$ para inserir ou remover
- BST: n inserções/remoções, na média $\Theta(n \log n)$
- AVL: eficiência espacial

Heaps: uma árvore binária com as seguintes propriedades

- Formato (shape property): é uma árvore binária completa (i.e., todos os níveis estão preenchidos, exceto possivelmente o último; nós folhas faltando mais à direita)
- Dominância do pai (parental dominance): a chave em cada nó é
  (a) maior ou igual (max-heap) do que seus filhos
  (b) menor ou igual (min-heap) do que seus filhos

Introdução



[1]

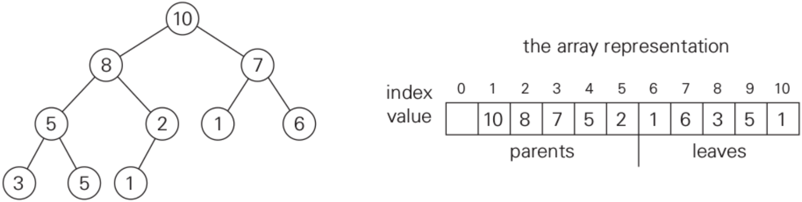
Qualquer sequência de valores da raiz para uma folha é decrescente (ou não-crescente, na presença de chaves repetidas)

BST: ordem total

Heap: ordem parcial

[1] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Introdução



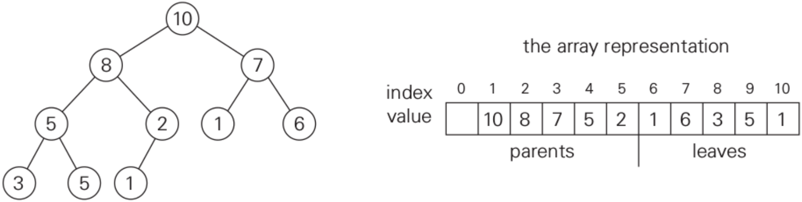
[2]

Propriedades:

- Altura de uma heap com n nós = $\lfloor log_2 n \rfloor$
- Maior elemento: raiz (em uma max-heap)
- Toda sub-árvore é uma heap
- Implementação eficiente com arrays
  - Primeiro elemento no índice 1

[2] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Introdução



[3]

Implementação eficiente com arrays

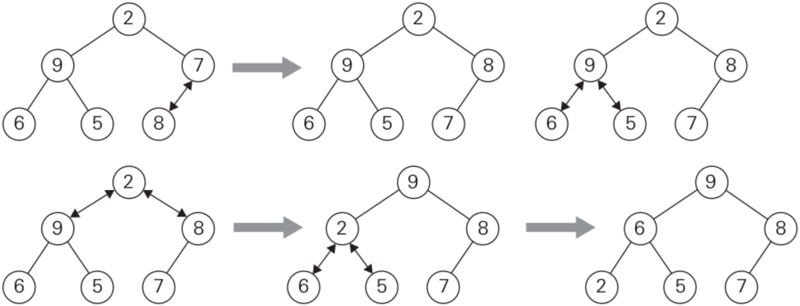
- Nós internos: as primeiras $\lfloor n/2 \rfloor$ posições
  - Índices: $1 \leq i \leq \lfloor n/2 \rfloor$
  - Filhos nas posições $2i$ e $2i + 1$ (se valor $\leq n$)
- Nós folhas: as últimas $\lceil n/2 \rceil$ posições
  - Índices: $2 \leq i \leq n$
  - Pai na posição $\lfloor i/2 \rfloor$

[3] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Introdução

Inserção (bottom-up): 2, 9, 7, 6, 5, 8

- Insere todos, depois heapify do último para primeiro nó interno



4

Construção de uma heap com tamanho n: $\Theta(n)$

[4] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Introdução

Construção de uma heap bottom-up

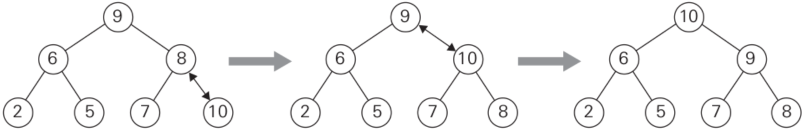
Algoritmo: void HeapBottomUp(H [1..n])

```
for i ← ⌊n/2⌋ downto 1 do
    k ← i;
    v ← H[k];
    heap ← false;
    while ¬ heap ∧ 2 ∗ k ≤ n do
        j ← 2 ∗ k;
        if j < n then
            // possui dois filhos
            if H[j] < H[j + 1] then j ← j + 1;
        if v ≥ H[j] then
            heap ← true;
        else
            H[k] ← H[j];
            k ← j;
    H[k] ← v;
```

Introdução

Inserção (top-down): 2, 9, 7, 6, 5, 8

- Insere um-por-um, heapify logo após cada inserção



[5]

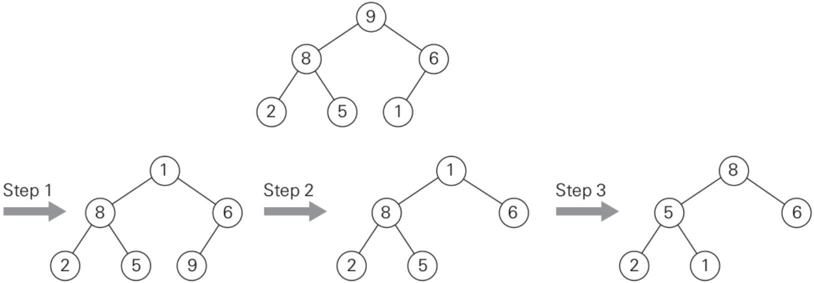
Construção de uma heap com tamanho *n*: $\Theta(n \log n)$

[5] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Introdução

Remoção (da raiz)

1. Troque a raiz com o último elemento da heap. Vá para o passo 2.
2. Decremente o tamanho da heap em 1. Vá para o passo 3.
3. Faça um heapify da nova raiz como na abordagem bottom-up.



[6]

Remoção da raiz em uma heap com tamanho n: $\Theta(log\ n)$

[6] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Agenda

1 Introdução

2 Heapsort

3 Bibliografia

Heapsort

Algoritmo de ordenação baseado em heap

1. Construa uma heap para o array de entrada. Vá para o passo 2.
2. Aplique o algoritmo de remoção da raiz $n - 1$ vezes.

Observe que os elementos são removidos do maior para o menor (max-heap) ou do menor para o maior (min-heap).

O array estará ordenado de forma crescente (max-heap) ou decrescente (min-heap)

- Na presença de chaves repetidas, não-decrescente (max-heap) ou não-crescente (min-heap)

Heapsort

Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 **7** 6 5 8	**9** 6 8 2 5 7
2 **9** 8 6 5 7	7 6 8 2 5 | **9**
2 9 8 6 5 7	**8** 6 7 2 5
9 **2** 8 6 5 7	5 6 7 2 | **8**
9 6 8 2 5 7	**7** 6 5 2
	2 6 5 | **7**
	6 2 5
	5 2 | **6**
	5 2
	2 | **5**
	2

[7]

[7] Fonte: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Custo computacional

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

[8]

Mesma classe do Mergesort, com melhor eficiência espacial

Em geral, menos eficiente que o Quicksort

Achar os k maiores elementos: $\Theta(n + k \ log \ n)$

[8] http://bigocheatsheet.com/

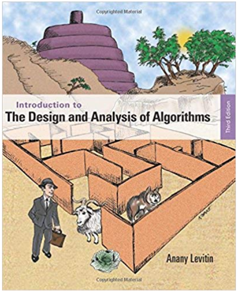
Agenda

1 Introdução

2 Heapsort

3 Bibliografia

Bibliografia + leitura recomendada



Capítulo 6 (pp. 226–232)
Anany Levitin.
Introduction to the Design and Analysis of Algorithms.
3a edição. Pearson. 2011.

Capítulo 5 (pp. 170–177)
Capítulo 7 (pp. 243–244)
Clifford Shaffer.
Data Structures and Algorithm Analysis.
Dover, 2013.

HEAPS

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil