Heap Binário e HeapSort

André Vignatti

Estrutura de Dados

Estrutura de dados: forma de organizar dados para manipulação eficiente

- estrutura de dados fornece funções para manipulação (API)
  - ex: inserção, remoção, busca, ...
- objetivo: eficiência
  - sabedoria comum: não dá pra ser eficiente em todas as operações, por isso existem diversas estruturas de dados (adequada para diversas situações)

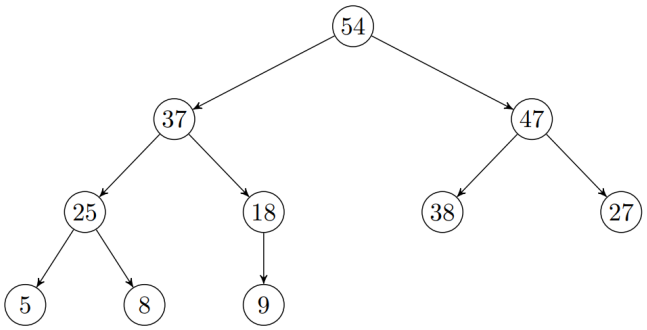
Heap

Heap: estrutura de dados especializada na obtenção do maior (menor) elemento.

- heaps estruturam os dados usando árvores
- há vários heaps diferentes, veremos o chamado **heap binário**
- heap binário usa árvores binárias quase completas

Propriedade de heap : $v[\text{pai}(i)] \geq v[i]$, para todo nodo i, exceto a raiz.

Exemplo:



Arrumando o Heap

Problema: Arrumar Heap

Instância: (v, i, n), onde

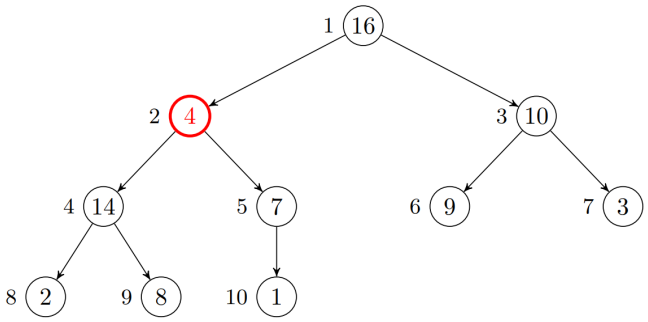
- $v[1..n]$ é um vetor que representa uma árvore binária quase completa
- $1 \leq i \leq n$
- as sub-árvores enraizadas em direita(i) e esquerda(i) são heaps

Resposta: sub-árvore enraizada em i é heap.

max-heapify(v, i, n)

esq $\leftarrow$ esquerda(i)
dir $\leftarrow$ direita(i)
se $esq \leq n$ **e** $v[esq] > v[i]$ **então**
    maior $\leftarrow$ esq
senão
    maior $\leftarrow i$
se $dir \leq n$ **e** $v[dir] > v[maior]$ **então**
    maior $\leftarrow$ dir
se $maior \neq i$ **então**
    troca($v[i]$, v[maior])
    max-heapify(v, maior, n)

Exemplo: executar max-heapify$(v, 2)$



Análise de Pior Caso

No pior caso, sempre temos que maior $\neq i$, e as chamadas recursivas só terminam quando chegamos em uma folha.

$C(k)$: número de comparações com elementos do vetor para nó com altura k.

$$C(k) = \begin{cases} 0, & \text{se } k = 0 \\ 2 + C(k-1), & \text{se } k > 0 \end{cases}$$

Resolvendo a recorrência, temos que $C(k) = 2k$.

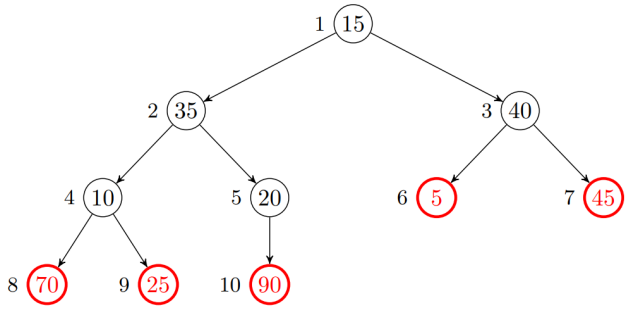
Construindo um Heap

Problema: Construir Heap
Instância: (v, n) onde v é um vetor indexado de 1 a n. **Resposta:** os elementos de v reorganizados como heap.

build-max-heap(v, n)
para $i \leftarrow \lfloor n/2 \rfloor$ **até** 1 **faça** max-heapify(v, i)

Exemplo:

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



(fazer execução no quadro)

Analise de Pior Caso

Teorema. *`build-max-heap` faz no máximo $4n$ comparações entre elem. do vetor.*

Demonstração. A quantidade de nós no nível i é 2^i e, para cada nó, a execução de `max-heapify` custa $2(h - i)$ no pior caso.

Somando todos os níveis:

$$\begin{aligned} C(n) &= 1 \cdot 2h + 2 \cdot 2(h-1) + 4 \cdot 2(h-2) + \ldots + 2^{h-1} \cdot 2 \cdot 1 + 2^h \cdot 2 \cdot 0 \\ &= 2\left(1 \cdot h + 2 \cdot (h-1) + 4 \cdot (h-2) + \ldots + 2^{h-1} \cdot 2 \cdot 1 + 2^h \cdot 2 \cdot 0\right) \\ &= 2 \sum_{i=0}^{h} 2^i \cdot (h-i) \\ &= 2 \left(h \sum_{i=0}^{h} 2^i - \sum_{i=0}^{h} 2^i \cdot i \right). \end{aligned}$$

De aulas passadas, sabemos que

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1 \qquad \text{e} \qquad \sum_{i=0}^{h} 2^i \cdot i = h2^{h+2} - (h+1)2^{h+1} + 2.$$

Assim, substituindo,

$$\begin{aligned} C(n) &= 2\left(h(2^{h+1}-1) - (h2^{h+2} - (h+1)2^{h+1} + 2)\right) \\ &= 2\left(h2^{h+1} - h - h2^{h+2} + h2^{h+1} + 2^{h+1} - 2\right) \\ &= 2\left(h2^{h+2} - h2^{h+2} - h + 2^{h+1} - 2\right) \\ &\leq 2^{h+2} \\ &= 4 \cdot 2^h \end{aligned}$$

Lembrando (aulas passadas), $h = \lfloor \log_2 n \rfloor$. Assim,

$$C(n) = 4 \cdot 2^{\lfloor \log_2 n \rfloor} \leq 4 \cdot 2^{\log_2 n} = 4n.$$

□

Heapsort

O algoritmo Heapsort serve para ordenar um vetor.

```
heapsort(v, n)
    build-max-heap(v, n)
    para i ← n até 2 faça
        troca(v[1], v[i])
        max-heapify(v, 1, i)
```

Exemplo

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

Após build-max-heap, 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

(executar no quadro a partir daqui)

Análise de Pior Caso

Em vetor de tamanho i (árvore binária quase completa), a maior altura é $\lfloor \log_2 i \rfloor$.
Portanto, `max-heapify` em vetor de tamanho i custa $2\lfloor \log_2 i \rfloor$ no pior caso.
Somando todas iterações,

$$\sum_{i=2}^{n} 2\lfloor \log_2 i \rfloor \leq \sum_{i=1}^{n} 2 \log_2 n = 2n \log_2 n.$$

Finalmente, somando com o custo do `build-max-heap`

$$C(n) \leq 2n \log_2 n + 4n.$$