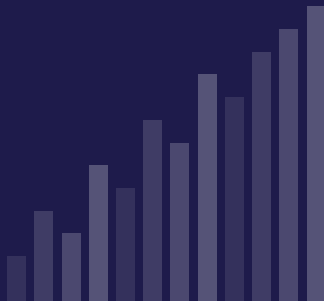

AULA 8 · BLOCO 1: FUNDAMENTOS, ORDENAÇÃO E CUSTO

Heaps e heapsort

Capítulo 6 (seções 6.1 a 6.4)

Estruturas de dados e seus algoritmos · José Dihego · IFBA



Roteiro da aula

Ao final, você será capaz de

- ▶ Representar uma árvore binária quase completa em um vetor
- ▶ Manter a propriedade de heap com MAX-Heapify
- ▶ Mostrar que BUILD-MAX-Heap é $O(n)$, não $O(n \lg n)$
- ▶ Medir comparações: $\approx 2n$ na construção, $\approx 2n \lg n$ no heapsort



Teoria

Heap em vetor, Max-Heapify, construção linear, heapsort



Prática em C

Heapsort em C com índices a partir de 0



Experimento

Contar comparações da construção e da ordenação



Conexões

Sistemas operacionais, simulação, embarcados e jogos

Um heap é uma árvore dentro de um vetor

- ▶ Árvore binária **quase completa**: todos os níveis cheios, exceto talvez o último, preenchido da esquerda.
- ▶ Pai de i : $\lfloor i/2 \rfloor$; filhos: $2i$ e $2i + 1$. Sem ponteiros.
- ▶ **Propriedade de heap de máximo**: $A[\text{pai}(i)] \geq A[i]$. A raiz é o maior.
- ▶ Altura: $\lfloor \lg n \rfloor$.

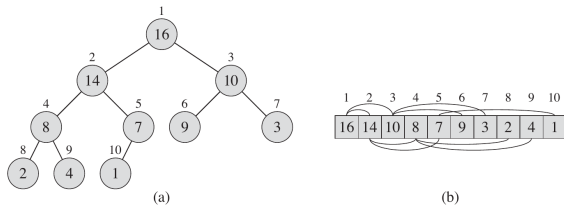


Figura 6.1 de Cormen et al. (2009) — (a) heap de máximo com 10 elementos; (b) o mesmo heap no vetor.

Mantendo a propriedade

Supõe que as subárvores de i já são heaps. O elemento $A[i]$ "desce" trocando com o maior filho.

Custo: $O(h) = O(\lg n)$. Pela recorrência,
 $T(n) \leq T(2n/3) + \Theta(1)$.

Max-Heapify(A, i)

```

1  $l \leftarrow 2i; \quad r \leftarrow 2i + 1$ 
2 se  $l \leq A.tamanho$  e  $A[l] > A[i]$  então
3    $maior \leftarrow l$ 
4 senão
5    $maior \leftarrow i$ 
6 se  $r \leq A.tamanho$  e  $A[r] > A[maior]$  então
7    $maior \leftarrow r$ 
8 se  $maior \neq i$  então
9   troque  $A[i]$  com  $A[maior]$ 
10  Max-Heapify( $A, maior$ )
  
```

Mantendo a propriedade (exemplo)

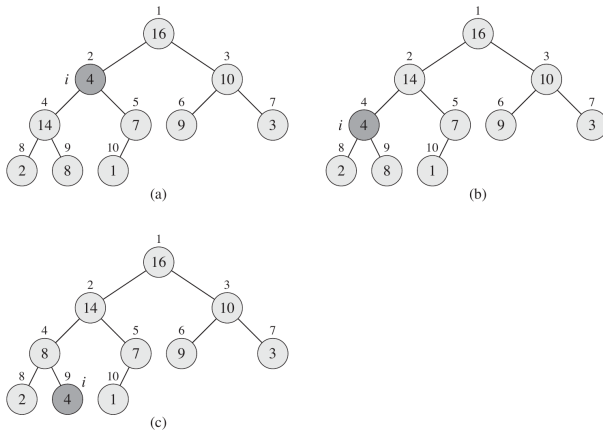


Figura 6.2 de Cormen et al. (2009) — $\text{Max-Heapify}(A, 2)$: o 4 desce até o lugar certo

Construir um heap custa $O(n)$

BUILD-MAX-Heap: para $i = \lfloor n/2 \rfloor$ **descendo**
até 1, Max-Heapify(A, i).

Análise ingênua: $n/2$ chamadas $\times O(\lg n) = O(n \lg n)$. Mas a maioria dos nós é baixa:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h \geq 0} \frac{h}{2^h}\right) = O(2n) = O(n)$$

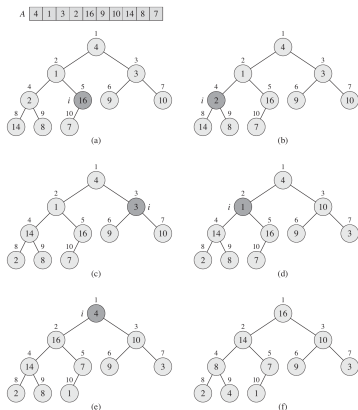


Figura 6.3 de Cormen et al. (2009) — BUILD-MAX-Heap

Heapsort

$O(n) + (n - 1) \cdot O(\lg n) = O(n \lg n)$, **no lugar** (sem vetor auxiliar), mas **não estável**.

HEAPSORT(A)

- 1 BUILD-MAX-Heap(A)
- 2 **para** $i \leftarrow A.\text{comprimento}$ **descendo até 2 faça**
- 3 troque $A[1]$ com $A[i]$
- 4 $A.\text{tamanho} \leftarrow A.\text{tamanho} - 1$
- 5 MAX-Heapify($A, 1$)

Heapsort (exemplo)

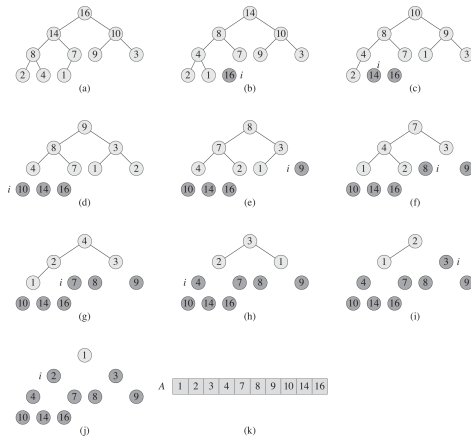


Figura 6.4 de Cormen et al. (2009) — o máximo vai para o fim: o heap encolhe



Prática em C

- ▶ Índices de 0: filhos em $2i + 1$ e $2i + 2$
- ▶ Max-Heapify recursivo, construção e ordenação
- ▶ Teste com o heap da Figura 6.1

Heapsort em C (1/3)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Heap em vetor com índices 0..n-1: filhos de i em 2i+1 e 2i+2
5     (no livro, com índices 1..n: 2i e 2i+1). */
6  #define ESQ(i) (2 * (i) + 1)
7  #define DIR(i) (2 * (i) + 2)
8
9  void troca(int A[], int i, int j) { int t = A[i]; A[i] = A[j]; A[j] = t; }
10
11 /* MAX-HEAPIFY: supõe que as subárvores de i já são heaps e faz i "descer". */
12 void max_heapify(int A[], int tam, int i) {
13     int maior = i;
14     if (ESQ(i) < tam && A[ESQ(i)] > A[maior]) maior = ESQ(i);
15     if (DIR(i) < tam && A[DIR(i)] > A[maior]) maior = DIR(i);
```

Heapsort em C (2/3)

```
16     if (maior != i) {
17         troca(A, i, maior);
18         max_heapify(A, tam, maior);
19     }
20 }
21
22 void constroi_heap(int A[], int n) {           // BUILD-MAX-HEAP: das folhas para a raiz
23     for (int i = n / 2 - 1; i >= 0; i--) max_heapify(A, n, i);
24 }
25
26 void heapsort(int A[], int n) {
27     constroi_heap(A, n);
28     for (int fim = n - 1; fim > 0; fim--) {
29         troca(A, 0, fim);                       // o máximo vai para o fim
```

Heapsort em C (3/3)

```
30         max_heapify(A, fim, 0);                // o heap encolhe uma posição
31     }
32 }
33
34 int main(void) {
35     int n;
36     if (scanf("%d", &n) != 1 || n <= 0) return 1;
37     int *A = malloc(n * sizeof(int));
38     for (int k = 0; k < n; k++)
39         if (scanf("%d", &A[k]) != 1) return 1;
40     heapsort(A, n);
41     for (int k = 0; k < n; k++) printf("%d%c", A[k], k < n - 1 ? ' ' : '\n');
42     free(A);
43     return 0;
44 }
```



Experimento

- ▶ Construção: $\leq 2n$ comparações
- ▶ Heapsort completo: $\approx 2n \lg n$
- ▶ $n = 2^{12}$ a 2^{21} , média de 3 vetores

Construção linear, ordenação $n \lg n$

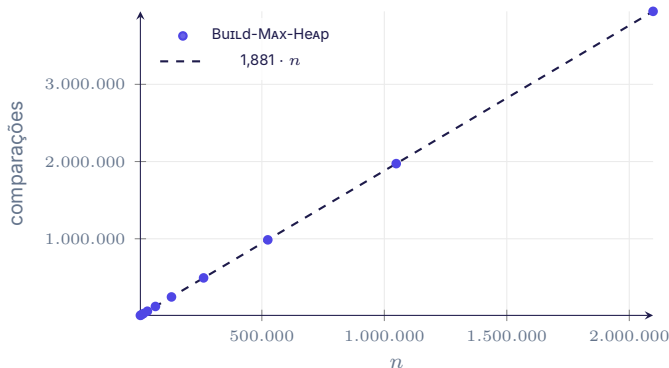
Hipótese. MAX-Heapify faz até 2 comparações por nível descido. Como $\sum h/2^h = 2$, a **construção** faz no máximo cerca de $2n$ comparações: linear. Cada uma das $n - 1$ extrações desce até $\lg n$ níveis: o heapsort faz cerca de $2n \lg n$ comparações.

Método

1. MAX-Heapify iterativo com contador de comparações
2. $n = 2^{12}, \dots, 2^{21}$; média de 3 vetores aleatórios
3. Construção e ordenação medidas separadamente
4. Ajuste de $c \cdot n$ e $c \cdot n \lg n$

🧪 EXPERIMENTO

Construir o heap é linear



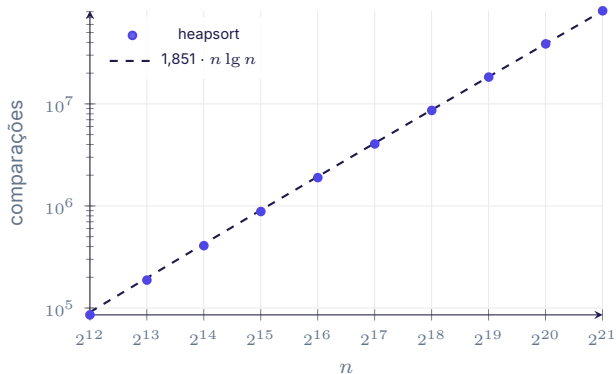
n	medido	razão
4.096	7.672	1,873
16.384	30.792	1,879
65.536	123.280	1,881
262.144	493.275	1,882
1.048.576	1.973.082	1,882
2.097.152	3.945.408	1,881

razão = medido $\div$ n

✓ desvio máx. 0,0%

A constante fica perto de 1,88: abaixo do limite 2 da análise, porque nem todo nó desce até as folhas.

Heapsort completo



n	medido	razão
4.096	85.666	1,743
16.384	408.183	1,780
65.536	1.894.642	1,807
262.144	8.628.106	1,829
1.048.576	38.705.688	1,846
2.097.152	81.604.351	1,853

razão = medido $\div$ $n \lg n$

✓ desvio máx. 1,7%

Constante perto de 1,8: cada extração desce quase até o fundo, com 2 comparações por nível.

EXPERIMENTO

O que o experimento mostrou

A construção é **linear** (cerca de $1,9n$ comparações), confirmando que a análise ingênua $O(n \lg n)$ era pessimista. O heapsort completo fica perto de $1,8n \lg n$: o dobro de comparações do merge sort, mas **no lugar**, sem memória extra.

Desafio: o heapsort com *descida até a folha* (Floyd) reduz as comparações para perto de $n \lg n$. Implemente e meça.

➤ Reproduza: `gcc -O2 aula08_experimento.c -o exp -lm && ./exp`



Conexões

- ▶ Sistemas operacionais
- ▶ Simulação de redes
- ▶ Sistemas embarcados
- ▶ Jogos e mapas

Onde isso aparece na computação

Sistemas operacionais

Temporizadores do kernel, filas de prioridade de processos e o temporizador do runtime de Go usam heaps para achar o próximo evento em $O(\lg n)$.

Simulação de redes

Simuladores de eventos discretos (ns-3, OMNeT++) mantêm uma fila de eventos ordenada por tempo: extrair o próximo é EXTRACT-MIN.

Sistemas embarcados

Heapsort garante $O(n \lg n)$ sem memória extra nem recursão profunda; o `std::sort` do C++ (introsort) recorre a ele quando o quicksort degenera.

Jogos e mapas

Dijkstra e A*, usados em GPS e na movimentação de personagens, dependem de uma fila de prioridade em heap (aulas 17 e 31).

Para casa

Exercícios

1. (6.1-1) Quais o mínimo e o máximo de elementos num heap de altura h ?
2. (6.2-1) Ilustre $\text{MAX-HEAPIFY}(A, 3)$ em $A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$.
3. (6.3-1) Ilustre BUILD-MAX-HEAP em $A = \langle 5, 3, 17, 10, 84, 19, 6, 22, 9 \rangle$.
4. (6.4-3) Qual o tempo do heapsort num vetor já ordenado? E em ordem decrescente? Meça.
5. Escreva MAX-HEAPIFY iterativo e explique por que ele evita a pilha de chamadas.

Leitura

Cormen et al., seções 6.1 a 6.4.

Código

josedihego.net/estruturas