Governo Federal

Ministério da Educação

deINF departamentodeInformática

Universidade Federal do Maranhão

A Universidade que Cresce com Inovação e Inclusão Social

Algoritmos de Ordenação HeapSort

Estrutura de Dados II

Prof. João Dallyson

Email: Joao.dallyson@ufma.br

HeapSort

- **Possui o mesmo princípio de funcionamento da ordenação por seleção.**

- **Algoritmo:**
  - Selecione o menor item do vetor.
  - Troque-o com o item da primeira posição do vetor.
  - Repita estas operações com os n − 1 itens restantes, depois com os n − 2 itens, e assim sucessivamente.
- **O custo para encontrar o menor (ou o maior) item entre n itens é n – 1 comparações.**
- **Isso pode ser reduzido utilizando uma fila de prioridades.**

HeapSort

- **Filas de Prioridades**
  - É uma estrutura de dados onde a chave de cada item reflete sua habilidade relativa de abandonar o conjunto de itens rapidamente.

- **Aplicações:**
  - SOs usam filas de prioridades, nas quais as chaves representam o tempo em que eventos devem ocorrer.
  - Métodos numéricos iterativos são baseados na seleção repetida de um item com maior (menor) valor.
  - Sistemas de gerência de memória usam a técnica de substituir a página menos utilizada na memória principal por uma nova página.

Filas de Prioridades

- **Filas de Prioridades - Tipo Abstrato de Dados**
  - Operações:
    - Constrói uma fila de prioridades a partir de um conjunto com n itens.
    - Informa qual é o maior item do conjunto.
    - Retira o item com maior chave.
    - Insere um novo item.
    - Aumenta o valor da chave do item i para um novo valor que é maior que o valor atual da chave.
    - Substitui o maior item por um novo item, a não ser que o novo item seja maior.
    - Altera a prioridade de um item.
    - Remove um item qualquer.
    - Ajunta duas filas de prioridades em uma única.

Filas de Prioridades

- **Representação através de uma lista linear ordenada:**
  - Neste caso, Constrói leva tempo O(n log n).
  - Insere é O(n).
  - Retira é O(1).
  - Ajunta é O(n).

- **Representação é através de uma lista linear não ordenada:**
  - Neste caso, Constrói tem custo linear.
  - Insere é O(1).
  - Retira é O(n).
  - Ajunta é O(1) para apontadores e O(n) para arranjos.

Estrutura Heap

- **Filas de Prioridades – Representação**
  - A melhor representação é através de uma estruturas de dados chamada *heap*:
    - Neste caso, Constrói é O(n).
    - Insere, Retira, Substitui e Altera são O(log n).
- **Observação:**
  - Para implementar a operação Ajunta de forma eficiente e ainda preservar um custo logarítmico para as operações Insere, Retira, Substitui e Altera é necessário utilizar estruturas de dados mais sofisticadas, tais como árvores binomiais (Vuillemin, 1978).

Heap e Filas de Prioridades

- **Filas de Prioridades - Algoritmos de Ordenação**
  - As operações das filas de prioridades podem ser utilizadas para implementar algoritmos de ordenação.
  - Basta utilizar repetidamente a operação Insere para construir a fila de prioridades.
  - Em seguida, utilizar repetidamente a operação Retira para receber os itens na ordem reversa.
  - O uso de *heaps* corresponde ao método Heapsort .

HeapSort

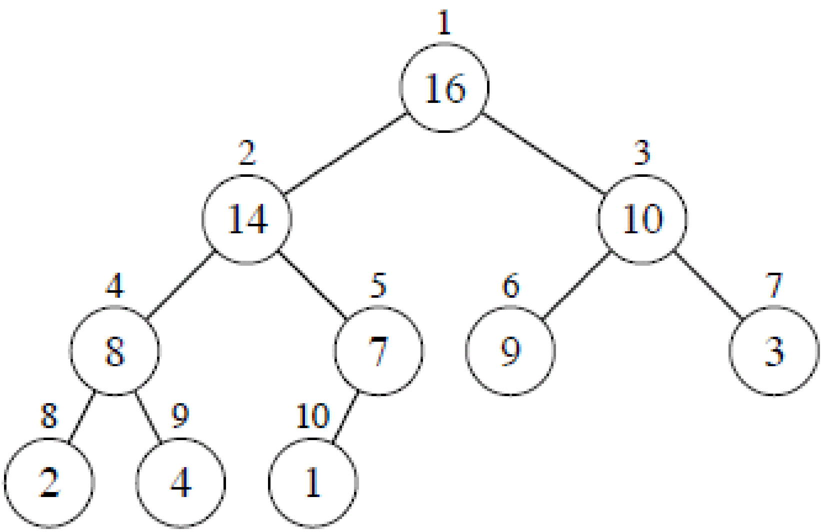
- **Heaps**
  - É uma seqüência de itens com chaves c[1], c[2], . . . , c[n], tal que:

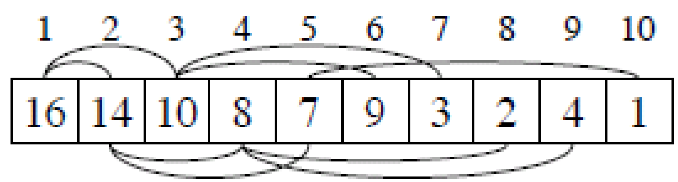
$$c[i] \geq c[2i],$$
$$c[i] \geq c[2i+1],$$

  - para todo i = 1, 2, . . . , n/2.
  - MinHeap e MaxHeap

HeapSort

- **Heap é uma estrutura de prioridades na forma de árvore binária semi-completa que representa uma ordem parcial entre os elementos do conjunto.**





HeapSort

- **Os nós são numerados de 1 a n.**
- **O primeiro nó é chamado raiz**
  - O nó k/2 é o pai do nó k, para 1 < k <= n.
  - Os nós 2k e 2k + 1 são os filhos à esquerda e à direita do nó k, para 1 <= k <= k/2.

HeapSort

- **As chaves na árvore satisfazem a condição do *heap*.**
- **As chaves em cada nós são maiores do que as chaves em seus filhos.**
- **A chave no nó raiz é a maior chave do conjunto**
- **Uma árvore binária completa pode ser representada por um arranjo**

HeapSort

- **A representação é extremamente compacta**
- **Permite caminhar pelos nós da árvore facilmente.**
- **Os filhos de um nó i estão nas posições 2i e 2i + 1.**
- **O pai de um nó i está na posição i / 2.**
- **Na representação do *heap* em um arranjo, a maior chave está sempre na posição 1 do vetor.**
- **Os algoritmos para implementar as operações sobre o *heap* operam ao longo de um dos caminhos da árvore.**

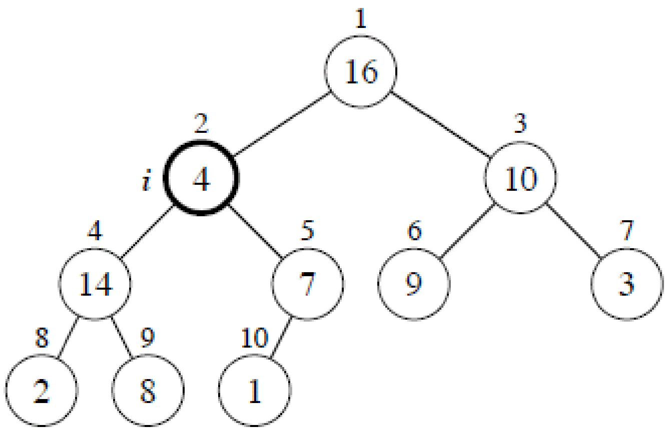
HeapSort

- **Um algoritmo elegante para construir o *heap* foi proposto por Floyd em 1964.**
- **O algoritmo não necessita de nenhuma memória auxiliar.**
- **Dado um vetor v[1], v[2], . . . , v[n].**
- **Os itens v[n/2 + 1], v[n/2 + 2], . . . , v[n] formam um *heap*:**
  – Neste intervalo não existem dois índices i e j tais que j = 2i ou j = 2i + 1.

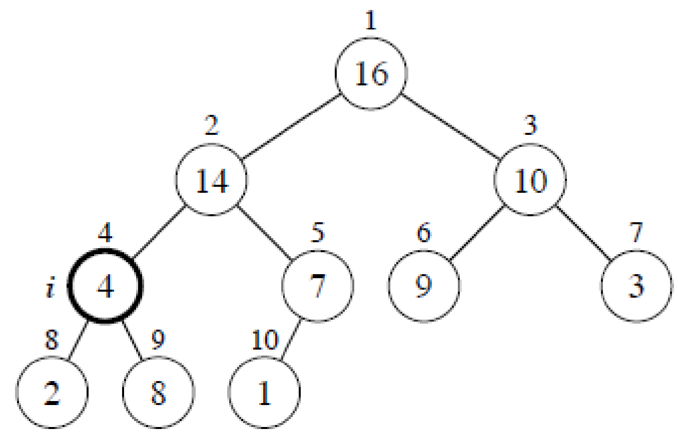
HeapSort

- **Funções**
  - Max-Heapfy
  - Build Max-Heap
  - HeapSort

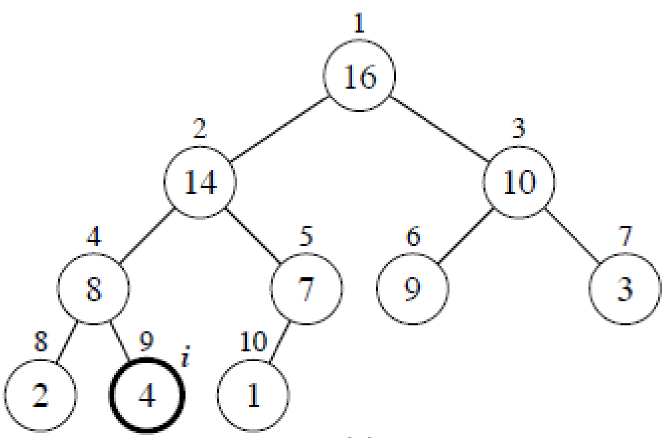
Max-Heapfy



(a)



(b)



(c)

HeapSort

$\text{MAX-HEAPIFY}(A, i, n)$

$l \leftarrow \text{LEFT}(i)$
$r \leftarrow \text{RIGHT}(i)$
if $l \leq n$ and $A[l] > A[i]$
  then $largest \leftarrow l$
  else $largest \leftarrow i$
if $r \leq n$ and $A[r] > A[largest]$
  then $largest \leftarrow r$
if $largest \neq i$
  then exchange $A[i] \leftrightarrow A[largest]$
    $\text{MAX-HEAPIFY}(A, largest, n)$

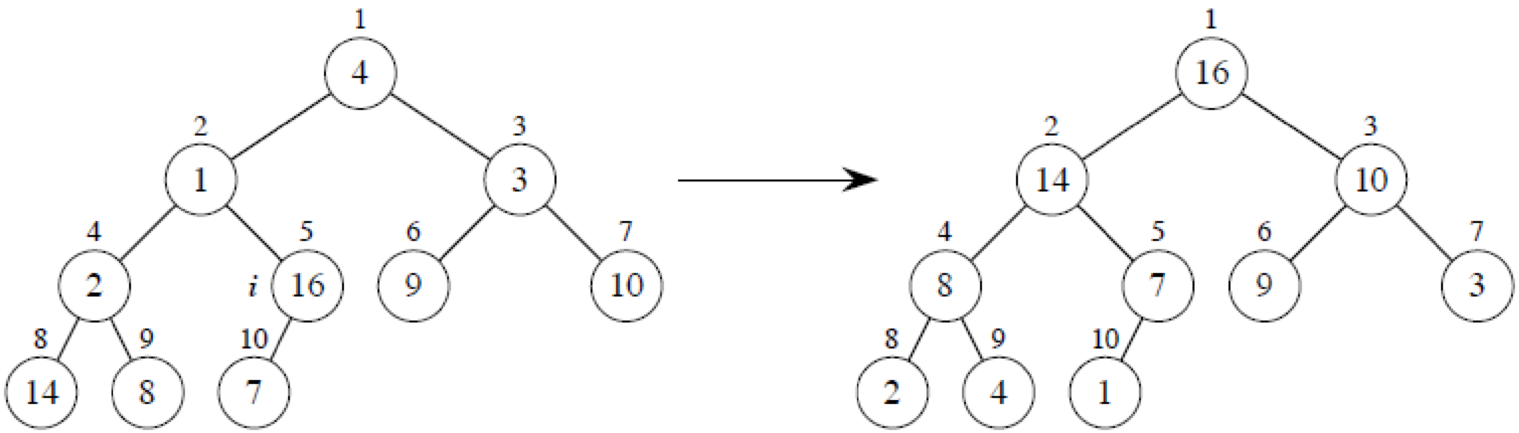
HeapSort

BUILD-MAX-HEAP(A, n)

for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 1

  do MAX-HEAPIFY(A, i, n)

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7



Heapsort

$\text{HEAPSORT}(A, n)$

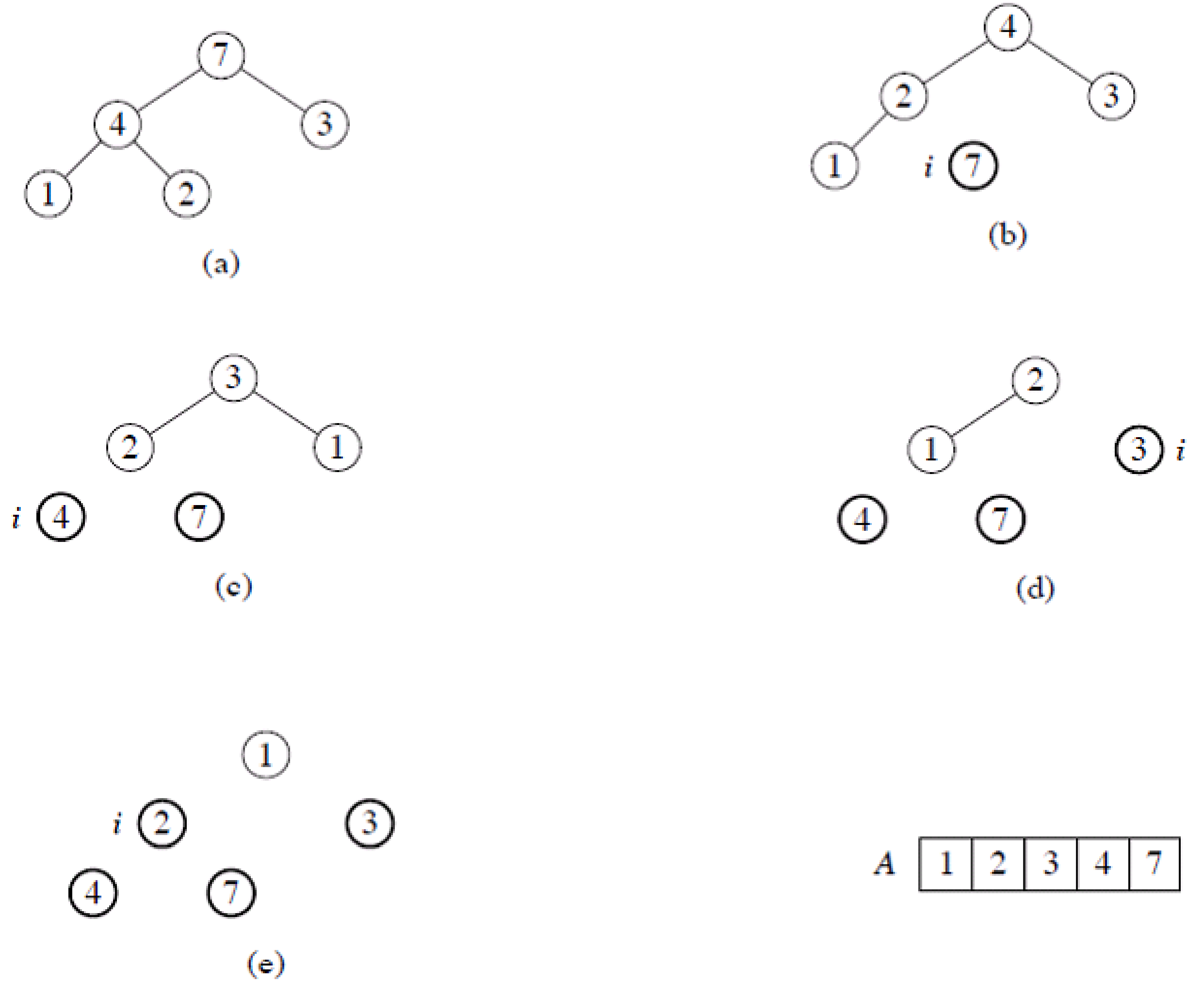
$\text{BUILD-MAX-HEAP}(A, n)$

for $i \leftarrow n$ **downto** 2

    do exchange $A[1] \leftrightarrow A[i]$

        $\text{MAX-HEAPIFY}(A, 1, i-1)$

HeapSort



Heapsort

- **O procedimento Build-Max-Heap gasta cerca de log n operações, no pior caso.**
  - Baseado no fato onde um numero menor que $n/2^{h+1}$ nós possui altura h em um momento
  - Assim: $$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) \qquad O(n)$$
- **Logo, Heapsort gasta um tempo de execução proporcional a nlog n, no pior caso.**

Heapsort

- **Vantagens:**
  - O comportamento do Heapsort é sempre O(n log n), qualquer que seja a entrada.
- **Desvantagens:**
  - O anel interno do algoritmo é bastante complexo se comparado com o do Quicksort .
  - O Heapsort não é **estável**.
- **Recomendado:**
  - Para aplicações que não podem tolerar eventualmente um caso desfavorável.
  - Não é recomendado para arquivos com poucos registros, por causa do tempo necessário para construir o *heap*.

Comparação entre os Métodos

	Complexidade
Inserção	$O(n^2)$
Seleção	$O(n^2)$
Shellsort	$O(n \log n)$
Quicksort	$O(n \log n)$
Heapsort	$O(n \log n)$

Comparação entre os Métodos

- **Registros na ordem aleatória:**

	5.00	5.000	10.000	30.000
Inserção	11,3	87	161	–
Seleção	16,2	124	228	–
Shellsort	1,2	1,6	1,7	2
Quicksort	1	1	1	1
Heapsort	1,5	1,6	1,6	1,6

Comparação entre os Métodos

- **Registros na ordem ascendente:**

	500	5.000	10.000	30.000
Inserção	1	1	1	1
Seleção	128	1.524	3.066	–
Shellsort	3,9	6,8	7,3	8,1
Quicksort	4,1	6,3	6,8	7,1
Heapsort	12,2	20,8	22,4	24,6

Comparação entre os Métodos

- **Registros na ordem descendente:**

	500	5.000	10.000	30.000
Inserção	40,3	305	575	–
Seleção	29,3	221	417	–
Shellsort	1,5	1,5	1,6	1,6
Quicksort	1	1	1	1
Heapsort	2,5	2,7	2,7	2,9

Comparação entre os Métodos

- Shellsort , Quicksort e Heapsort têm a mesma ordem de grandeza.
- O Quicksort é o mais rápido para todos os tamanhos aleatórios experimentados.
- A relação Heapsort/Quicksort se mantém constante para todos os tamanhos, sendo o Heapsort mais lento.
- A relação Shellsort/Quicksort aumenta à medida que o número de elementos aumenta; para arquivos pequenos (500 elementos), o Shellsort é mais rápido que o Heapsort ; porém, quando o tamanho da entrada cresce, essa relação se inverte.
- Entre os algoritmos de custo $O(n^2)$, o Inserção é melhor para todos os tamanhos aleatórios experimentados.

Comparação entre os Métodos

- **Influência da ordem inicial do registros:**

	Shellsort			Quicksort			Heapsort		
	5.000	10.000	30.000	5.000	10.000	30.000	5.000	10.000	30.000
Asc	1	1	1	1	1	1	1,1	1,1	1,1
Des	1,5	1,6	1,5	1,1	1,1	1,1	1	1	1
Ale	2,9	3,1	3,7	1,9	2,0	2,0	1,1	1	1

- **O Shellsort é bastante sensível à ordenação ascendente ou descendente da entrada;**
- **Em arquivos do mesmo tamanho, o Shellsort executa mais rápido para arquivos ordenados.**
- **O Quicksort é sensível à ordenação ascendente ou descendente da entrada.**

Comparação entre os Métodos

- **Em arquivos do mesmo tamanho, o Quicksort executa mais rápido para arquivos ordenados**
- **O Quicksort é o mais rápido para qualquer tamanho para arquivos na ordem ascendente.**
- **O Heapsort praticamente não é sensível à ordenação da entrada.**

Referências

Básica

- CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. Editora Campus, 2002
- Ziviani, N. Projeto de Algoritmos Com Implementações em Pascal e C, Cengage Learning, 2004.

Complementar

- TENENBAUM, Aaron; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C. São Paulo: Makron Books, 1995. ISBN: 9788534603485*
- ASCENCIO, Ana Fernanda Gomes; ARAUJO, Graziela Santos. Estruturas de Dados: Algoritmos, análise da complexidade e implementações em Java e C/C++. Pearson Prentice Hall, 2010
- DROZDEK, Adam. Adam Drozdek. Data Structures and Algorithms in Java. 2. Cengage Learning. 2004. 2. Cengage Learning. 2004
- GOODRICH, Michael T. Estruturas de dados e algoritmos em java. 4 ED. Porto Alegre: Bookman, 2007. 600.
- SKIENA, Steven S.. **The Algorithm Design Manual**. 2. Springer-Verlag. 2008
- Notas de aula: prof. Ítalo Cunha – UFMG. 2012.

Perguntas....