Análise de Algoritmos
Algoritmos de Ordenação

Nelson Cruz Sampaio Neto
nelsonneto@ufpa.br

Universidade Federal do Pará
Instituto de Ciências Exatas e Naturais
Faculdade de Computação

5 de abril de 2016

Por que ordenar?

- Às vezes, a necessidade de ordenar informações é inerente a uma aplicação. Por exemplo: Para preparar os extratos dos clientes, o banco precisa ordenar os cheques pelo número.
- Outros algoritmos usam frequentemente a ordenação como uma sub-rotina chave. Exemplo: Pesquisa binária.
- A ordenação é um problema de interesse histórico, logo, existe uma variedade de algoritmos de ordenação que empregam um rico conjunto de técnicas.
- Muitas questões de engenharia surgem ao se implementar algoritmos de ordenação, como hierarquia de memória do computador e do ambiente de *software*.

Relembrando alguns métodos de ordenação

- A **ordenação por inserção** tem complexidade no tempo linear no melhor caso (vetor ordenado) e quadrática no pior caso (vetor em ordem decrescente).
- Já o método **BubbleSort** e a **ordenação por seleção** têm complexidade no tempo $\Theta(n^2)$.
- Métodos considerados extremamente complexos!
- Por isso, não são recomendados para programas que precisem de velocidade e operem com quantidade elevada de dados.

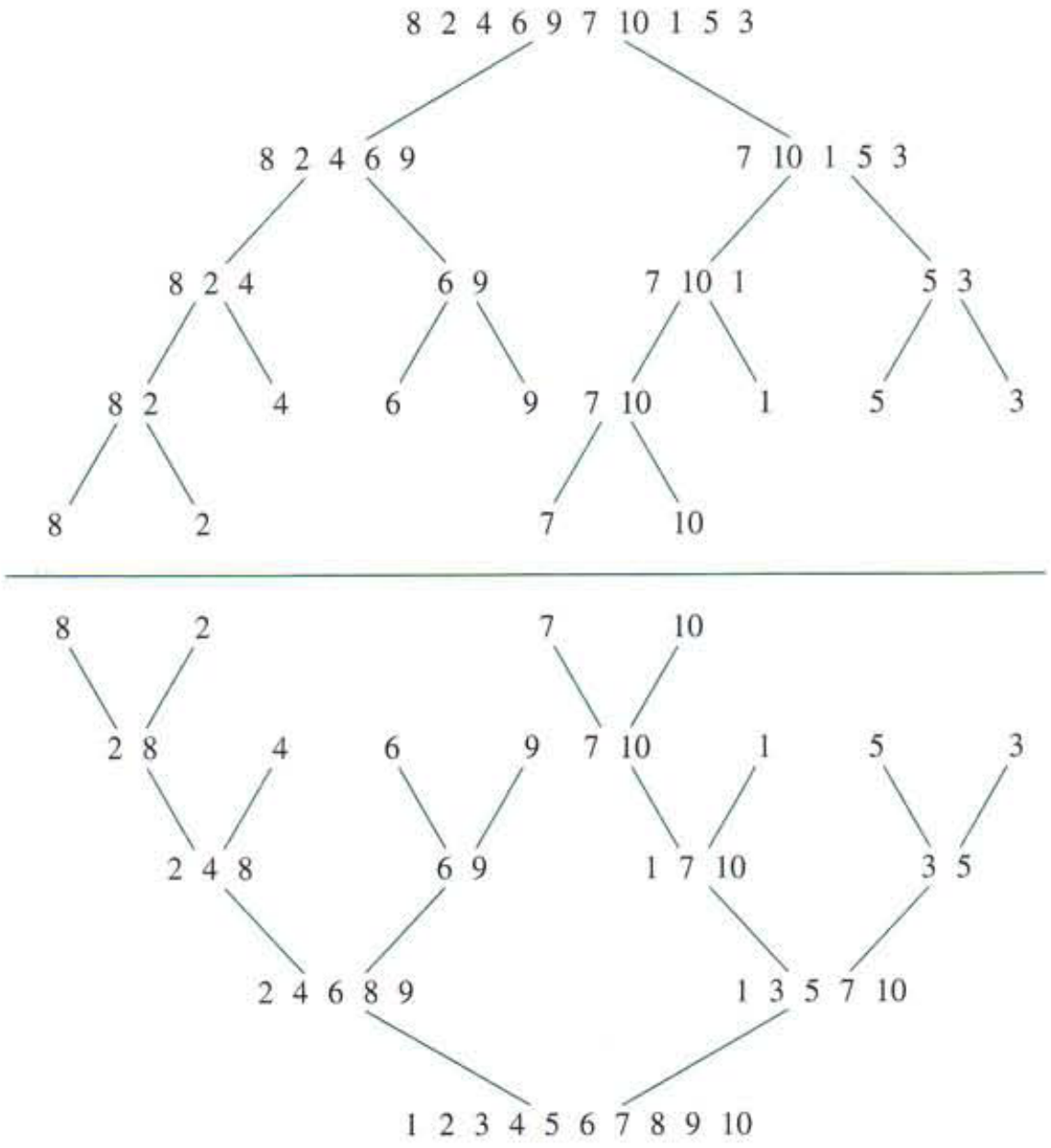
Método MergeSort

- Também chamado de ordenação por intercalação, ou mistura.

```
MERGE-SORT (A, p, r)
1. se ( p < r ) então
2.     q = (p + r) / 2
3.     MERGE-SORT (A, p, q)
4.     MERGE-SORT (A, q + 1, r)
5.     MERGE (A, p, q, r)
```

- Sua complexidade no tempo é $\Theta(n\ log_2 n)$, dado que a função MERGE é $\Theta(n)$ e um vetor com 1 elemento está ordenado.
- Vantagem: A complexidade do MergeSort não depende da sequência de entrada.
- Desvantagem: A função MERGE requer um vetor auxiliar, o que aumenta o consumo de memória e tempo de execução.

Exemplo de operação do MergeSort



Método QuickSort

- O QuickSort é provavelmente o algoritmo mais usado na prática para ordenar vetores.
- O passo crucial do algoritmo é escolher um elemento do vetor para servir de **pivô**. Por isso, seu tempo de execução depende dos dados de entrada.
- Sua complexidade no melhor caso é $\Theta(n\ log_2\ n)$. Semelhante ao MergeSort, mas necessitada apenas de uma pequena pilha como memoria auxiliar.
- Sua complexidade de pior caso é $\Theta(n^2)$, mas a chance dela ocorrer fica menor à medida que n cresce.

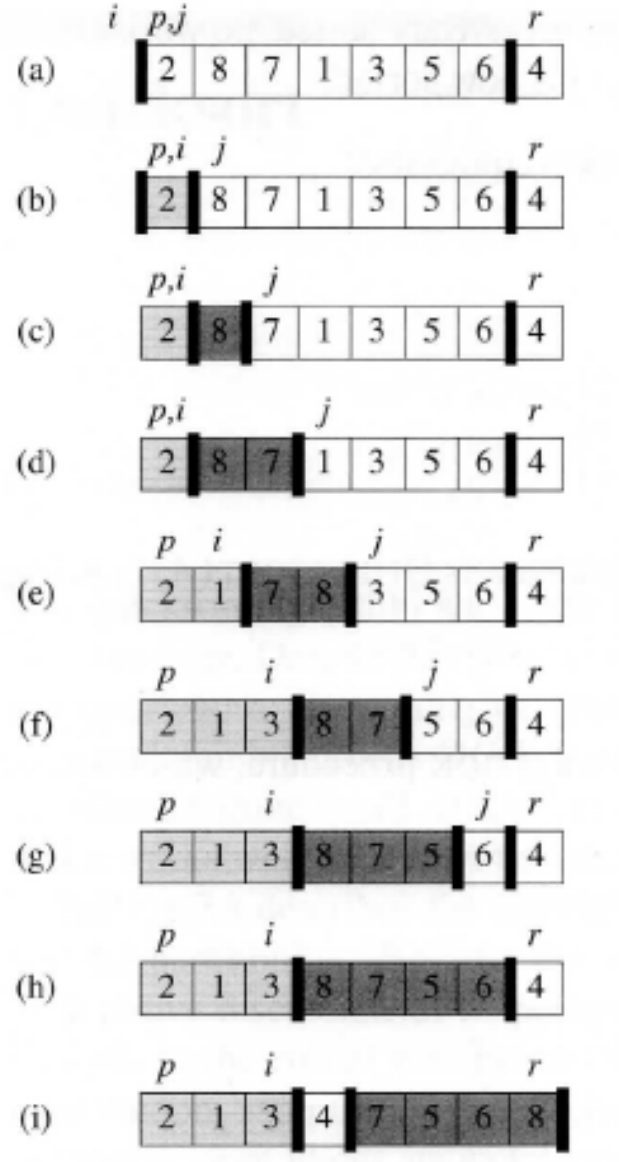
Particionamento do vetor

- A chave do QuickSort é a rotina PARTICAO, que escolhe o elemento pivô e o coloca na sua posição correta.

```
PARTICAO (A, p, r)
1. x = A[r]  // o último elemento é o pivô
2. i = p - 1
3. para (j = p) até (r - 1) faça
4.     se (A[j] <= x) então
5.         i = i + 1
6.         troca A[i] com A[j]
7. troca A[i + 1] com A[r]
8. retorna (i + 1)
```

- O tempo de execução do algoritmo PARTICAO é $\Theta(n)$.

Exemplo de operação do algoritmo PARTICAO



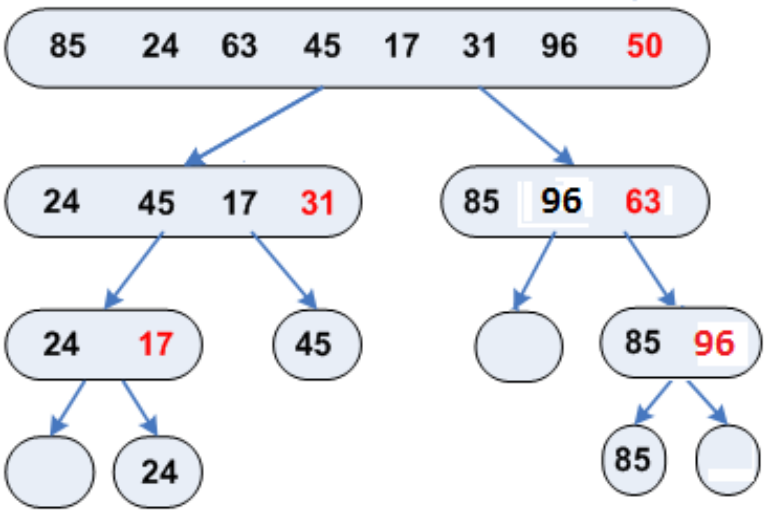
Algoritmo QuickSort

- O algoritmo abaixo implementa o QuickSort (recursivo), até que todos os segmentos tenham tamanho ≤ 1.

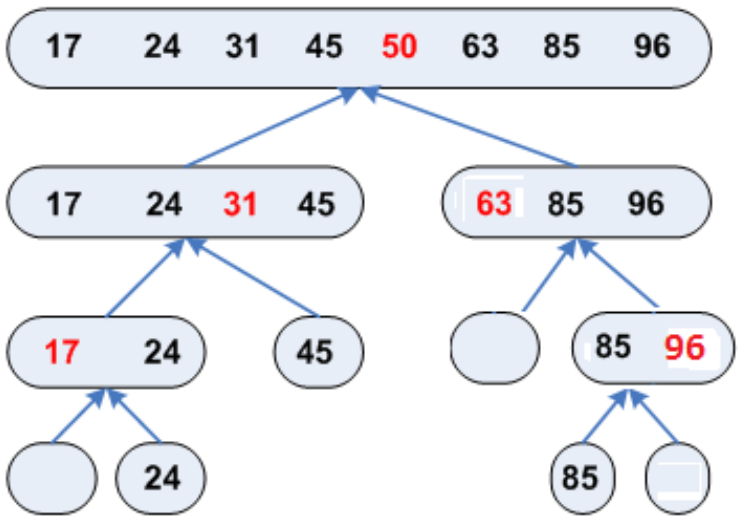
```
QUICK-SORT (A, p, r)
1. se (p < r) então
2.     q = PARTICAO (A, p, r)
3.     QUICK-SORT (A, p, q - 1)
4.     QUICK-SORT (A, q + 1, r)
```

- Após particionar o vetor em dois, os segmentos são ordenados recursivamente, primeiro o da esquerda e depois o da direita.

Exemplo de operação do QuickSort



(a) Fase de Divisão



(b) Fase de Conquista

- O tempo de execução depende do particionamento do vetor: balanceado (melhor caso) ou não balanceado (pior caso).

Particionamento não balanceado

- O comportamento do QuickSort no pior caso ocorre quando a rotina PARTICAO produz um segmento com $n-1$ elementos e outro com 0 (zero) elementos.

- Nesse caso, a fórmula de recorrência para o algoritmo assume a seguinte forma:

$$T(0) = \Theta(1)$$
$$T(1) = \Theta(1)$$
$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$T(n) = T(n-1) + \Theta(n), \text{ para } n > 1$$

- Resolvendo a formulação acima, obtém-se $\Theta(n^2)$.

- Situação quando o vetor de entrada já está ordenado.

Particionamento não balanceado

- Passo base: $T(1) = 1$

- **Expandir**:

$k = 1$: $T(n) = T(n-1) + n$

Particionamento não balanceado

- Passo base: $T(1) = 1$

- **Expandir**:

$k = 1$: $T(n) = T(n-1) + n$

$k = 2$: $T(n) = [T(n-2) + n - 1] + n = T(n-2) + n - 1 + n$

Particionamento não balanceado

- Passo base: $T(1) = 1$

- **Expandir**:

$k = 1$: $T(n) = T(n-1) + n$

$k = 2$: $T(n) = [T(n-2) + n - 1] + n = T(n-2) + n - 1 + n$

$k = 3$: $T(n) = [T(n-3) + n - 2] + n - 1 + n = T(n-3) + n - 2 + n - 1 + n$

Particionamento não balanceado

- Passo base: $T(1) = 1$

- **Expandir**:

$k = 1$: $T(n) = T(n-1) + n$

$k = 2$: $T(n) = [T(n-2) + n - 1] + n = T(n-2) + n - 1 + n$

$k = 3$: $T(n) = [T(n-3) + n - 2] + n - 1 + n = T(n-3) + n - 2 + n - 1 + n$

- **Conjecturar**: Após k expansões, temos

$$T(n) = T(n-k) + \sum_{i=0}^{k-1} (n-i).$$

Observando a expansão, ela irá parar quando $k = n - 1$, isso porque a base da recorrência é definida para 1 (um).

Logo, $T(n) = T(1) + \sum_{i=0}^{n-2} (n-i) = 1 + \frac{n(n+1)}{2} - 1 = \Theta(n^2)$.

Particionamento balanceado

- O comportamento no melhor caso ocorre quando a rotina PARTICAO produz dois segmentos, cada um de tamanho não maior que a metade de n.

- Nesse caso, a fórmula de recorrência para o algoritmo assume a seguinte forma:

$$T(1) = \Theta(1)$$

$$T(n) = T(\tfrac{n}{2}) + T(\tfrac{n}{2}) + \Theta(n)$$

$$T(n) = 2\ T(\tfrac{n}{2}) + \Theta(n), \text{ para } n > 1$$

- Resolvendo a recorrência acima pelo método mestre, obtém-se $\Theta(n\ log_2\ n)$.

Comportamento no caso médio

- O tempo de execução do caso médio do QuickSort é muito mais próximo do melhor caso que do pior caso.

- Podemos analisar a afirmação acima entendendo como o equilíbrio do particionamento se reflete na recorrência:

  $T(1) = \Theta(1)$

  $T(n) = T(\frac{9n}{10}) + T(\frac{n}{10}) + \Theta(n)$, para $n > 1$

- Resolvendo a formulação acima, obtém-se $\Theta(n\ log_{\frac{10}{9}}\ n)$.

- Assintoticamente o mesmo tempo que levaria se a divisão fosse feita exatamente no meio.

Comportamento no caso médio

- No entanto, é pouco provável que o particionamento sempre ocorra do mesmo modo em todos os níveis.
- Espera-se que algumas divisões sejam razoavelmente bem equilibradas e outras bastante desequilibradas.
- Então, no caso médio, o QuickSort produz uma mistura de divisões “boas” e “ruins” com tempo de execução semelhante ao do melhor caso.

Uma versão aleatória do QuickSort

- Como visto, a escolha do pivô influencia decisivamente no tempo de execução do QuickSort.
- Por exemplo, um vetor de entrada ordenado leva a $\Theta(n^2)$, caso o pivô escolhido seja o último elemento.
- A escolha do **elemento do meio** como pivô melhora muito o desempenho quando o vetor está ordenado, ou quase.
- Outra alternativa é escolher o pivô **aleatoriamente**. Às vezes adicionamos um caráter aleatório a um algoritmo para obter bom desempenho no caso médio sobre todas as entradas.

Uma versão aleatória do QuickSort

- Ao invés de sempre usar o $A[r]$ como pivô, usaremos um elemento escolhido ao acaso dentro do vetor $A[p \,..\, r]$.

```
RAND-PARTICAO (A, p, r)
1. i = RANDOM (p, r)
2. troca A[r] com A[i]
3. retorna PARTICAO (A, p, r)
```

- Essa modificação assegura que o elemento pivô tem a mesma probabilidade de ser qualquer um dos elementos do vetor.

- Como o elemento pivô é escolhido ao acaso, espera-se que a divisão do vetor seja bem equilibrada na média.

Uma versão aleatória do QuickSort

- O algoritmo QuickSort aleatório é descrito abaixo.

```
RAND-QUICK-SORT (A, p, r)
1. se (p < r) então
2.     q = RAND-PARTICAO (A, p, r)
3.     RAND-QUICK-SORT (A, p, q - 1)
4.     RAND-QUICK-SORT (A, q + 1, r)
```

- A aleatoriedade **não** vai evitar o pior caso!

- Muitas pessoas consideram a versão aleatória do QuickSort o algoritmo preferido para entradas grandes o suficiente.

Exercícios

1. Ilustre a operação da rotina PARTICAO sobre o vetor $A = [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]$.

2. De que maneira você modificaria o QuickSort para fazer a ordenação de forma decrescente?

3. Qual é a complexidade no tempo do QuickSort quando todos os elementos do vetor A têm o mesmo valor?

4. Quando o vetor A contém elementos distintos, está ordenado em ordem decrescente e o pivô é o último elemento, qual é a complexidade no tempo do QuickSort?

5. A análise feita no exercício anterior muda caso o pivô seja o elemento do meio?

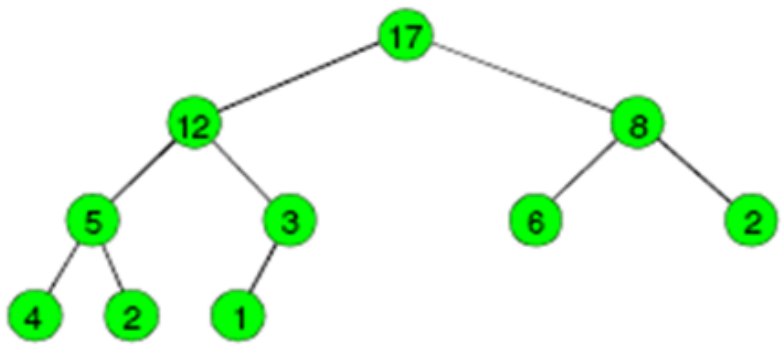
6. Dado o vetor $[f\ e\ d\ h\ a\ c\ g\ b]$ e tomando o elemento d como pivô, ilustre a operação do RAND-PARTICAO sobre o vetor.

Método HeapSort

- Utiliza uma estrutura de dados com um critério bem-definido baseada em árvore binária (*heap*) para organizar a informação durante a execução do algoritmo.
- Sua complexidade é $\Theta(n\ log_2\ n)$. Semelhante ao MergeSort e QuickSort, mas não necessita de memória adicional.
- O QuickSort geralmente supera o HeapSort na prática, mas o seu pior caso $\Theta(n^2)$ é inaceitável em algumas situações.
- Já o HeapSort é mais adequado para quem necessita garantir tempo de execução e não é recomendado para arquivos com poucos registros.

Heap

- A estrutura de dados *heap* é um objeto arranjo que pode ser visto como uma árvore binária completa.
- Um *heap* deve satisfazer uma das seguintes condições:
  - Todo nó deve ter valor maior ou igual que seus filhos (**Heap Máximo**). O maior elemento é armazenado na raiz.
  - Todo nó deve ter valor menor ou igual que seus filhos (**Heap Mínimo**). O menor elemento é armazenado na raiz.



- Acima, um *heap* máximo de altura 3. Note que o último nível pode não conter os nós mais à direita.

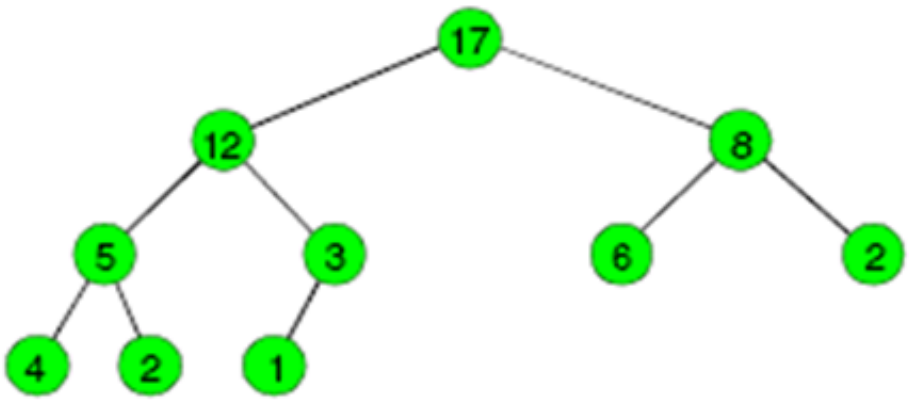
Heap

- Tendo em vista que um *heap* de n elementos é baseado em uma árvore binária completa, sua altura h é $\lfloor log_2\ n \rfloor$.
- A altura de um nó i é o número de nós do maior caminho de i até um de seus descendentes. As folhas têm altura zero.
- Se o *heap* for uma árvore binária completa com o último nível cheio, seu número total de elementos será $2^{h+1} - 1$.
- Se o último nível do *heap* tiver apenas um elemento, seu número total de elementos será 2^h.
- Logo, o número n de elementos de um *heap* é dado por:

$$2^h \leq n \leq 2^{h+1} - 1$$

O que acontece na prática?

- Na prática, quando se trabalha com *heap*, recebe-se um vetor que será representado por árvore binária da seguinte forma:
  - Raiz da árvore: primeira posição do vetor;
  - Filhos do nó na posição i: posições $2i$ e $2i + 1$; e
  - Pai do nó na posição i: posição $\lfloor \frac{i}{2} \rfloor$.
- Exemplo: O vetor $A = [17\ 12\ 8\ 5\ 3\ 6\ 2\ 4\ 2\ 1]$ pode ser representado pela árvore binária (*max-heap*) abaixo:



Algoritmos para *heap*

- A representação em vetores, permite relacionar os nós do *heap* da seguinte forma:

```
Pai (i)
1. retorna (int) i/2
```

```
Esq (i)
1. retorna 2*i
```

```
Dir (i)
1. retorna 2*i + 1
```

Exercícios

1. Um arranjo (ou vetor) de números inteiros que está ordenado de forma crescente é um *heap* mínimo?

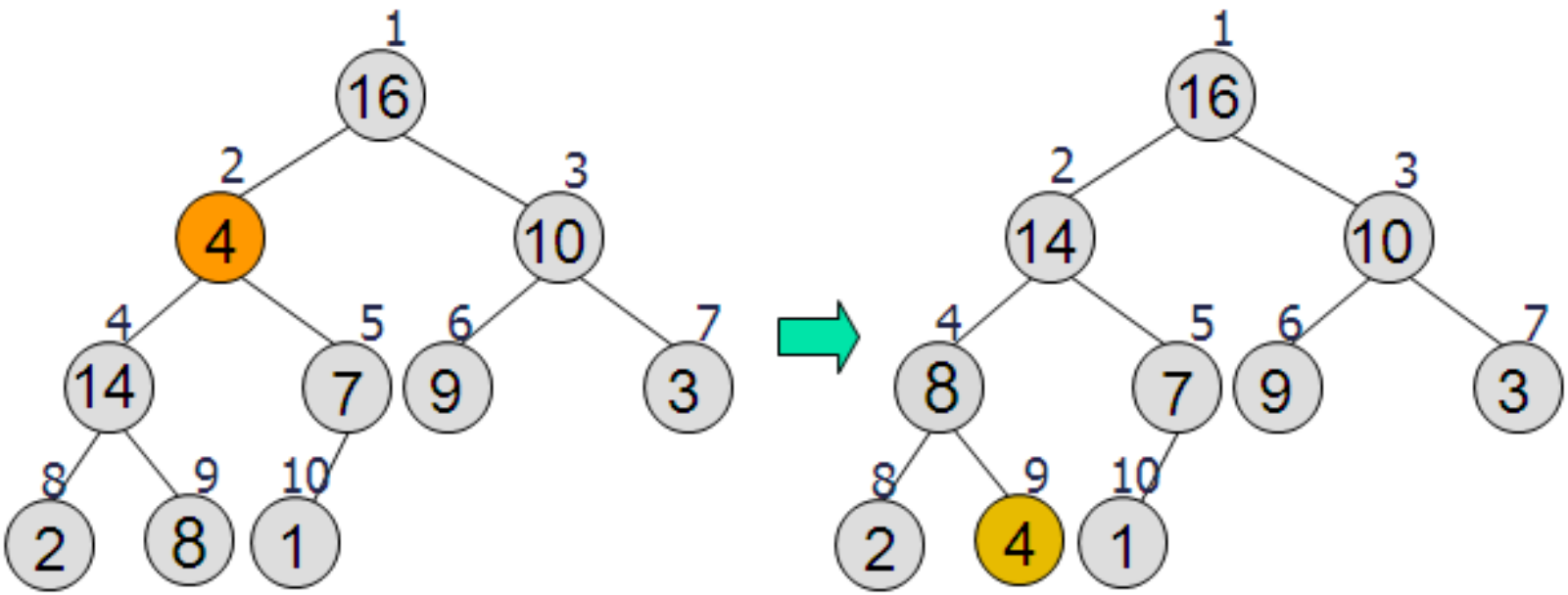
2. A sequência $[23, 17, 14, 6, 13, 10, 1, 5, 7]$ é um *heap* máximo?

3. Onde em um *heap* máximo o menor elemento poderia residir, supondo-se que todos os elementos sejam distintos?

4. Todo *heap* é uma árvore binária de pesquisa? Por quê?

Procedimento PENEIRA

- Mas o que acontece se a condição *max-heap* for quebrada?
- Exemplo: O vetor $A = [16\ 4\ 10\ 14\ 7\ 9\ 3\ 2\ 8\ 1]$ é representado pela árvore mais a esquerda, que precisa ser reorganizada na posição 2 para adquirir a condição *max-heap*.



Procedimento PENEIRA

- O procedimento PENEIRA mantém a condição *max-heap*.

```
PENEIRA (A, n, i)
1. l = Esq (i)  //  l = 2i
2. r = Dir (i)  //  r = 2i + 1
3. maior = i
4. se (l <= n) e (A[l] > A[maior])
5.    maior = l
6. se (r <= n) e (A[r] > A[maior])
7.    maior = r
8. se (maior != i)
9.    troca A[i] com A[maior]
10.   PENEIRA (A, n, maior)
```

Procedimento PENEIRA

- O tempo de execução do PENEIRA é $\Theta(1)$ para corrigir o relacionamento entre os elementos $A[i]$, $A[l]$ e $A[r]$, mais o tempo para rodá-lo recursivamente em uma subárvore com raiz em um dos filhos do nó i.

- As subárvores de cada filho têm tamanho máximo de $2n/3$ - o pior caso acontece quando o último nível da árvore está exatamente metade cheio - e o tempo de execução pode ser expresso pela recorrência:

  $$T(n) \leq T(2n/3) + \Theta(1).$$

- A solução para essa recorrência é $T(n) = O(log_2\ n)$. Ou seja, podemos caracterizar o tempo de execução do PENEIRA em um nó de altura h como $O(h)$.

Procedimento PENEIRA não recursivo

- A complexidade no tempo também é $O(log_2\ n)$, porém sem a necessidade de alocação de memória auxiliar.

```
INT-PENEIRA (A, n, i)
1.  enquanto 2i <= n faça
2.     maior = 2i
3.     se (maior < n) e (A[maior] < A[maior + 1])
4.        maior = maior + 1
5.     fim
6.     se (A[i] >= A[maior])
7.        i = n
8.     senão
9.        troca A[i] com A[maior]
10.       i = maior
11.    fim
12. fim
```

Procedimento MAX-HEAP

- O procedimento MAX-HEAP abaixo converte um vetor A de n elementos em um *heap* máximo.

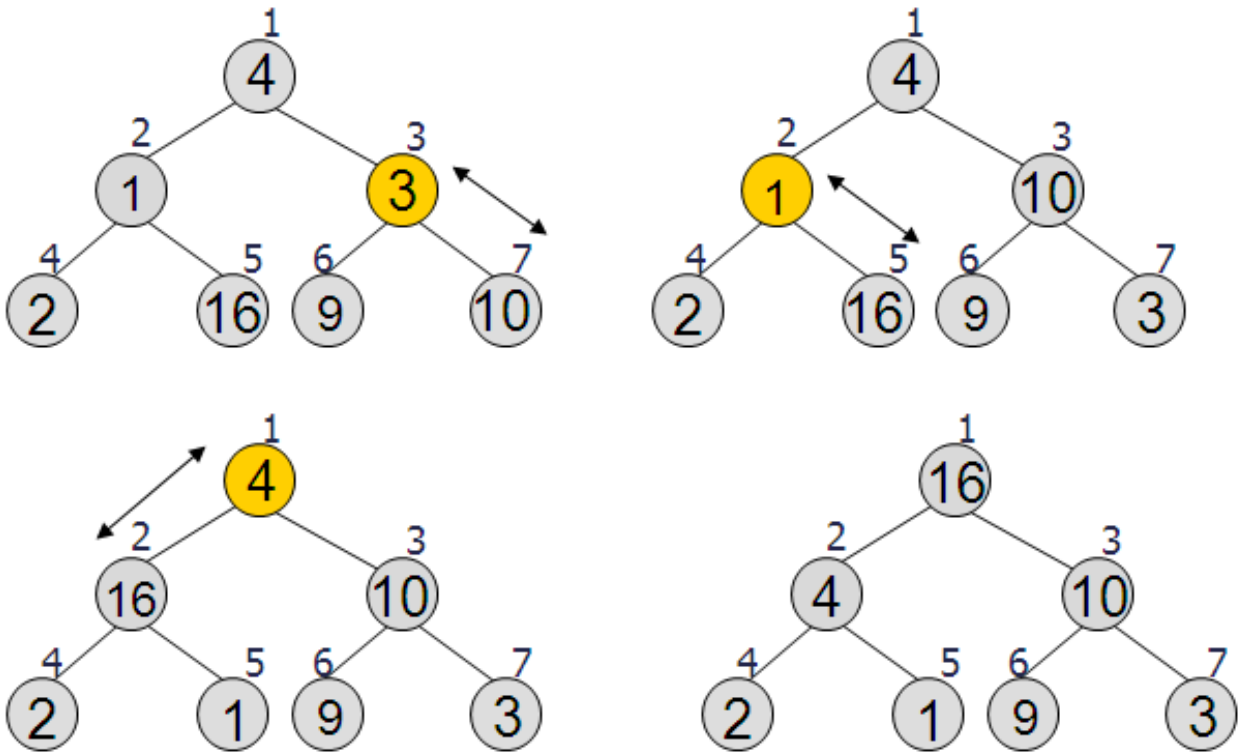
```
MAX-HEAP (A, n)
1. para (i = n/2) até 1 faça
2.     PENEIRA (A, n, i)
```

- Os elementos de $A[\frac{n}{2} + 1]$ até $A[n]$ correspondem às folhas da árvore e, portanto, são *heaps* de um elemento.

- Logo, basta chamar o procedimento PENEIRA para os demais elementos do vetor A, ou seja, de $A[\frac{n}{2}]$ até $A[1]$.

Procedimento MAX-HEAP

- Exemplo: A figura abaixo ilustra a operação do procedimento MAX-HEAP para o vetor $A = [4\ 1\ 3\ 2\ 16\ 9\ 10]$.



Procedimento MAX-HEAP

- Tempo de execução do MAX-HEAP é $O(n\ log_2\ n)$. Esse limite superior, embora correto, não é assintoticamente restrito.
- De fato, o tempo de execução do PENEIRA sobre um nó varia com a altura do nó na árvore, e as alturas na maioria dos nós são pequenas.
- A análise mais restrita se baseia em duas propriedades:
  - Um *heap* de n elementos tem altura $\lfloor log_2\ n \rfloor$; e
  - No máximo $\lceil \frac{n}{2^{h+1}} \rceil$ nós de qualquer altura h.

Procedimento MAX-HEAP

- Assim, expressa-se o custo total do MAX-HEAP por

$$\sum_{h=0}^{\lfloor log_2\ n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor log_2\ n \rfloor} \frac{h}{2^h})$$

$$\leq O(n \sum_{h=0}^{\infty} \frac{h}{2^h})$$

Sabe-se que $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2$

Desse modo, o tempo de execução do MAX-HEAP pode ser limitado como $O(n)$.

- Ou seja, podemos transformar um vetor desordenado em um *heap* máximo em tempo linear!

Algoritmo HeapSort

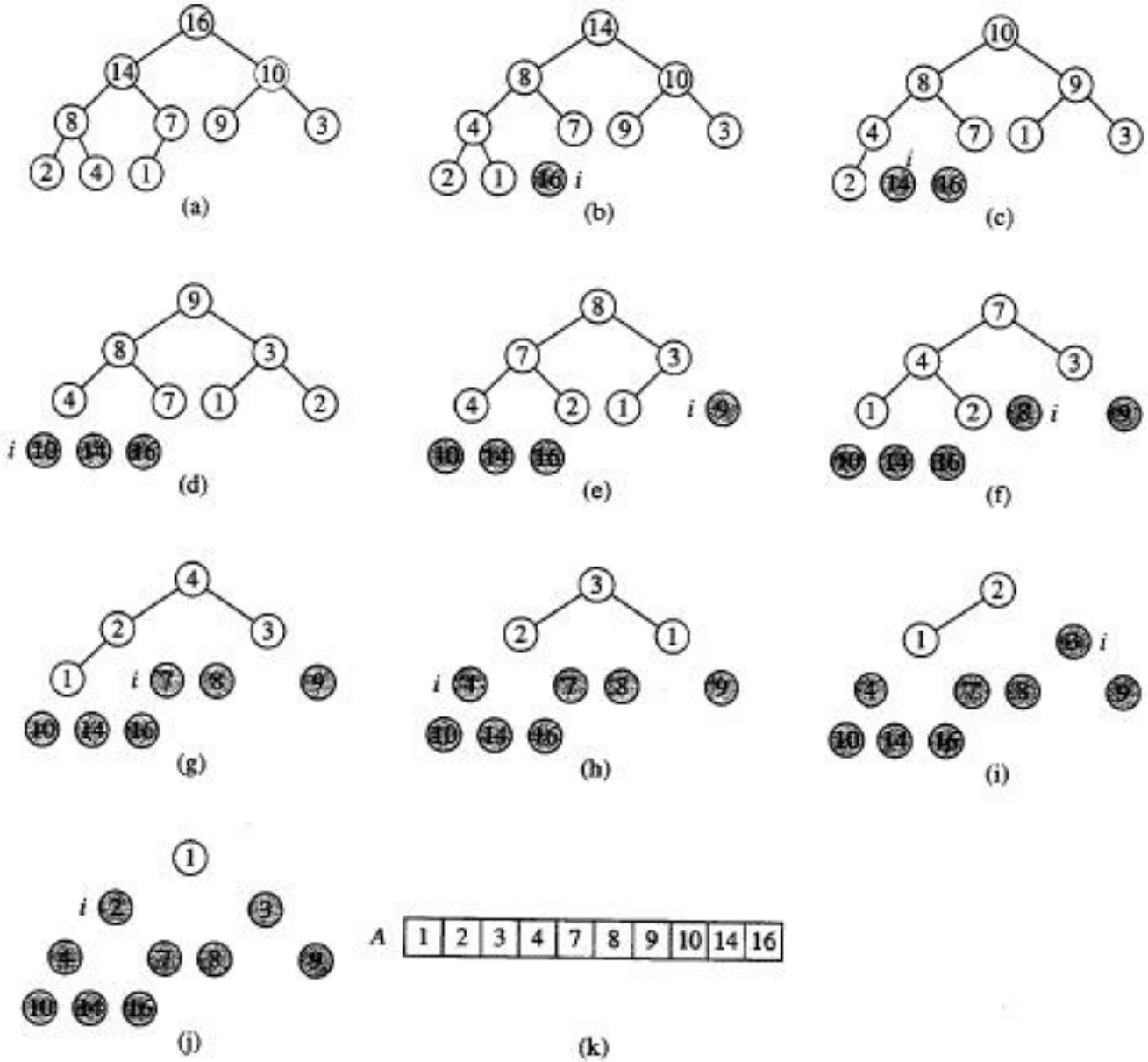
- O algoritmo abaixo implementa o método HeapSort.

```
HEAP-SORT (A, n)
1. MAX-HEAP (A, n) // constrói um max-heap
2. tamanho = n
3. para (i = n) até 2 faça
4.    troca A[1] com A[i] // raiz no final
5.    tamanho = tamanho - 1
6.    PENEIRA (A, tamanho, 1)
```

- Após construir um *max-heap*, a raiz é movida para o final e o vetor é reorganizado. Esse processo é repetido até que o *heap* tenha tamanho 2.

- Complexidade: $\Theta(n \log_2 n)$. Na prática, o tempo de execução tende a ser menor caso o vetor de entrada esteja reversamente ordenado. Por quê?

Algoritmo HeapSort



Exercícios

1. O vetor $T = [23\ 17\ 14\ 6\ 13\ 10\ 1\ 5\ 7\ 12]$ é um *heap* máximo? Qual é a altura do *heap* formado?

2. Ilustre a operação do procedimento PENEIRA(A, 14, 3) sobre o vetor $A = [27\ 17\ 3\ 16\ 13\ 10\ 1\ 5\ 7\ 12\ 4\ 8\ 9\ 0]$.

3. Ilustre a operação do procedimento MAX-HEAP para construir um *heap* a partir do vetor $A = [5\ 3\ 17\ 10\ 84\ 19\ 6\ 22\ 9]$.

4. Ilustre a operação do algoritmo HEAP-SORT sobre o vetor $A = [5\ 13\ 2\ 25\ 7\ 17\ 20\ 8\ 4]$.

Exercício complementar

- A tabela abaixo mostra o resultado (seg) de experimentos para ordenar um vetor de 10^6 elementos de forma crescente considerando quatro situações iniciais.

Algoritmo	**Aleatória**	**Crescente**	**Reversa**	**Igual**
MergeSort	0,9	0,8	0,8	0,8
QuickSort	0,6	0,3	0,3	35,0
HeapSort	0,8	0,8	0,7	0,2

- Qual foi a versão do QuickSort usada nos experimentos?
- Faça uma análise do desempenho do HeapSort.
- Qual algoritmo você usaria para ordenar $2x10^6$ elementos?
- Você mudaria de ideia se a aplicação não pudesse tolerar eventuais casos desfavoráveis?

Fila de prioridades

- O HeapSort é um algoritmo excelente, contudo, uma boa implementação do QuickSort, normalmente o supera.
- Porém, a estrutura de dados *heap* é de grande utilidade.
- Exemplo: Uso de *heap* como uma **fila de prioridades**.
- Fila de prioridades é uma estrutura de dados para manutenção de um conjunto de dados, cada qual com um valor associado chamado **chave**.
- Existem dois tipos de fila de prioridades: **máxima** e **mínima**.

Fila de prioridades

- Em muitas aplicações, dados de uma coleção são acessados por ordem de prioridade.
- A prioridade associada ao dado pode ser qualquer coisa: tempo, custo... mas precisa ser um escalar.
- Exemplos: execução de processos (máxima) e simulação orientada a eventos (mínima).
- Uma fila de prioridades admite as seguintes operações: seleção e remoção do dado com maior (ou menor) chave; alteração de prioridade; e inserção.

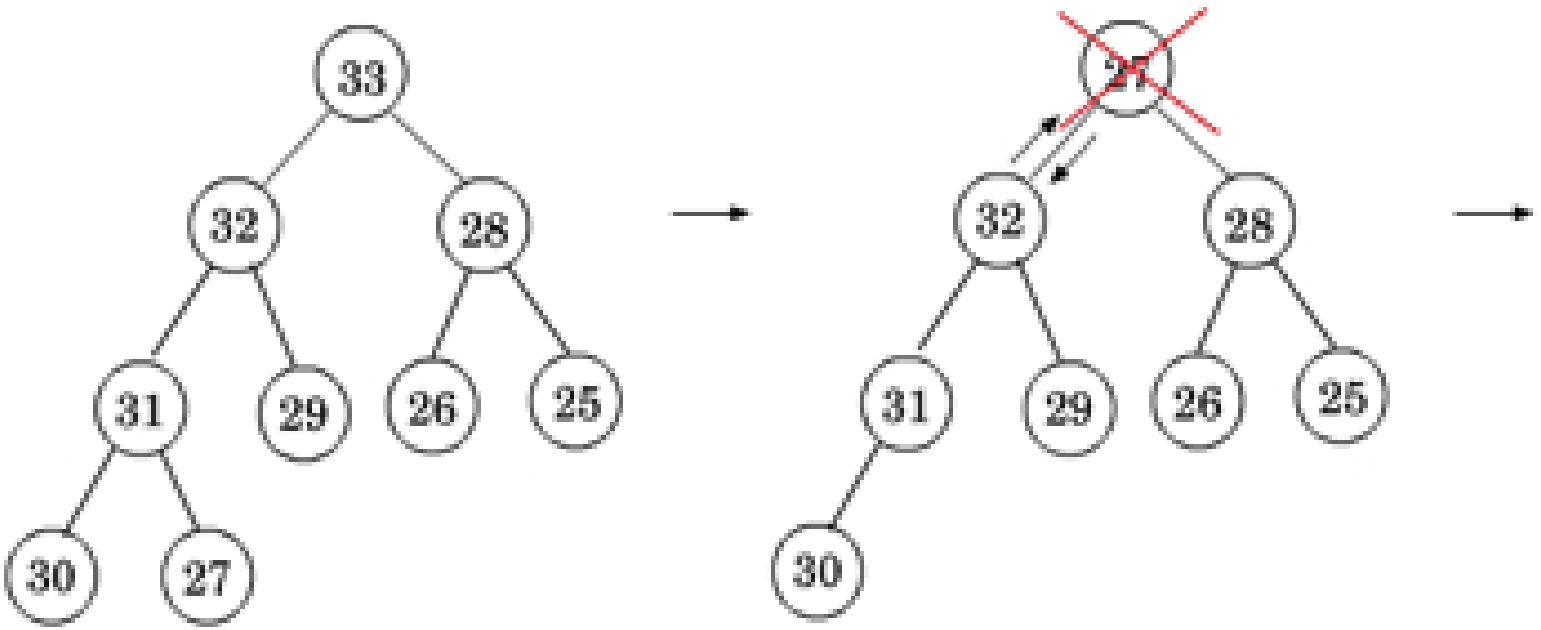
Algoritmos: Seleção e Remoção

```
HEAP-MAXIMUM (A)
1. retorna A[1]
```

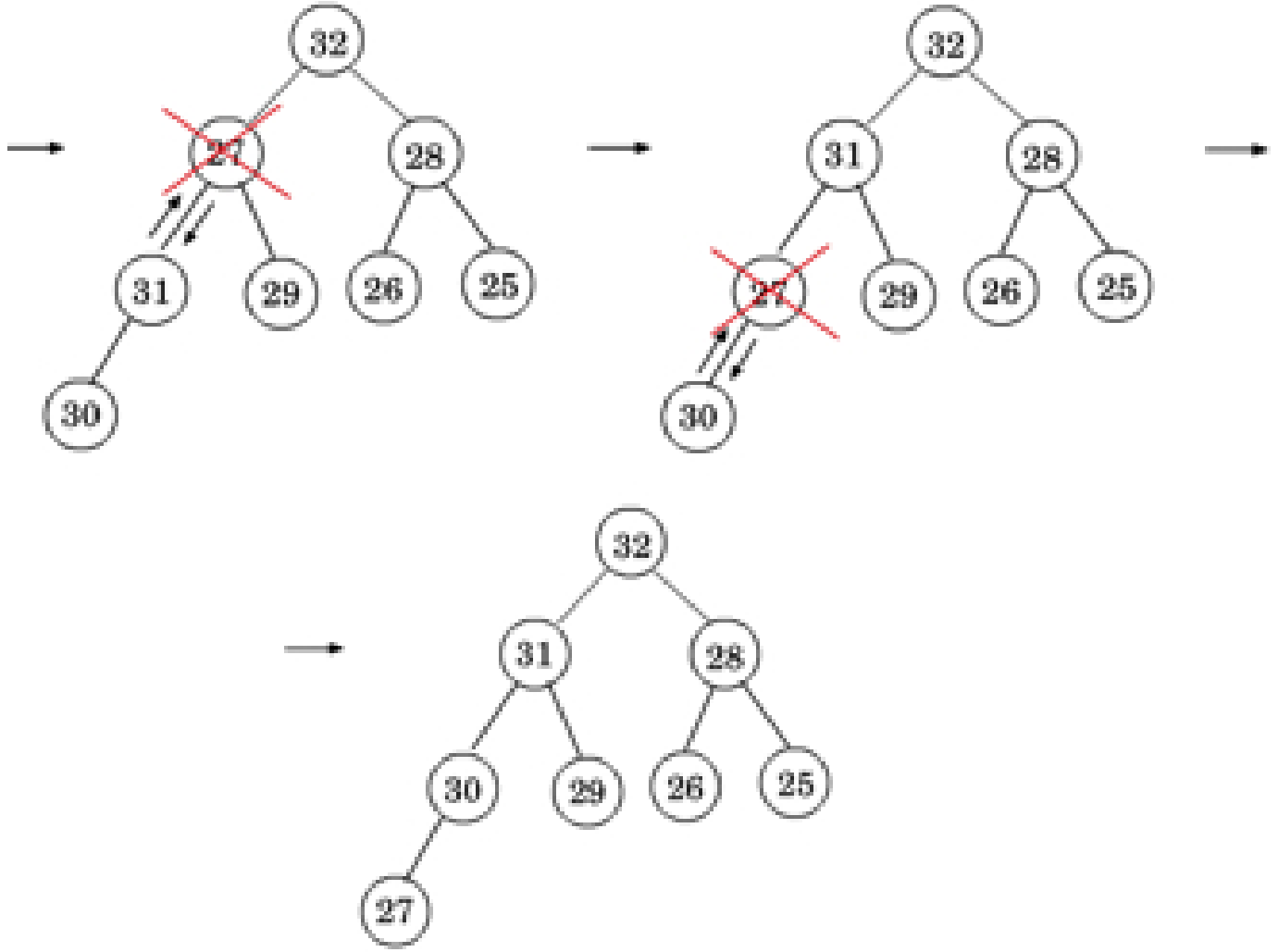
```
HEAP-EXTRACT-MAX (A, n)
1. se n < 1
2.   então erro 'heap vazio'
3. maximo = A[1]
4. A[1] = A[n]
5. n = n - 1
6. PENEIRA (A, n, 1)
7. retorna maximo
```

Procedimento HEAP-EXTRACT-MAX

- Exemplo: A figura abaixo ilustra a operação do procedimento HEAP-EXTRACT-MAX.



Procedimento HEAP-EXTRACT-MAX



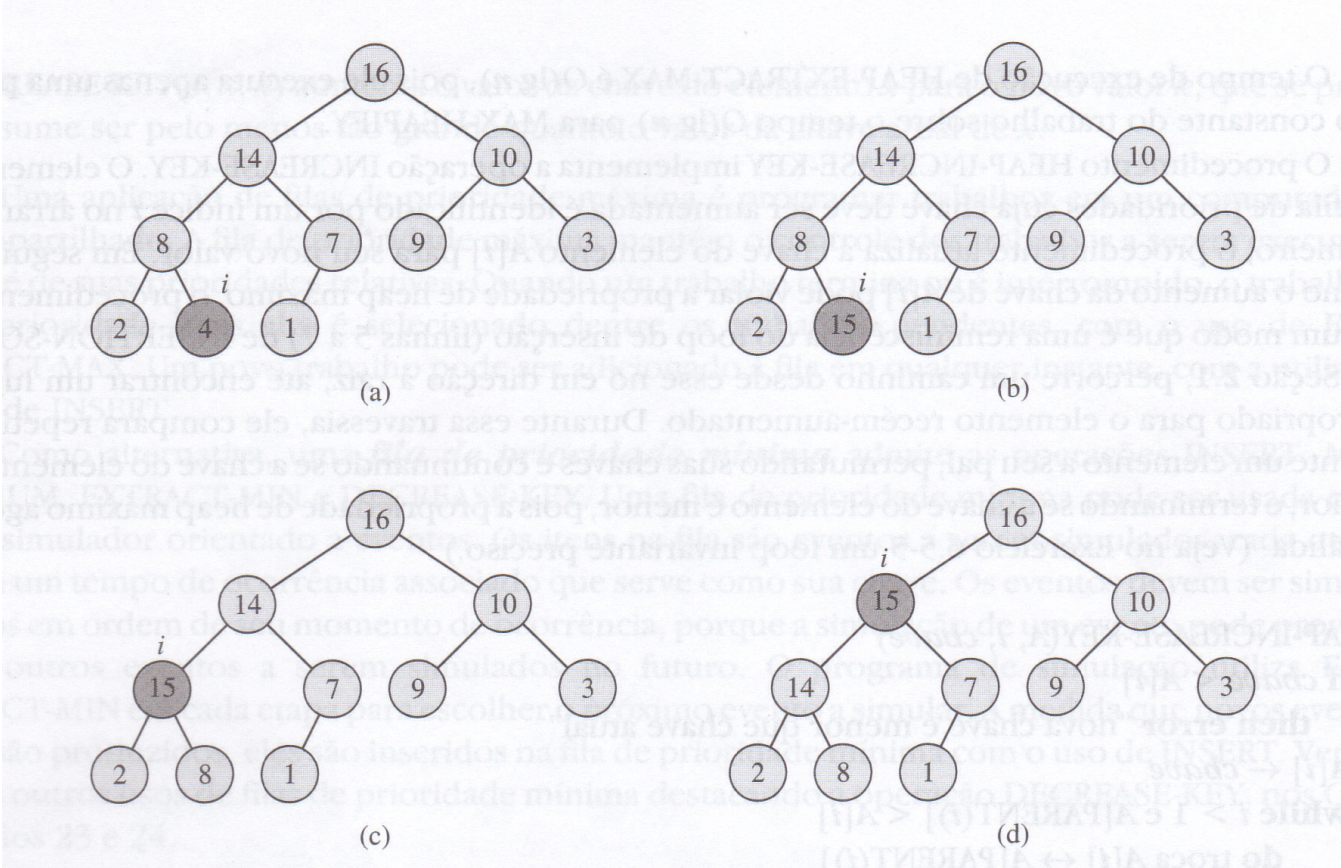
Algoritmos: Alteração e Inserção

```
HEAP-INCREASE-KEY (A, i, chave)
1. se chave < A[i]
2.    então erro 'nova chave é menor que chave atual'
3. A[i] = chave
4. enquanto (i > 1 e A[Pai(i)] < A[i])
5.    troca A[i] com A[Pai(i)]
6.    i = Pai(i)
```

```
HEAP-INSERT (A, chave, n)
1. n = n + 1
2. A[n] = -inf  % recebe um valor muito pequeno
3. HEAP-INCREASE-KEY (A, n, chave)
```

Procedimento HEAP-INCREASE-KEY

- Exemplo: A figura abaixo ilustra a operação do procedimento HEAP-INCREASE-KEY para aumentar a chave do nó i.



Fila de prioridades

- A tabela abaixo mostra a complexidade no tempo para diferentes implementações de fila de prioridades.

Operação	**Lista**	**Lista ordenada**	**Árvore balanceada**	**Heap binário**
Seleção-max	$O(n)$	$O(1)$	$O(logn)$	$O(1)$
Remoção-max	$O(n)$	$O(1)$	$O(logn)$	$O(logn)$
Alteração	$O(1)$	$O(n)$	$O(logn)$	$O(logn)$
Inserção	$O(1)$	$O(n)$	$O(logn)$	$O(logn)$
Construção	$O(n)$	$O(nlogn)$	$O(nlogn)$	$O(n)$

- Percebe-se um *trade-off* na implementação por listas, apesar de serem extremamente simples de codificar.
- Para maior eficiência, usa-se a implementação por *heap*.

Exercícios

1. Ilustre a operação do procedimento HEAP-EXTRACT-MAX sobre o vetor $A = [15\ 13\ 9\ 5\ 12\ 1]$.

2. Ilustre a operação do algoritmo HEAP-INSERT sobre o vetor $A = [15\ 13\ 9\ 5\ 12\ 1]$ ao inserir um elemento com chave 16.

3. A operação HEAP-DECREASE-KEY diminui o valor da chave do elemento x para o novo valor k. Codifique essa operação em tempo $O(logn)$ para um *heap* máximo de n elementos.

Ordenação em tempo linear

- Vimos até agora algoritmos que podem ordenar n números no tempo $O(n\ log_2\ n)$.
- Os algoritmos MergeSort e HeapSort alcançam esse limite superior no pior caso; e o QuickSort o alcança na média.
- Esses algoritmos se baseiam apenas em **comparações** entre os elementos de entrada.
- No entanto, existem algoritmos de ordenação executados em **tempo linear** no limite inferior.
- Para isso, utilizam técnicas diferentes de comparações para determinar a sequência ordenada.

Ordenação em tempo linear

Os seguintes algoritmos de ordenação têm complexidade $O(n)$:

- **CountingSort**: Os elementos a serem ordenados são números inteiros "pequenos", ou seja, inteiros k sendo $O(n)$.
- **RadixSort**: Os valores são números inteiros de comprimento máximo d constante, isto é, independente de n.
- **BucketSort**: Os elementos do vetor de entrada são números reais uniformemente distribuídos sobre um intervalo.

Ordenação por contagem (CountingSort)

- Pressupõe que cada um dos n elementos do vetor de entrada é um inteiro no intervalo de 0 a k, para algum inteiro k.
- Podemos ordenar o vetor contando, para cada inteiro i no vetor, quantos elementos do vetor são menores que i.
- É exatamente o que faz o algoritmo CountingSort!
- Desvantagens: Faz uso de vetores auxiliares e o valor de k deve ser previamente conhecido.

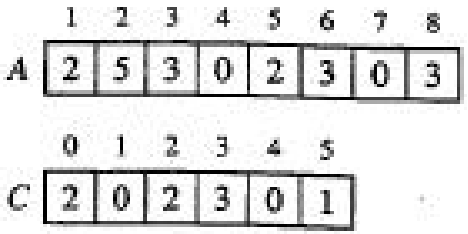
Algoritmo CountingSort

```
COUNTING-SORT (A, B, n, k)
1. para i = 0 até k faça
2.     C[i] = 0
3. para j = 1 até n faça
4.     C[A[j]] = C[A[j]] + 1
// Agora C[i] contém o número de elementos iguais a i
5. para i = 1 até k faça
6.     C[i] = C[i] + C[i - 1]
// Agora C[i] contém o número de elementos <= i
7. para j = n até 1 faça
8.     B[C[A[j]]] = A[j]
9.     C[A[j]] = C[A[j]] - 1
```

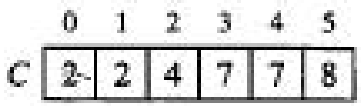
Claramente, a complexidade do COUNTING-SORT é $O(n + k)$.
Quando k é $O(n)$, ele tem complexidade $O(n)$.

Algoritmo CountingSort

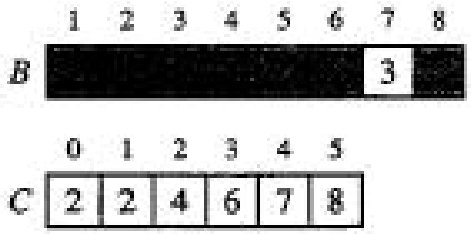
- Exemplo: Operação do COUNTING-SORT sobre o vetor de entrada $A = [2\ 5\ 3\ 0\ 2\ 3\ 0\ 3]$.



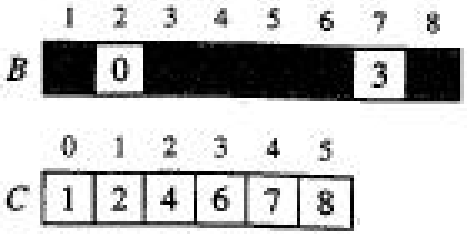
(a)



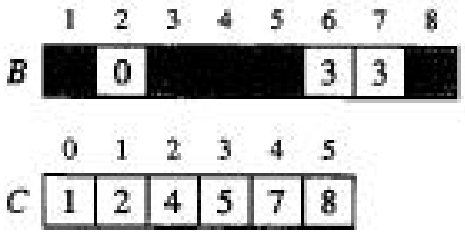
(b)



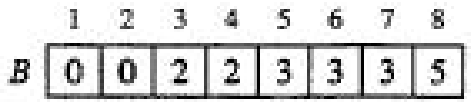
(c)



(d)



(e)



(f)

Considerações sobre o CountingSort

- A ordenação por contagem (CountingSort) supera o limite inferior de $\Omega(n\ log_2\ n)$, pois não ordena por comparação.
- É um método que utiliza os valores reais dos elementos para efetuar a indexação em um arranjo.
- Se o uso de memória auxiliar for muito limitado, é melhor usar um algoritmo de ordenação por comparação **local**.
- É um **algoritmo estável**: Elementos iguais ocorrem no vetor ordenado na mesma ordem em que aparecem na entrada.
- O MergeSort também é um algoritmo de ordenação estável, já o QuickSort e o HeapSort não são.

Exercícios

1. Ilustre a operação do algoritmo COUNTING-SORT sobre o arranjo $A = [6\ 0\ 2\ 0\ 1\ 3\ 4\ 6]$.

2. Prove que o algoritmo COUNTING-SORT é estável.

3. Suponha que o cabeçalho do laço "para" na linha 7 do procedimento COUNTING-SORT seja reescrito:

```
7. para j = 1 até n faça
```

   a. Mostre que o algoritmo ainda funciona corretamente.

   b. O algoritmo modificado é estável?

Ordenação da base (RadixSort)

- Considere o problema de ordenar o vetor A sabendo que todos os seus n elementos inteiros tem d dígitos.
- Por exemplo, os elementos do vetor A podem ser CEPs, ou seja, inteiros de 8 dígitos.
- O método RadixSort ordena os elementos do vetor dígito a dígito, começando pelo menos significado.
- Para que o RadixSort funcione corretamente, ele deve usar um método de ordenação estável, como o CountingSort.

Algoritmo RadixSort

- O algoritmo abaixo implementa o método RadixSort.

```
RADIX-SORT (A, n, d)
1. para i = 1 até d faça
2.    Ordene A[1..n] pelo i-ésimo dígito
       usando um método estável
```

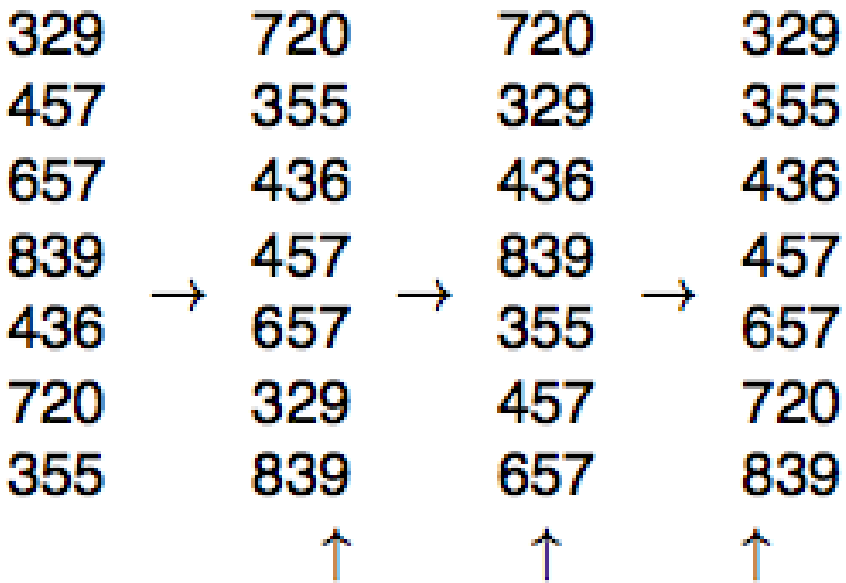
- A complexidade do RADIX-SORT depende da complexidade do algoritmo estável usado para ordenar cada dígito.

- Se o algoritmo estável for, por exemplo, o COUNTING-SORT, obtém-se a complexidade $O(d(n+k))$.

- Quando k é $O(n)$ e d é $O(1)$, o RADIX-SORT possui uma complexidade linear em $O(n)$.

Algoritmo RadixSort

- Exemplo: Operação do RADIX-SORT sobre o vetor de entrada $A = [329\ 457\ 657\ 839\ 436\ 720\ 355]$.



Exercícios

- Ilustre a operação do algoritmo RADIX-SORT sobre o arranjo $A = [COW\ DOG\ SEA\ RUG\ ROW\ MOB\ BOX]$.

- Sabemos que o algoritmo de ordenação por comparação MergeSort tem complexidade no tempo $\Theta(n\ log_2\ n)$.

  Mostre que o método RadixSort pode ser mais vantajoso que o MergeSort quando $d < log_2\ n$.

- Suponha que desejamos ordenar um vetor de 2^{20} números de 64 bits usando o algoritmo RadixSort tendo o CountingSort como método estável.

  a. Explique o funcionamento e a complexidade do algoritmo.

  b. Qual seria o tamanho dos vetores auxiliares?

Considerações sobre o RadixSort

- Se o maior valor a ser ordenado for $O(n)$, então, $O(log_2\ n)$ é uma estimativa para a quantidade de dígitos dos números.
- A vantagem do RadixSort fica evidente quando interpretamos os dígitos de forma mais geral que a base decimal ([0..9]).
- Suponha que desejemos ordenar um conjunto de 2^{20} números de 64 bits. Então, o MergeSort faria cerca de $nlogn = 2^{20}x20$ comparações e usaria um vetor auxiliar de tamanho 2^{20}.
- Agora, suponha que interpretamos cada número como tendo 4 dígitos em base $k = 2^{16}$. Então, o RadixSort faria cerca de $d(n + k) = 4(2^{20} + 2^{16})$ operações. Mas, note que usaríamos dois vetores auxiliares, de tamanhos 2^{16} e 2^{20}.

Ordenação por balde (BucketSort)

- Funciona em tempo linear quando a entrada é gerada a partir de uma distribuição uniforme sobre um intervalo.
- Primeiramente, divide o intervalo que vai de 0 até k em n subintervalos, ou buckets, de mesmo tamanho.
- Depois, distribui os n números do vetor de entrada entre os buckets. Na prática, esses buckets são listas encadeadas.
- Em seguida, os elementos de cada lista são ordenados por um método qualquer.
- Finalmente, as listas ordenadas são concatenadas em ordem crescente, gerando uma lista final ordenada.

Algoritmo BucketSort

- O algoritmo abaixo implementa o método BucketSort.

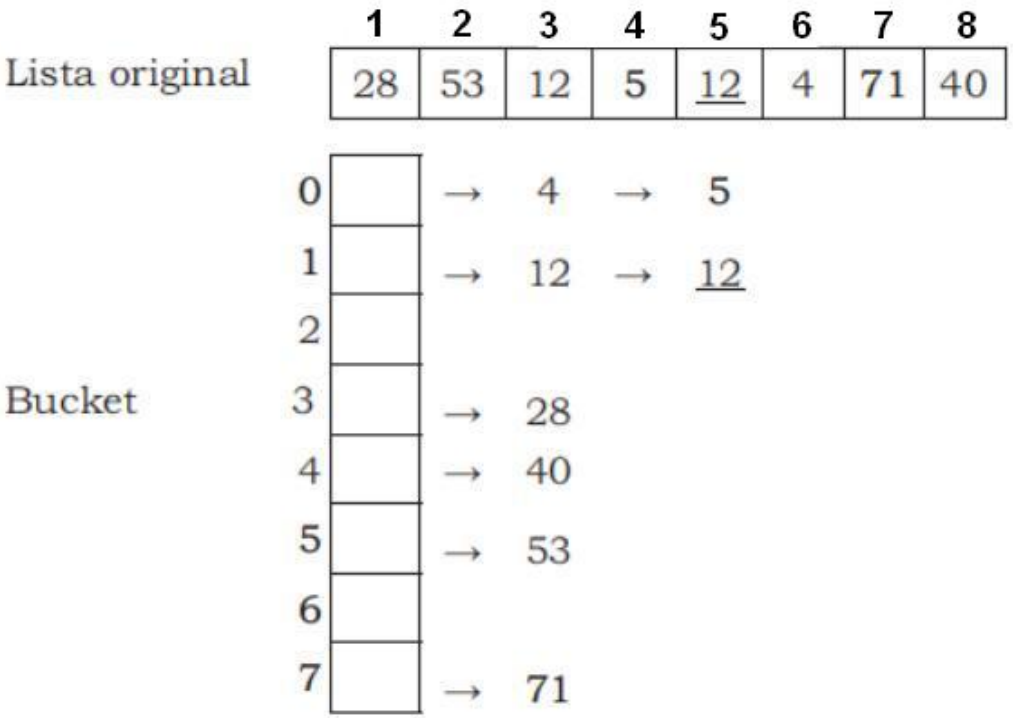
```
BUCKET-SORT (A, n, k)
1. para i = 0 até n - 1 faça
2.   B[i] = 0
// Vetor auxiliar de listas encadeadas
3. para i = n até 1 faça
4.   Insira A[i] no início da
      lista B[(A[i] * n)/(k + 1)]
5. para i = 0 até n - 1 faça
6.   Ordene a lista B[i]
7. Concatene as listas B[0], ..., B[n - 1]
```

- Exemplo: Se o vetor de entrada tem 8 elementos e o maior deles é 71, obtém-se 8 intervalos: [0,9[, [9,18[, ..., [63,72[.

Algoritmo BucketSort

- Intuitivamente, espera-se que o número de elementos seja pequeno e aproximadamente igual em todas as listas.

- Portanto, o algoritmo BUCKET-SORT funciona no tempo esperado linear $O(n)$.



Exercícios

1. Ilustre a operação do algoritmo BUCKET-SORT sobre o arranjo $A = [79\ 13\ 16\ 64\ 39\ 20\ 89\ 53\ 71\ 42]$.

2. Apresente a operação do algoritmo BUCKET-SORT sobre o vetor $A = [15\ 13\ 6\ 9\ 180\ 26\ 12\ 4\ 20\ 71]$. Depois, explique o tempo de execução esperado para essa operação.

3. Prove que o algoritmo BUCKET-SORT é estável.

4. É preferível usar o BucketSort a um algoritmo de ordenação baseado em comparação, como o QuickSort, por exemplo? Quais seriam os prós e contras?