Algoritmos Estrutura de Dados

Marcelo Lobosco

DCC/UFJF

Algoritmos de Ordenação

Parte 2 – Aula 06

Agenda

Algoritmos de Ordenação

- Quicksort
- Métodos de Ordenação em Tempo Linear

Heapsort

- Combina melhores atributos dos dois algoritmos de ordenação anteriores
  - Ordena localmente como o ordenação por inserção
    - Não necessita de estruturas auxiliares, como a ordenação por intercalação
  - Tem tempo de execução baixo, como a ordenação por intercalação
    - O(n.lgn)

Heapsort

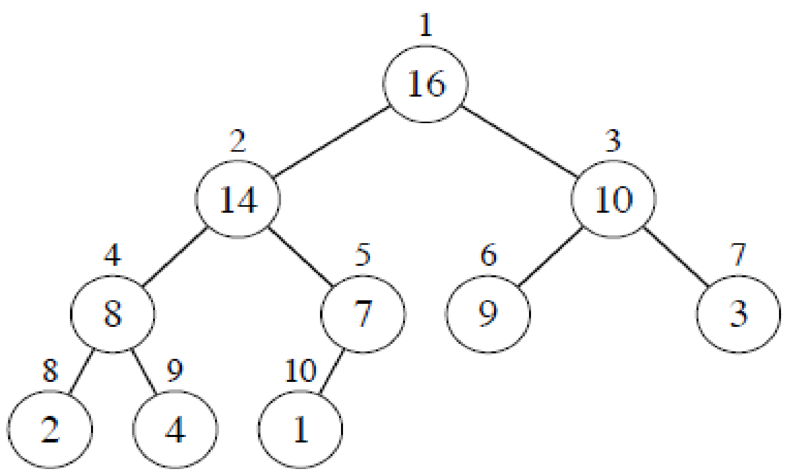
- Algoritmo utiliza estrutura de dados conhecida como heap
- Heap é um vetor que pode ser visto como uma árvore binária praticamente completa
  - Cada nó da árvore corresponde a um elemento do vetor
  - Árvore completamente preenchida em todos os níveis
    - Exceto talvez no mais baixo, sempre preenchido da esquerda para a direita até certo ponto

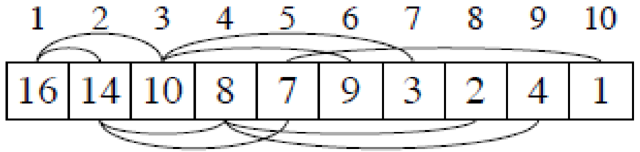
Heapsort

- Vetor A que representa uma heap tem dois atributos
  - comprimento[A]: número de elementos no vetor
  - tamanho-da-heap[A]: número de elementos da heap armazenado dentro do vetor A
  - Ou seja, tamanho-da-heap[A] <= comprimento[A]
- Raiz da árvore em A[1]

Heapsort

- Dado índice i de um nó, índices de seu pai e filhos pode ser calculado facilmente
  - PARENT(i) = piso (i/2)
  - LEFT(i) = 2i
  - RIGHT(i) = 2i+1





Heapsort

- Na maioria dos computadores, operações podem ser feitas com única instrução
  - PARENT(i): desloca i um bit para a direita. P.ex.: 4 (100). Pai: 2 (10)
  - LEFT(i): desloca i um bit para a esquerda. P.ex.: 4 (100). Esquerda: 8 (1000)
  - RIGHT(i): desloca i um bit para a esquerda, colocando o bit 1 como bit menos significativo. P.ex.: 4 (100). Direita: 9 (1001)
  - Por questões de desempenho, implementados como macros

Heapsort

- Dois tipos de heaps
  - Heap máximo: A[PARENT(i)]>=A[i]
    - Ou seja, valor de um nó no máximo o valor de seu pai
    - Maior elemento armazenado na raiz
  - Heap mínimo: organizado de modo inverso
    - A[PARENT(i)]<=A[i]
    - Menor elemento da heap na raiz
- No algoritmo heapsort, usamos heaps máximos
- Heaps mínimos empregados em problemas de prioridades

Heapsort

- Altura de um nó
  - Número de arestas no caminho descendente simples mais longo até uma folha
  - Como heap baseada em árvore binária completa, altura é lg n
  - Veremos que operações executadas em um tempo proporcional a sua altura
- Altura de uma heap: altura de sua raiz

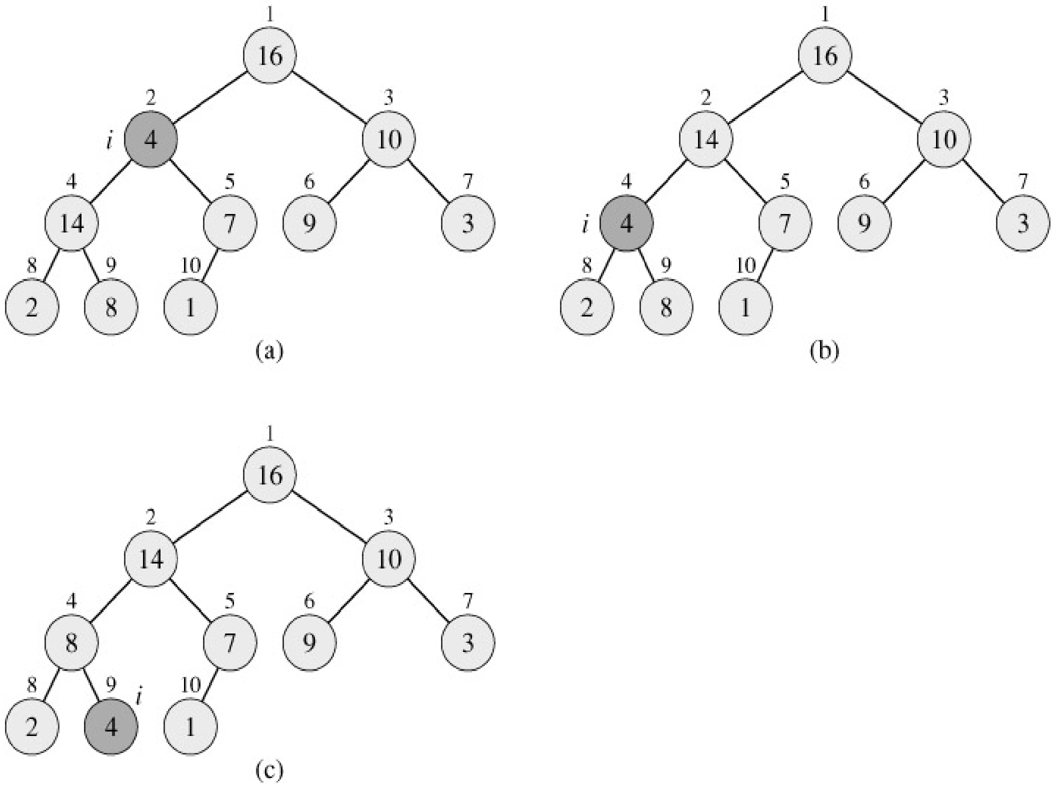
Heapsort

- Manutenção da propriedade da heap
  - Sub-rotina MAX-HEAPIFY
  - Entradas: vetor A, índice i e tamanho-da-heap(A)
  - Quando chamada, supomos que árvores binárias com raízes em LEFT(i) e RIGHT(i) são heaps máximos, mas que A[i] pode ser menor que seus filhos
  - Idéia da função: valor A[i] possa descer pela árvore, de tal forma que subárvore com raiz no índice i se torne heap máximo

Heapsort

- Manutenção da propriedade da heap (cont.)

```
MAX-HEAPIFY(A, i, n)
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
   then largest ← l
   else  largest ← i
if r ≤ n and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
          MAX-HEAPIFY(A, largest, n)
```



Heapsort

- Tempo de execução de MAX-HEAPIFY
  - Tempo constante para corrigir relacionamentos entre i e os seus filhos: Θ (1)
  - Tempo para executar MAX-HEAPIFY em subárvore com raiz em um dos filhos de I
  - Quantos elementos tem subárvore com raiz em um dos filhos de i? No pior caso, quando última linha da árvore está exatamente meio cheia, 2n/3 elementos
  - $T(n) <= T(2n/3) + \Theta(1)$
  - Pelo caso 2 do teorema mestre, $T(n) = \Theta(\lg n)$

Heapsort

- Construção de uma heap
  - Folhas da árvore estão em A[(piso(n/2)+1)..n]
  - MAX-HEAPIFY utilizado nos demais nós
  - Subrotina BUILD-MAX-HEAP
  - Parâmetros A e n: vetor e tamanho da heap

```
BUILD-MAX-HEAP(A, n)
for i ← ⌊n/2⌋ downto 1
    do MAX-HEAPIFY(A, i, n)
```

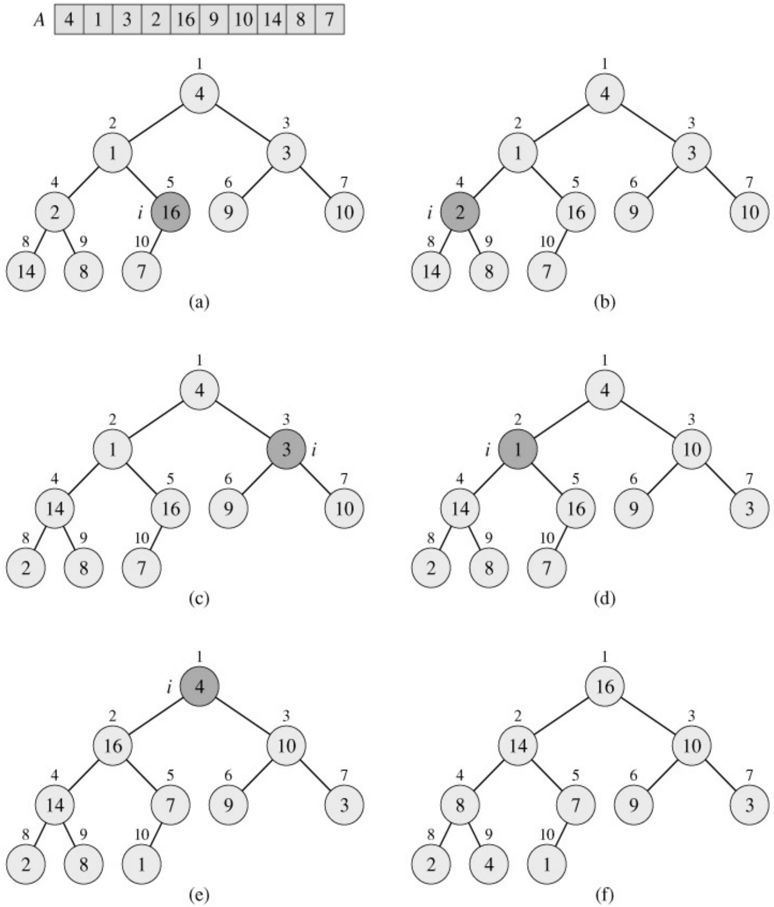
Heapsort

- Construção de uma heap (cont.)

BUILD-MAX-HEAP(A, n)

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

  do MAX-HEAPIFY(A, i, n)



Heapsort

- Prova da corretude da construção de uma heap
  - Invariante: no início de cada iteração, nó i+1, i+2, …, n é a raiz de um heap máximo
  - Inicialização: Antes da primeira iteração do loop, i = piso (n/2). Nós posteriores são folha, portanto raiz de heap máximo trivial
  - Manutenção: Filhos de i são raízes de heaps máximos (numerados com valores maiores que i), o que é a condição para chamada a MAX-HEAPIFY. MAX-HEAPIFY preserva propriedade de que i+1, .., n são raízes de heaps máximos. Decrementar i reestabelece loop invariante para próxima iteração
  - Término: Quando i = 0, pelo loop invariante, cada nó 1,...,n é a raiz de um heap máximo, sendo que nó 1 é raiz

Heapsort

- Construção de uma heap (cont.)
  - Custo de BUILD-MAX-HEAP = O(n).O(lg n) = O (nlg n) (correto, mas assintoticamente não restrito)
  - Custo de BUILD-MAX-HEAP = O(n)

Heapsort

- Algoritmo de ordenação heapsort
  - Algoritmo simples: heap máximo contruído inicialmente
  - Maior valor encontra-se no nó raiz
  - Este é retirado da raiz, e trocado por elemento na última posição do vetor
  - MAX-HEAPIFY utilizado para reestabelecer a propriedade da heap
  - Processo repetido até se chegar a posição inicial do vetor

Heapsort

- Algoritmo de ordenação heapsort

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i \leftarrow n$ **downto** 2
    do exchange $A[1] \leftrightarrow A[i]$
        MAX-HEAPIFY$(A, 1, i - 1)$

Heapsort

- Complexidade do algoritmo de ordenação heapsort
  - BUILD-MAX-HEAP: O(n)
  - Loop executado n-1 vezes
  - Operação de troca: O(1)
  - Operação MAX-HEAPIFY: O(lg n)
  - Custo total: O(n.lg n)

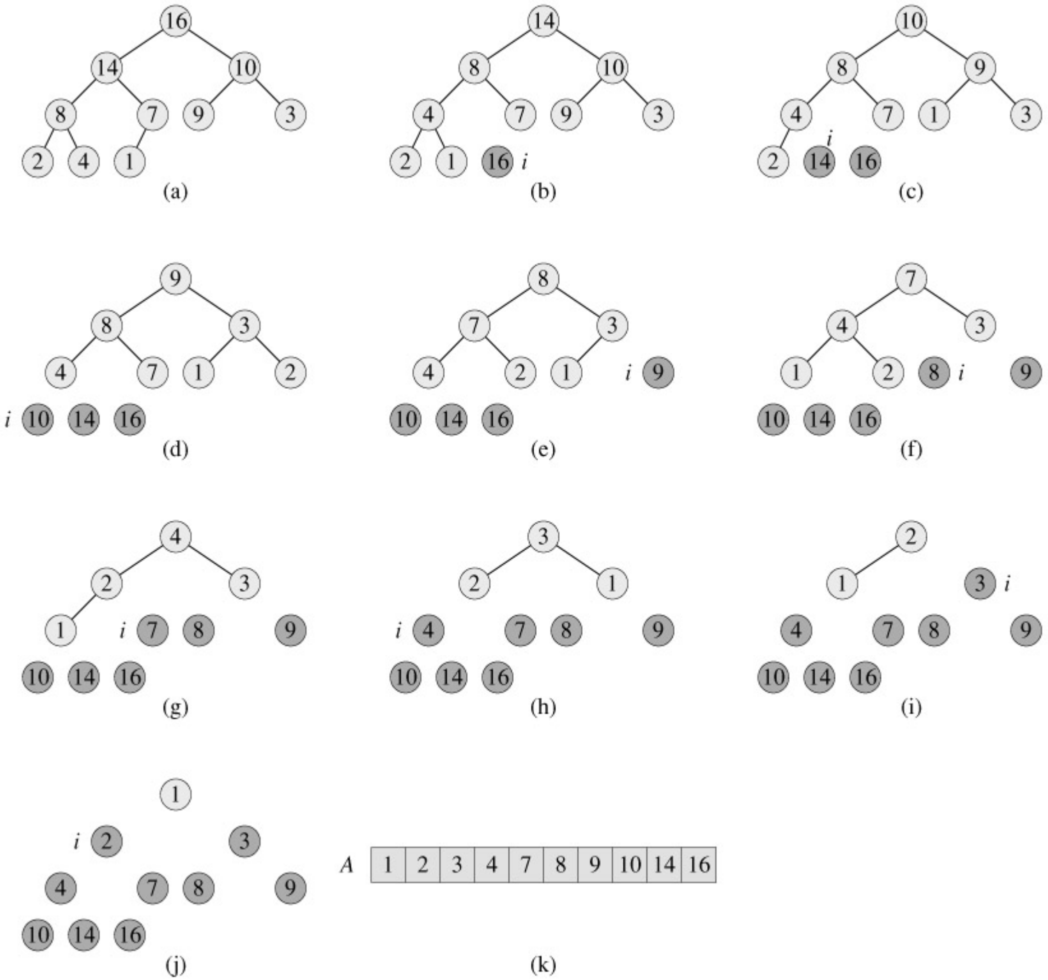
```
HEAPSORT(A, n)
BUILD-MAX-HEAP(A, n)
for i ← n downto 2
    do exchange A[1] ↔ A[i]
       MAX-HEAPIFY(A, 1, i − 1)
```

Heapsort

- Algoritmo de ordenação heapsort

HEAPSORT(A, n)

BUILD-MAX-HEAP(A, n)
for $i \leftarrow n$ **downto** 2
    do exchange $A[1] \leftrightarrow A[i]$
        MAX-HEAPIFY($A, 1, i-1$)



Fila de Prioridades

- Algoritmo de ordenação heapsort normalmente superado na prática por quicksort
- Ainda assim estrutura de dados tem utilidade enorme
- Um dos principais usos: fila de prioridade
- Dois tipos de filas: prioridade máxima e mínima
- Focaremos na prioridade máxima, baseados em heaps máximos

Fila de Prioridades

- Fila de prioridades: estrutura de dados para manutenção de um conjunto S de elementos
  - Cada elemento com valor associado: chave
- Fila admite seguintes operações:
  - INSERT (S,x): insere elemento x no conjunto S
  - MAXIMUM (S): retorna elemento S com maior chave
  - EXTRACT-MAX(S): remove e retorna elemento S com maior chave
  - INCREASE-KEY(S,x,k): aumenta valor da chave do elemento x para novo valor k

Fila de Prioridades

- Aplicação: selecionar trabalho a ser executado em um sistema de tempo compartilhado baseado em prioridades
  - Quando processo termina ou é interrompido, processo de prioridade mais alta selecionado com EXTRACT-MAX
  - Novo processo adicionado com INSERT

Fila de Prioridades

- Implementação das operações:

HEAP-MAXIMUM(A)

return $A[1]$

HEAP-EXTRACT-MAX(A, n)

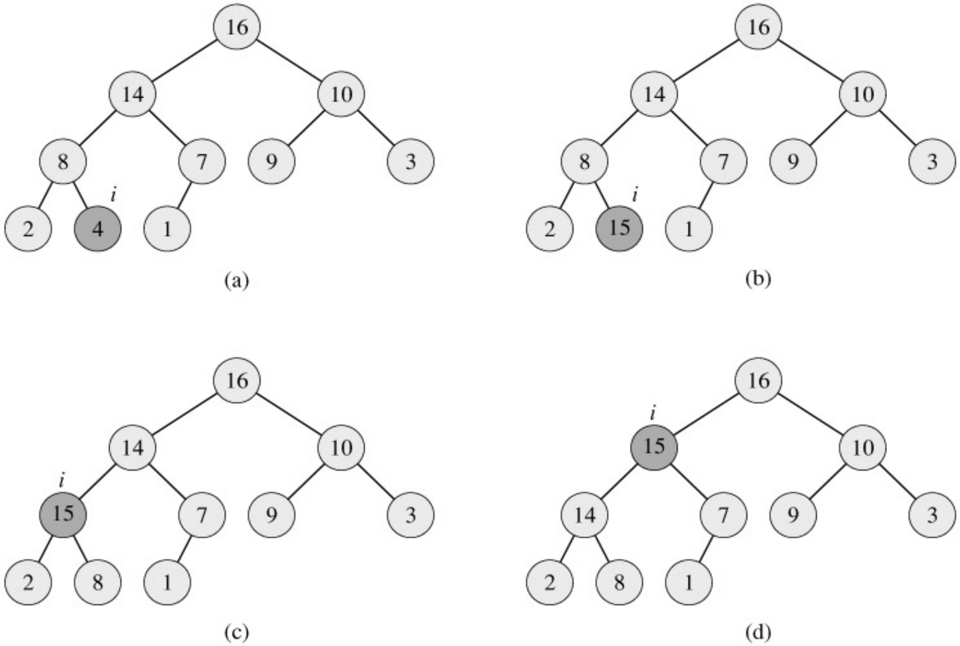
if $n < 1$
   then error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[n]$
MAX-HEAPIFY$(A, 1, n - 1)$ $\quad \triangleright$ remakes heap
return max

Fila de Prioridades

- Implementação das operações:

HEAP-INCREASE-KEY(A, i, key)

if $key < A[i]$
  then error "new key is smaller than current key"
$A[i] \leftarrow key$
while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
  do exchange $A[i] \leftrightarrow A[\text{PARENT}(i)]$
    $i \leftarrow \text{PARENT}(i)$



Fila de Prioridades

- Implementação das operações:

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

$A[n+1] \leftarrow -\infty$

$\text{HEAP-INCREASE-KEY}(A, n+1, key)$

Quicksort

- Baseia-se no paradigma dividir e conquistar
- Tempo de execução no pior caso: $\Theta(n^2)$
- Na prática, melhor opção para ordenação a sua eficiência na média
  - $\Theta(n \lg n)$
  - Vantagem da ordenação local
    - Sem uso de estruturas de dados auxiliares

Descrição do Quicksort

- Para ordenar subarranjo A[p..r]
  - Dividir: arranjo A[p..r] particionado em A[p..q-1] e A[q+1..r], tais que cada elemento de A[p..q-1] é menor ou igual a A[q], e este é menor ou igual a cada elemento de A[q+1..r]. Índice q calculado na operação de particionamento
  - Conquistar: Subarranjos A[p..q-1] e A[q+1..r] ordenados por chamadas recursivas a quicksort
  - Combinar: Como subarranjos ordenados localmente, arranjo inteiro A[p..r] está ordenado

Implementação do Quicksort

- Chamada inicial: QUICKSORT(A,1,n)

$\textsc{Quicksort}(A, p, r)$

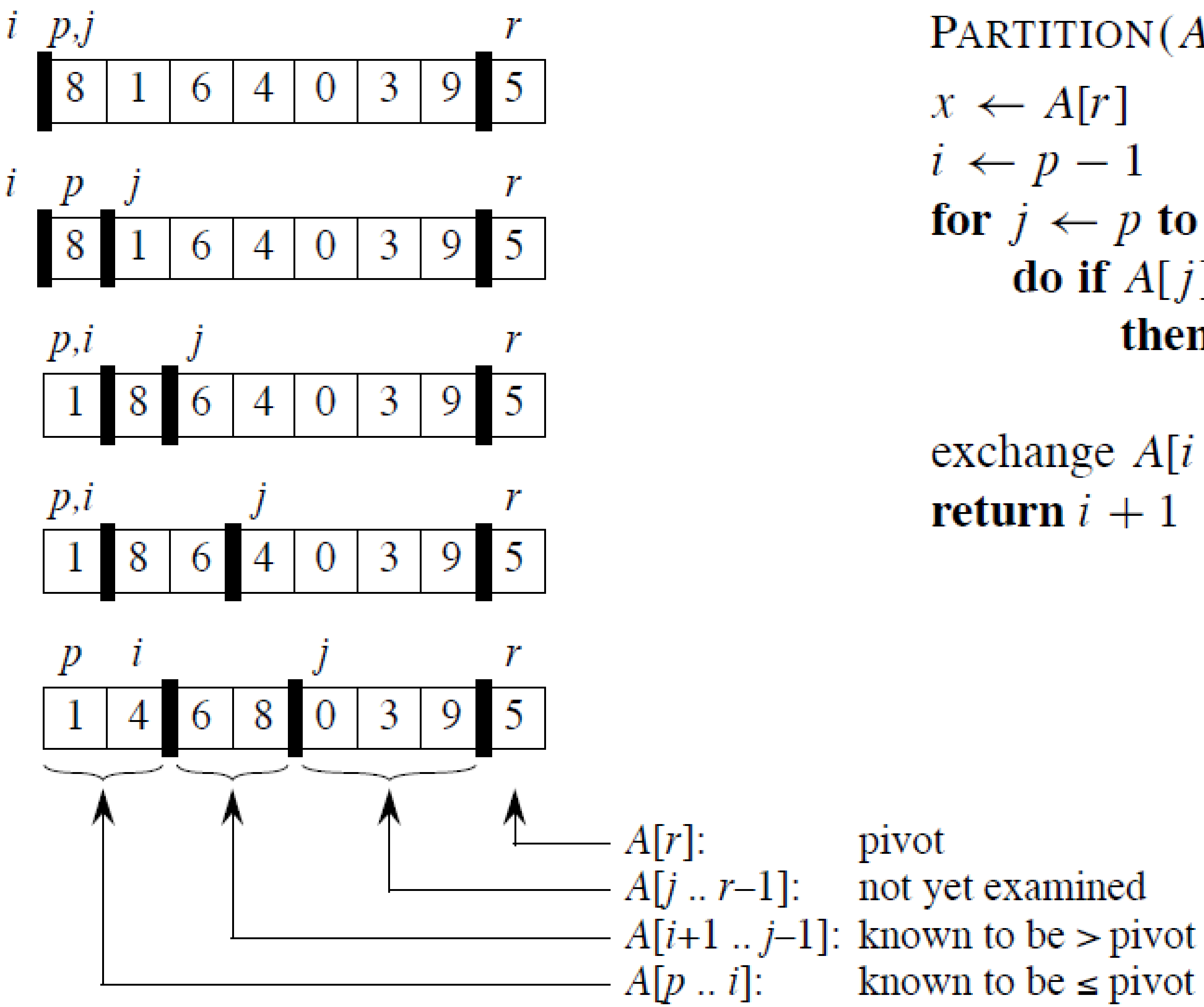
if $p < r$
  then $q \leftarrow \textsc{Partition}(A, p, r)$
    $\textsc{Quicksort}(A, p, q - 1)$
    $\textsc{Quicksort}(A, q + 1, r)$

Implementação do Quicksort

- Chave para Quicksort é procedimento PARTITION
  - Reorganiza subarranjo A[p..r] localmente
  - x=A[r] chamado de pivô: elemento que serve de base para particionamento do subarranjo

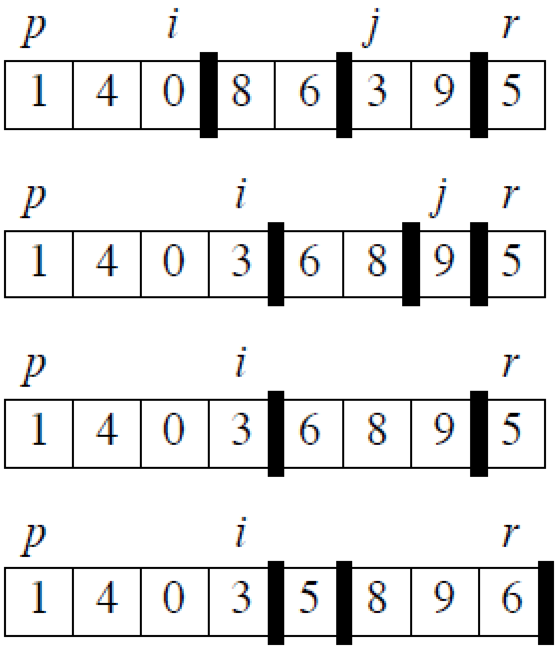
```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
    do if A[j] ≤ x
        then i ← i + 1
            exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Implementação do Quicksort




```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
     do if A[j] ≤ x
          then i ← i + 1
               exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Implementação do Quicksort




```
PARTITION(A, p, r)
x ← A[r]
i ← p − 1
for j ← p to r − 1
     do if A[j] ≤ x
           then i ← i + 1
                exchange A[i] ↔ A[j]
exchange A[i + 1] ↔ A[r]
return i + 1
```

Implementação do Quicksort

- No início de cada iteração do loop das linhas 3 a 6, para qualquer índice k do arranjo
  - Se $p <= k <= i$, então $A[k] <= x$
  - Se $i+1 <=k <= j-1$, então $A[k] > x$
  - Se $k = r$, então $A[k] = x$
  - Índices entre j e r-1 não cobertos por quaisquer casos: valores não tem relacionamento estabelecido com pivô
- Vamos mostrar a corretude do algoritmo

Corretude do Algoritmo de Partição

- Inicialização
  - Antes da primeira iteração, i=p-1 e j=p. Logo não há qualquer valor entre p e i, bem como nenhum valor entre i+1 e j-1. Assim duas primeiras condições do loop invariante satisfeitas; terceira condição satisfeita pela primeira atribuição do código

PARTITION(A, p, r)

$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ **to** $r - 1$
    do if $A[j] \leq x$
        then $i \leftarrow i + 1$
            exchange $A[i] \leftrightarrow A[j]$
exchange $A[i+1] \leftrightarrow A[r]$
return $i + 1$

Corretude do Algoritmo de Partição

- Manutenção
  - Dois casos a considerar, dependendo do teste da quarta linha.
  - Se A[j]>x, basta incrementar j: condição 2 válida para A[j-1]
  - Se A[j] <=x, i é primeiro incrementado, A[i] e A[j] são permutados e então j é incrementado. Por conta da troca, condição 1 satisfeita (A[i] <= x). Do mesmo modo, A[j-1]>x, pois, pelo loop invariante, A[i+1] (antes do incremento de i) é maior que x

Corretude do Algoritmo de Partição

- Término
  - No fim do algoritmo, j =r
  - Logo toda entrada em um dos arranjos descritos pelo invariante: menores ou iguais a x, maiores que x, e conjunto unitário contendo x

Desempenho do Algoritmo Quicksort

- Desempenho de particionamento: $\Theta(n)$
- Já desempenho de quicksort depende de quais elementos são utilizados para balancear
  - Se particionamento balanceado, tão rápido quando ordenação por intercalação
  - Se particionamento não balanceado, tão lento quanto ordenação por inserção

Desempenho do Algoritmo Quicksort

- Pior caso ocorre quando rotina de particionamento produz dois subproblemas, um com n-1 elementos, e o outro com 0 elementos
- Vamos supor que este desbalanceamento surge em cada chamada recursiva
  - $T(0) = \Theta(1)$, já que função simplesmente retorna
  - $T(n) = T(n-1) + T(0) + \Theta(n) = T(n-1) + \Theta(n)$, o que nos leva, por série aritmética, a $\Theta(n^2)$

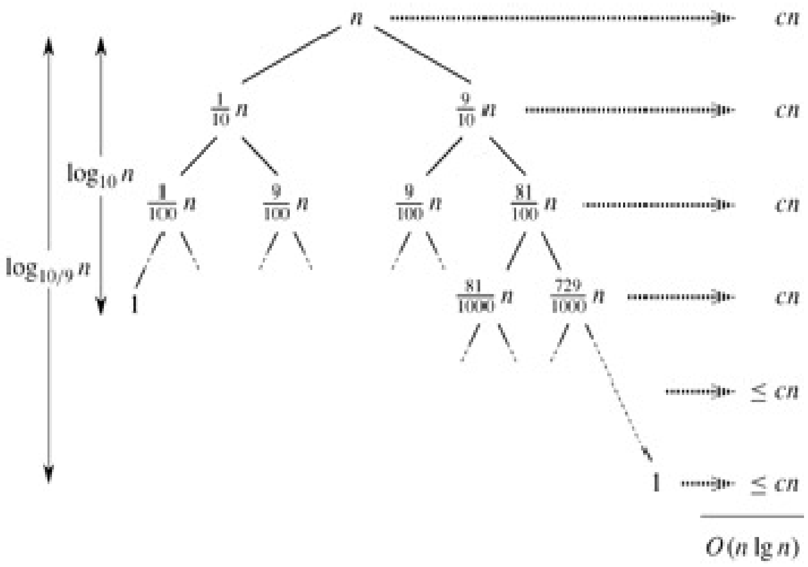
Desempenho do Algoritmo Quicksort

- Melhor caso ocorre quando rotina de particionamento produz dois subproblemas com tamanho igual a aproximadamente metade do arranjo
  - $T(n) = 2T(n/2) + \Theta(n)$, que, pelo caso 2 do teorema mestre, nos leva a $\Theta(n \lg n)$

Desempenho do Algoritmo Quicksort

- Tempo de execução do caso médio mais próximo do melhor caso do que pior caso
- Por exemplo, suponha algoritmo de particionamento que sempre produz divisão de 9 para 1
  - A principio parece desequilibrada...
  - $T(n) <= T(9n/10) + T(n/10) + \Theta(n)$
  - Vamos analisar recursão...

Desempenho do Algoritmo Quicksort



Desempenho do Algoritmo Quicksort

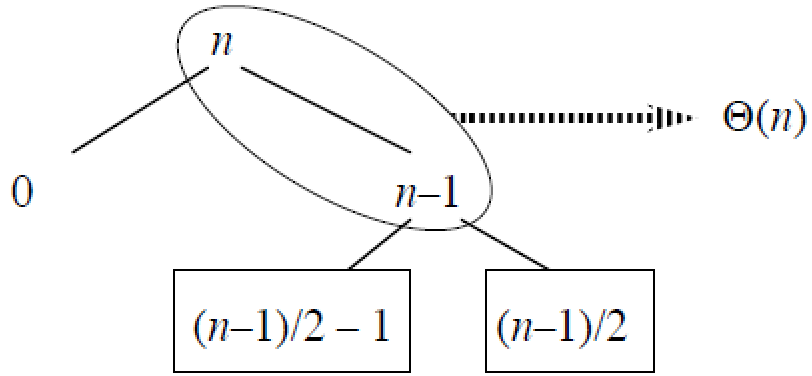
- Temos $\log_{10} n$ níveis cheios e $\log_{10/9} n$ níveis não completamente cheios
- Para a notação assintótica, a base do log não importa, desde que naturalmente seja uma constante
  - Podemos usar base 2
  - Qualquer uso de constantes nos daria lg n níveis
  - Como o custo de cada nível é menor ou igual a n, o custo total é portanto O(n lg n)

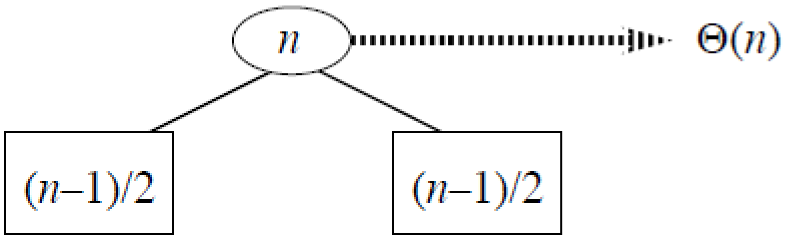
Desempenho do Algoritmo Quicksort

- Intuição para o caso médio
  - Vamos supor que arranjo da entrada é aleatório
  - Improvável que particionamento sempre ocorra do mesmo modo em todo nível
    - Algumas divisões equilibradas e outras desequilibradas
  - No caso médio, mistura de boas e más divisões por todos os níveis da árvore
  - Para fins de intuição, vamos supor intercalação, onde boa divisão seja o melhor caso, e má divisão seja o pior caso

Desempenho do Algoritmo Quicksort

- Intuição para o caso médio





Desempenho do Algoritmo Quicksort

- Combinação da divisão ruim com divisão boa produz três subarranjos de tamanho 0, (n-1)/2 -1 e (n-1)/2
- Custo combinado: Θ (n) + Θ (n-1) = Θ (n)
- Intuitivamente, custo da divisão ruim absorvido pela divisão boa
- Tempo de execução médio semelhante ao tempo de execução para divisões boas
- Diferença nas constantes, que são ocultadas pela notação O

Limites Inferiores para Ordenação

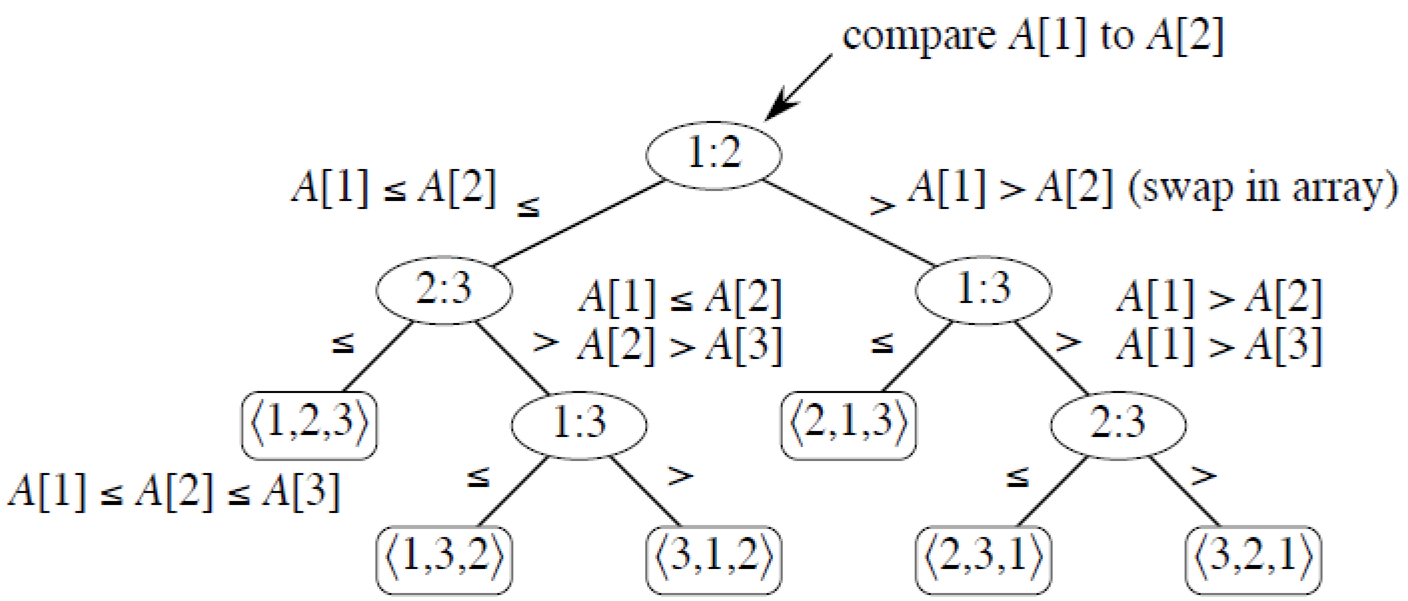
- Algoritmos apresentados até o momento compartilham propriedade interessante: utilizam comparação entre os elementos de entrada para estabelecer ordenação
- Algoritmos de ordenação por comparação
- Vamos supor que todos os elementos da entrada são distintos
  - Comparações <=, >=, >, < equivalentes para estabelecer ordem relativa de duas entradas a_i e a_j

Limites Inferiores para Ordenação

- Árvores de decisão
  - Ordenações de comparação podem ser vistas de modo abstrato como árvores de decisão
  - Trata-se de uma árvore binária cheia que representa as comparações executadas por algoritmo de ordenação
  - Cada nó interno anotado por i:j, para algum i e j no intervalo 1<=i,j <= n, onde n é o número de elementos na sequência de entrada
  - Cada folha anotada por uma permutação (π(1),π(2),...,π(n))

Limites Inferiores para Ordenação

- Árvore de decisão



Limites Inferiores para Ordenação

- Execução do algoritmo de ordenação corresponde em traçar caminho da raiz até a folha
  - Em cada nó interno, feita comparação $a_i <= a_j$
  - Subárvore da esquerda determina comparações subsequentes para $a_i <= a_j$, e subárvore da direita para $a_i > a_j$
  - Quando chegamos a folha, algoritmo de ordenação estabelece ordem
  - Cada uma das n! permutações de n aparece como uma das folhas da árvore de decisão

Limites Inferiores para Ordenação

- Comprimento do caminho mais longo entre raiz e folha representa o número de comparações do pior caso do algoritmo de ordenação
  - Logo número de comparações igual a altura da árvore de decisão do algoritmo
- Vamos então usar essas informações para mostrar que algoritmos de ordenação por comparação exigem Ω (n. lg n) comparações no pior caso

Limites Inferiores para Ordenação

- Considere árvore de decisão de altura h
  - Cada uma das n! permutações aparece como alguma folha
  - Como árvore binária de altura h não tem mais do que 2^h folhas, temos que $n! <= 2^h$
  - Usando-se logaritmos, temos $h >= \lg (n!) = \Omega (n \lg n)$

Limites Inferiores para Ordenação

- Veja que heapsort e mergesort são ordenações por comparação assintoticamente ótimas
  - Os $O(n \lg n)$ limites superiores sobre os tempos de execução para cada algoritmo correspondem ao limite inferior do pior caso de qualquer algoritmo de ordenação por comparação, ou seja, $\Omega(n \lg n)$

Ordenação por Contagem

- Idéia é determinar, para cada elemento de entrada x, número de elementos menores que x
- Infomação usada para inserir elemento x diretamente em sua posição de saída
- Utiliza arranjo B para armazenar saída ordenada e arranjo C como temporário
  - Arranjo C tem k elementos, onde k é o maior elemento do arranjo

Ordenação por Contagem

COUNTING-SORT(A, B, n, k)

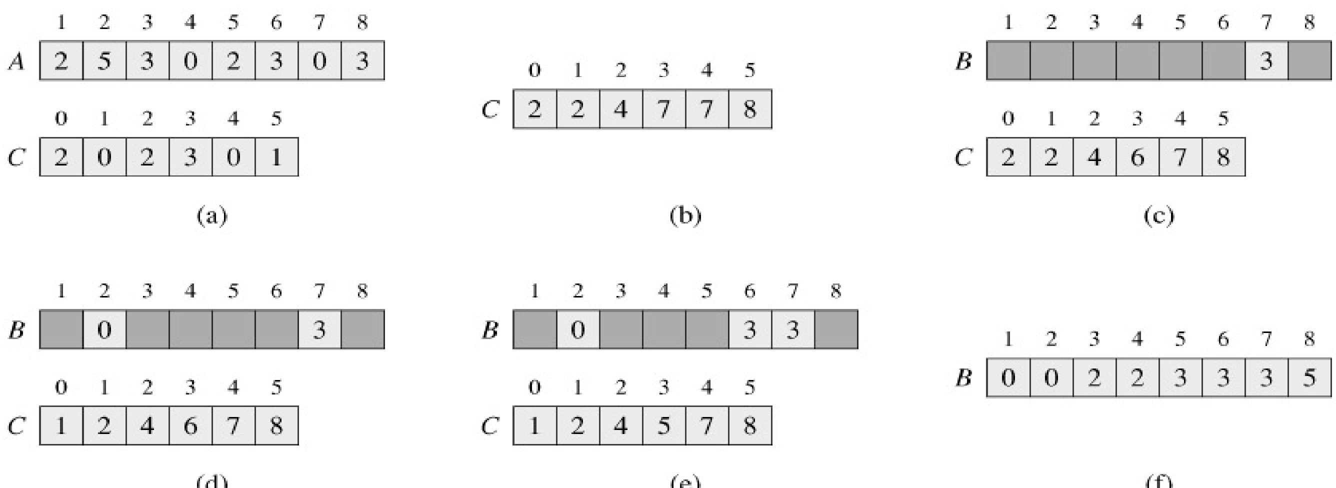
for $i \leftarrow 0$ **to** k
  do $C[i] \leftarrow 0$
for $j \leftarrow 1$ **to** n
  do $C[A[j]] \leftarrow C[A[j]] + 1$ ← número de elementos iguais a j
for $i \leftarrow 1$ **to** k
  do $C[i] \leftarrow C[i] + C[i-1]$ ← número de elementos menores ou iguais a j
for $j \leftarrow n$ **downto** 1
  do $B[C[A[j]]] \leftarrow A[j]$
    $C[A[j]] \leftarrow C[A[j]] - 1$

Ordenação por Contagem

```
COUNTING-SORT(A, B, n, k)
for i ← 0 to k
    do C[i] ← 0
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1
for i ← 1 to k
    do C[i] ← C[i] + C[i − 1]
for j ← n downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] − 1
```

← número de elementos iguais a j

← número de elementos menores ou iguais a j



Complexidade do Algoritmo de Ordenação por Contagem

- Complexidade simples de ser calculada
  - Dois loops de custo $\Theta(k)$ e dois loops de custo $\Theta(n)$
  - Na prática, usamos ordenação por contagem quando temos $k = O(n)$
  - Assim, temos tempo de execução $\Theta(n)$
  - Custo do algoritmo supera limite inferior $\Omega(n \lg n)$ porque não é ordenação por comparação

Ordenação por Contagem

- Algoritmo estável
  - Números com mesmo valor aparecem no arranjo de saída na mesmo ordem em que se encontram no arranjo de entrada
  - Propriedade será importante para a corretude do próximo algoritmo

Radix sort

- Como IBM tornou-se uma gigante da computação?
- Cartões perfurados para o censo americano de 1890
- Ordena uma coluna por vez
  – Começando pelos dígitos menos significativos até os mais significativos
  – Neste caso, essencial que ordenações sejam estáveis

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Radix sort

- Algoritmo pode ser aplicado, por exemplo, para ordenar datas (por dia, mês e ano)
- Algoritmo direto: procedimento supõe que cada elemento no arranjo de n elementos tem d dígitos, onde dígito 1 é dígito de mais baixa ordem e dígito d é o dígito de mais alta ordem

RADIX-SORT(A, d)

for $i \leftarrow 1$ **to** d

    do use a stable sort to sort array A on digit i

Complexidade de Radix sort

- Complexidade de cálculo simples: $\Theta(n)$, quando d constante e $k = O(n)$

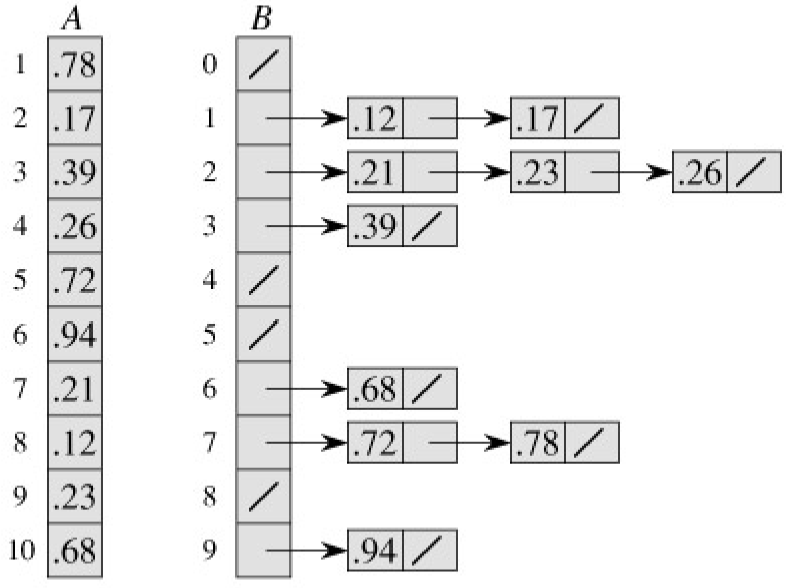
Bucket sort

- Funciona em tempo linear quando entrada é gerada a partir de distribuição uniforme sobre um intervalo
  - Idéia é dividir intervalo em n subintervalos de igual tamanho (chamados baldes)
  - Números distribuídos entre os baldes
  - Para produzir saída, ordenamos números em cada balde, percorrendo cada balde em ordem
  - Algoritmo pressupõe que entrada é arranjo de n elementos, e que cada elemento A[i] satisfaz a 0 <= A[i] < 1
  - Código exige uso de listas encadeadas

Bucket sort

BUCKET-SORT(A, n)

for $i \leftarrow 1$ **to** n
    do insert $A[i]$ into list $B[\lfloor n \cdot A[i] \rfloor]$
for $i \leftarrow 0$ **to** $n - 1$
    do sort list $B[i]$ with insertion sort
concatenate lists $B[0], B[1], \ldots, B[n-1]$ together in order
return the concatenated lists



(a) (b)

Corretude de Bucket sort

- Considere dois elementos A[i] e A[j]
  - Suponha sem perda de generalidade que A[i]<=A[j]
  - Se A[i] inserido no mesmo balde que A[j], ordenação por inserção os coloca em ordem correta
  - Se A[i] inserido em balde anterior a A[j], concatenação os coloca na ordem correta

Próxima aula...

- Estruturas de Dados Elementares