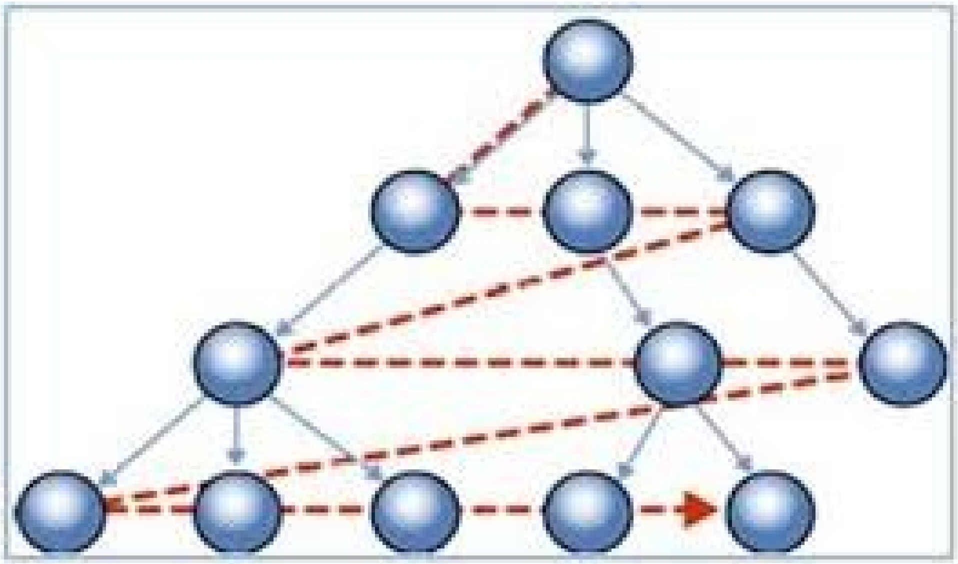
ACH2024

Aula 08 – Grafos: Busca em Largura e Aplicações: Caminhos mais curtos

Prof. Helton Hideraldo Bíscaro



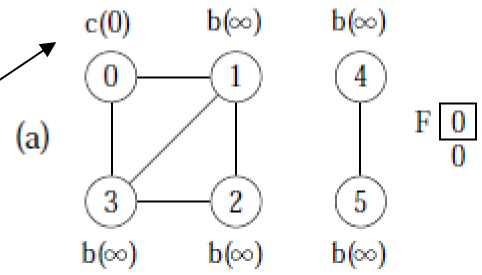
Busca em Largura

- Expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira.
- O algoritmo descobre todos os vértices a uma distância k do vértice origem antes de descobrir qualquer vértice a uma distância $k + 1$.

Busca em Largura

- Cada vértice é colorido de branco, cinza ou preto.
- Todos os vértices são inicializados branco.
- Quando um vértice é descoberto pela primeira vez ele torna-se cinza.
- Vértices cinza e preto já foram descobertos, mas são distinguidos para assegurar que a busca ocorra em largura.
- Se $(u, v) \in A$ e o vértice u é preto, então o vértice v tem que ser cinza ou preto.
- Vértices cinza podem ter alguns vértices adjacentes brancos, e eles representam a fronteira entre vértices descobertos e não descobertos.

Busca em Largura: Exemplo

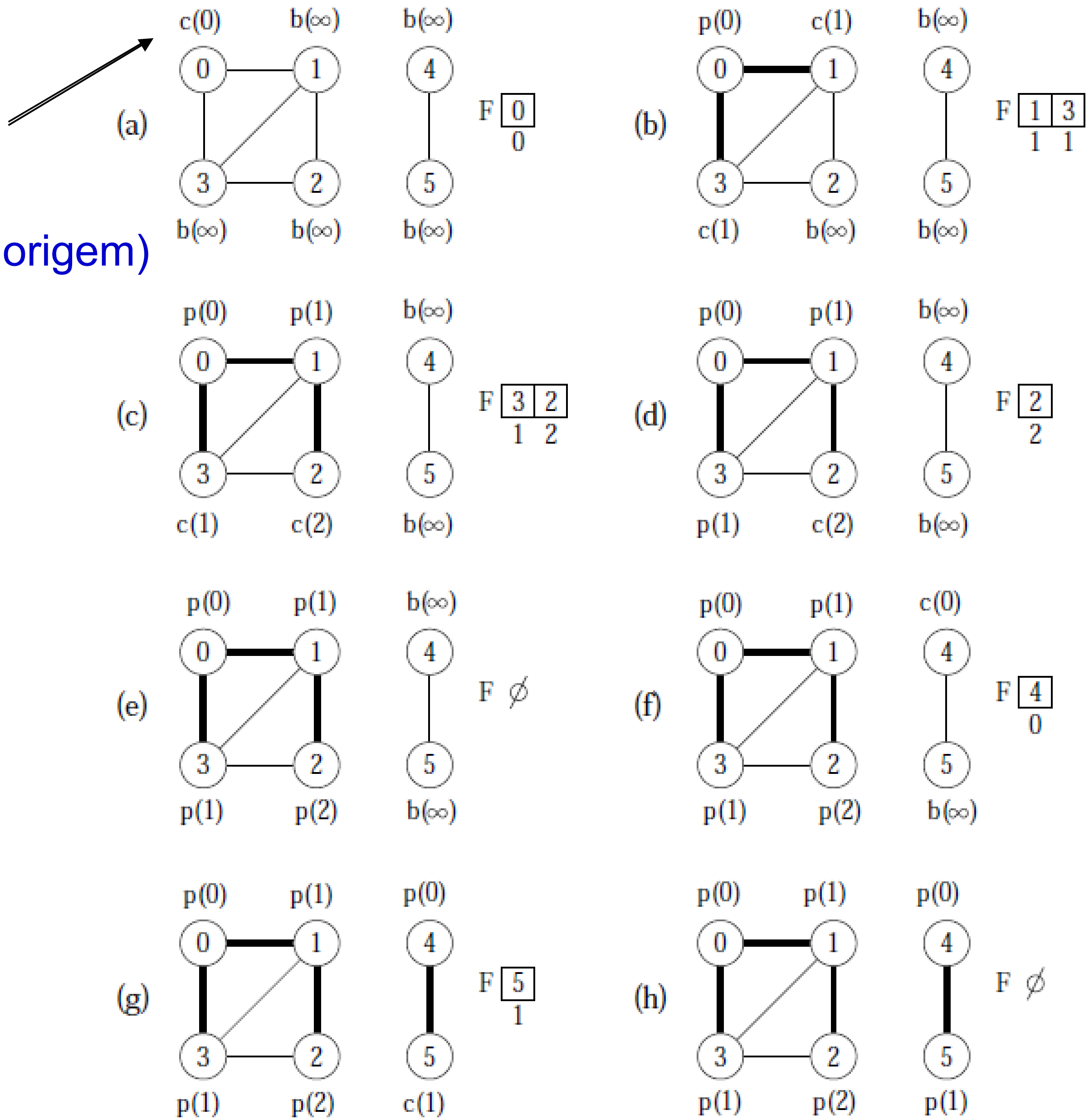


Cada vértice
tem:
cor(distância da "origem")

Fila: vértice cinza
distância desse vértice à "origem"

Busca em Largura: Exemplo

Cada vértice
tem:
cor(distância da origem)



Slide do livro do Ziviani – cap 7

Implementação

```
buscaEmLargura(grafo){
    Aloca vetores cor, antecessor, distancia com tamanho grafo->nrVertices
    Para cada vertice v
        cor[v] ← branco;  antecessor[v] ← -1; distancia[v] ← ∞;
    Para cada vertice v
        se cor[v] = branco
            visitaLargura(v, grafo, cor, antecessor, distancia);
}

visitaLargura(s, grafo, cor, antecessor, distancia){
    cor[s] ← cinza;
    distancia[s] ← 0;
    F ← Ø;
    insereFila(F, s);
    enquanto F ≠ Ø
        w ← removeFila(F)
        para cada vertice u da lista de adjacência de w
            se cor[u] = branco
                cor[u] ← cinza;
                antecessor[u] ← w;
                distancia[u] ← distancia[w] + 1;
                insereFila(F, u);
        cor[w] ← preto;
}
```

“VisitaBfs” no livro do Ziviani (slides seguintes)

Busca em Largura: Análise

- O custo de inicialização do primeiro anel em *BuscaEmLargura* é $O(|V|)$ cada um.
- O custo do segundo anel é também $O(|V|)$.
- *VisitaBfs*: enfileirar e desenfileirar têm custo $O(1)$, logo, o custo total com a fila é $O(|V|)$.
- Cada lista de adjacentes é percorrida no máximo uma vez, quando o vértice é desenfileirado.
- Desde que a soma de todas as listas de adjacentes é $O(|A|)$, o tempo total gasto com as listas de adjacentes é $O(|A|)$.
- Complexidade total: é $O(|V| + |A|)$.

Aplicações

- Para encontrar os vértices vizinhos dentro de um certo raio (por exemplo, em sistemas de computação móvel, navegação GPS, etc)
- Pessoas a uma certa distância em uma rede social
- Coleta de lixo em memória (melhor localidade de referência do que se usar busca em profundidade)
- Caminhos mais curtos

Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

Implementação: (u tendo sido a origem da busca em largura)

```
void imprimeCaminho(int u, int v, int antecessor[])
{
```

DICA: usando recursão!

Caminhos Mais Curtos

- A busca em largura obtém o **caminho mais curto** de u até v.
- O procedimento *VisitaBfs* contrói uma árvore de busca em largura que é armazenada na variável *Antecessor*.

Implementação: (u tendo sido a origem da busca em largura)

```
void imprimeCaminho(int u, int v, int antecessor[])
{
    if (d[v] == ∞) {
        printf ("Nao existe caminho de %d ate %d" , u, v) ;
        return;
    }
    if (u == v) { printf ( "%d " , u ; return; }
    else {
        imprimeCaminho(u, antecessor[v], antecessor);
        printf ( "%d " , v);
    }
}
```

Caminhos de Peso Mínimo

Caminhos de peso mínimo

Um caminho **mais curto** é aquele com **menor número de arestas**

Muitas vezes não estamos interessados no número de arestas, e sim no custo do caminho (soma dos pesos das arestas do caminho), ou seja, no caminho de peso mínimo

A aplicação direta da busca em largura, como feita para caminhos mais curtos, não é mais suficiente

Infelizmente, esse problema é também chamado “caminho mais curto”, o que causa uma certa confusão

Daqui para frente, usaremos o termo “caminho mais curto” como sinônimo de “caminho de peso mínimo”

Assume-se um grafo **direcionado** e **ponderado**

Caminhos Mais Curtos: Aplicação

- Um motorista procura o caminho mais curto entre Diamantina e Ouro Preto. Possui mapa com as distâncias entre cada par de interseções adjacentes.
- Modelagem:
  - $G = (V, A)$: grafo direcionado ponderado, mapa rodoviário.
  - V: interseções.
  - A: segmentos de estrada entre interseções
  - $p(u, v)$: peso de cada aresta, distância entre interseções.
- Peso de um caminho: $p(c) = \sum_{i=1}^{k} p(v_{i-1}, v_i)$
- Peso do caminho de peso mínimo (do caminho "mais curto"):

$$\delta(u, v) = \begin{cases} min\left\{p(c) : u \overset{c}{\rightsquigarrow} v\right\} & \text{se existir caminho de } u \text{ a } v \\ \infty & \text{caso contrário} \end{cases}$$

- **Caminho mais curto** do vértice u ao vértice v: qualquer caminho c com peso $p(c) = \delta(u, v)$.

Caminhos Mais Curtos

- **Caminhos mais curtos a partir de uma origem**: dado um grafo ponderado $G = (V, A)$, desejamos obter o caminho mais curto a partir de um dado vértice origem $s \in V$ até cada $v \in V$.
- Muitos problemas podem ser resolvidos pelo algoritmo para o problema origem única:
  - **Caminhos mais curtos com destino único**: reduzido ao problema origem única invertendo a direção de cada aresta do grafo.
  - **Caminhos mais curtos entre um par de vértices**: o algoritmo para origem única é a melhor opção conhecida.
  - **Caminhos mais curtos entre todos os pares de vértices**: resolvido aplicando o algoritmo origem única $|V|$ vezes, uma vez para cada vértice origem.

Caminhos mais curtos – quando eles não existem

Considerando a modelagem matemática proposta, há casos em que não existe um caminho mais curto entre dois vértices

Para um vértice origem s:

Se um vértice v não é alcançável por s:

$\delta(s, v)$ = infinito

Se houver algum ciclo alcancável por s com peso total negativo:

- Para cada vértice v deste ciclo ou para vértice v para o qual existe um caminho de s a v passando pelo ciclo,
  - Não existe um caminho mais curto de s a v,
  - Ou seja, $\delta(s, v)$ = -infinito

Caminhos mais curtos – quando eles não existem

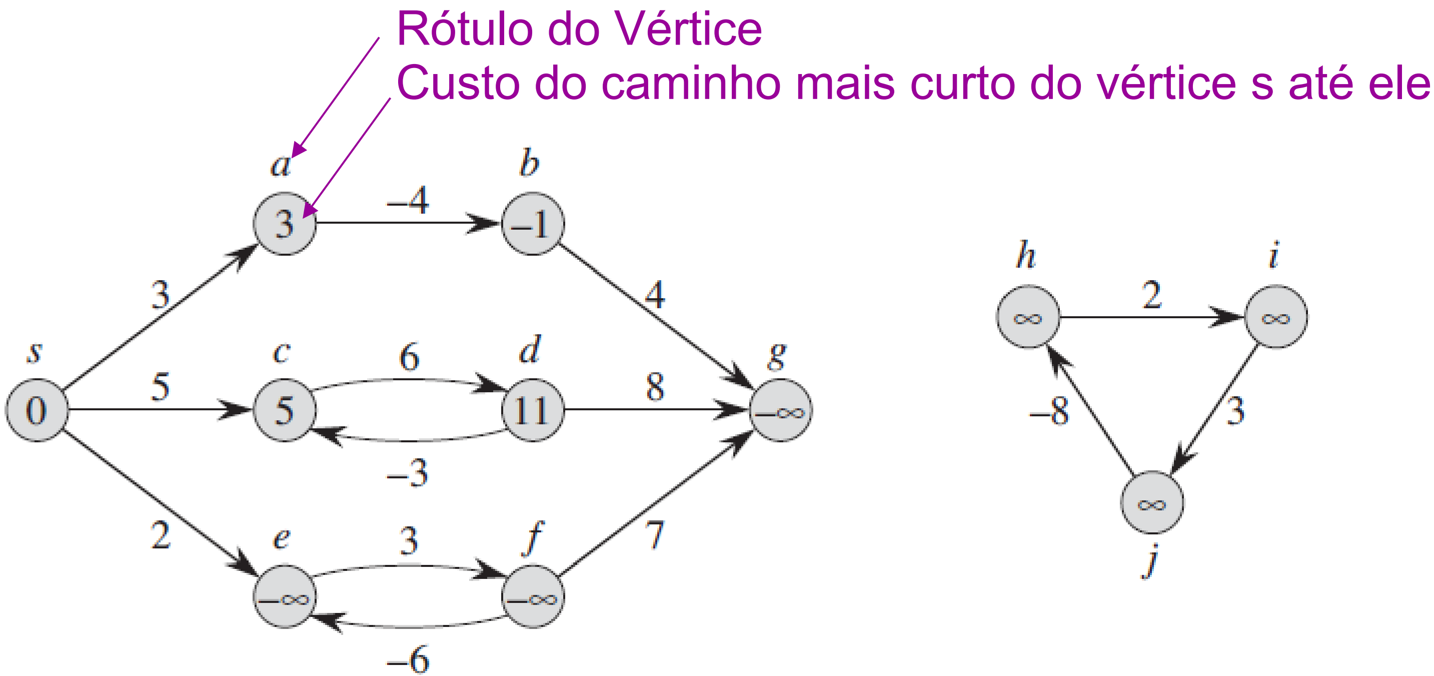


Figura: Livro do Cormem cap 24

Caminhos Mais Curtos

- A representação de caminhos mais curtos pode ser realizada pela variável *Antecessor*.
- Para cada vértice $v \in V$ o $Antecessor[v]$ é um outro vértice $u \in V$ ou *nil* (-1).
- O algoritmo atribui a *Antecessor* os rótulos de vértices de uma cadeia de antecessores com origem em v e que anda para trás ao longo de um caminho mais curto até o vértice origem s.
- Dado um vértice v no qual $Antecessor[v] \neq nil$, o procedimento *ImprimeCaminho* pode imprimir o caminho mais curto de s até v.
- Os valores em $Antecessor[v]$, em um passo intermediário, não indicam necessariamente caminhos mais curtos.
- Entretanto, ao final do processamento, *Antecessor* contém uma árvore de caminhos mais curtos definidos em termos dos pesos de cada aresta de G, ao invés do número de arestas. (vetor “distancia” armazenará soma dos pesos)
- Caminhos mais curtos não são necessariamente únicos.

π[v] no livro do Cormen

Caminhos Mais Curtos

- A representação de caminhos mais curtos pode ser realizada pela variável *Antecessor*.
- Para cada vértice $v \in V$ o $Antecessor[v]$ é um outro vértice $u \in V$ ou *nil* (-1).
- O algoritmo atribui a *Antecessor* os rótulos de vértices de uma cadeia de antecessores com origem em v e que anda para trás ao longo de um caminho mais curto até o vértice origem s.
- Dado um vértice v no qual $Antecessor[v] \neq nil$, o procedimento *ImprimeCaminho* pode imprimir o caminho mais curto de s até v.
- Os valores em $Antecessor[v]$, em um passo intermediário, não indicam necessariamente caminhos mais curtos.
- Entretanto, ao final do processamento, *Antecessor* contém uma árvore de caminhos mais curtos definidos em termos dos pesos de cada aresta de G, ao invés do número de arestas. (vetor "distancia" armazenará soma dos pesos)
- Caminhos mais curtos não são necessariamente únicos.

d[v] no livro do Cormen

Árvore de caminhos mais curtos

- Uma árvore de caminhos mais curtos com raiz em $s \in V$ é um subgrafo direcionado $G' = (V', A')$, onde $V' \subseteq V$ e $A' \subseteq A$, tal que:
  1. V' é o conjunto de vértices alcançáveis a partir de $s \in G$,
  2. G' forma uma árvore de raiz s,
  3. para todos os vértices $v \in V'$, o caminho simples de s até v é um caminho mais curto de s até v em G.

Árvore de caminhos mais curtos

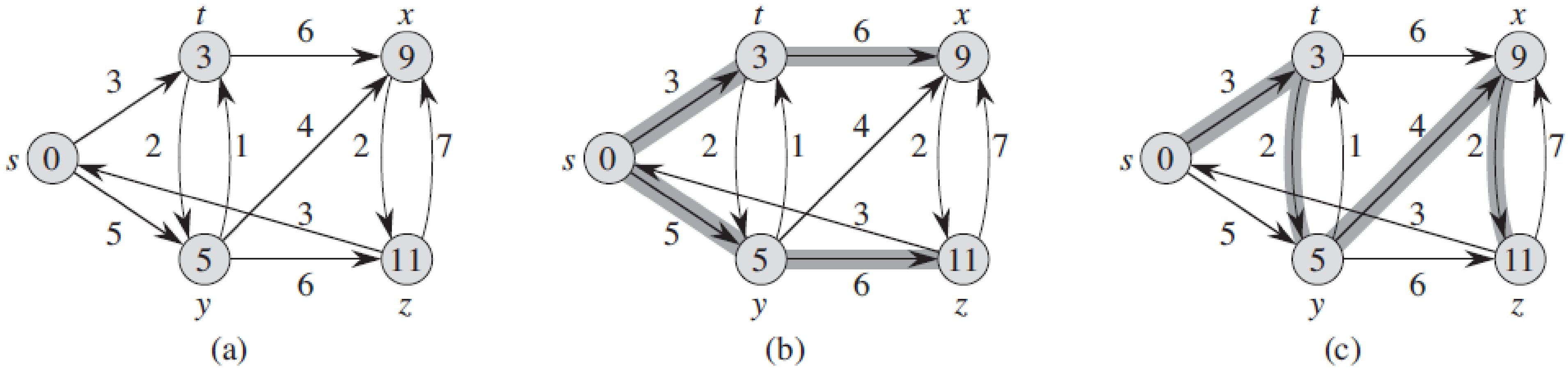


Figure 24.2 (**a**) A weighted, directed graph with shortest-path weights from source s. (**b**) The shaded edges form a shortest-paths tree rooted at the source s. (**c**) Another shortest-paths tree with the same root.

Relaxamento

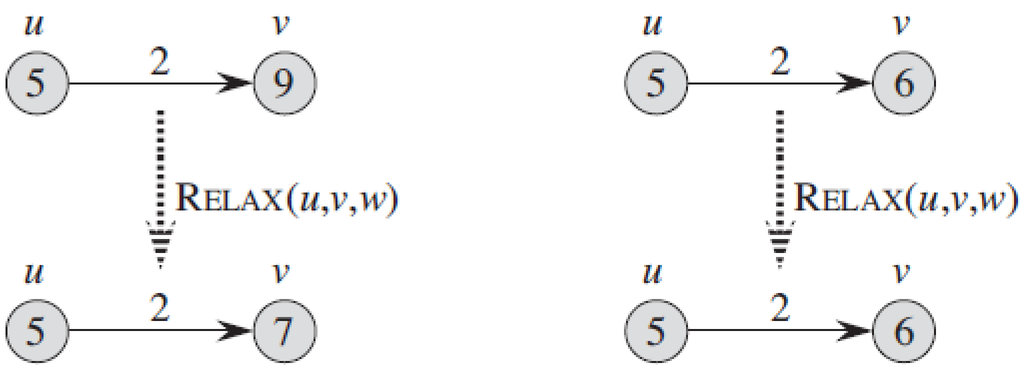
Técnica usada por algoritmos de caminhos mais curtos

Cada vértice v terá um valor d[v], que é uma estimativa de pior caso (limite superior) do custo mínimo do caminho de s (origem) a v

```
INITIALIZE-SINGLE-SOURCE(G, s)
1  for each vertex v ∈ G.V
2        d[v] = ∞
3        π[v] = NIL
4     d[s] = 0
```

Relaxamento

Relaxar uma aresta (u,v): verificar se d[v] pode ser decrementado ao se considerar um caminho de s a v passando por u (ou seja, verificar se usar essa aresta melhora a estimativa atual):



(a) (b)

w representa a informação dos pesos

RELAX(u, v, w)

if d[v] $>$ d[u] $+ w(u, v)$
    d[v] $=$ d[u] $+ w(u, v)$
    π[v] $= u$

Algoritmo de Dijkstra

Considerando que todas as arestas são não-negativas

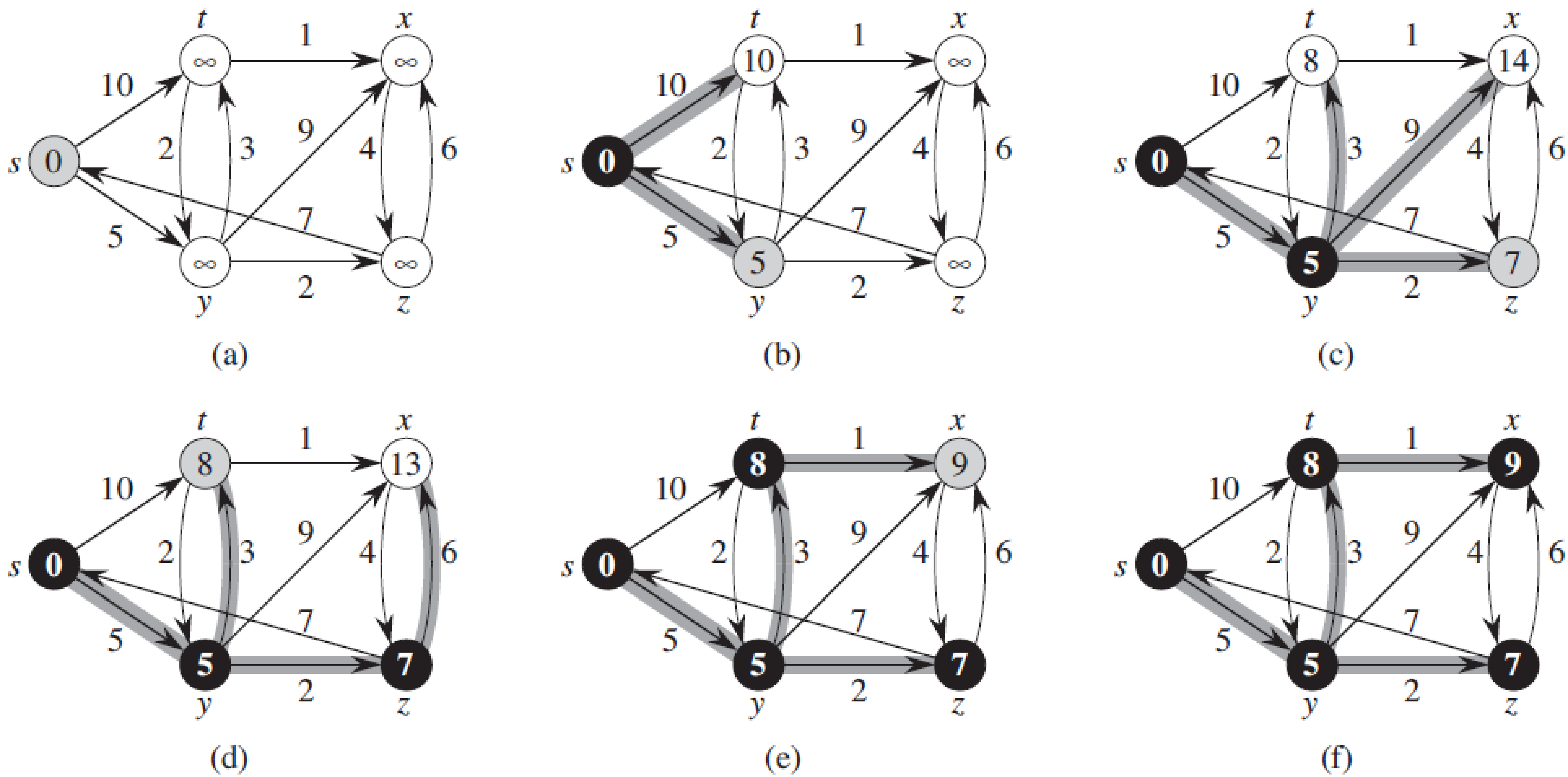
```
DIJKSTRA(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

S: usado para prova de corretude no livro do Cormen (conjunto de vértices já processados)

Q é uma fila de prioridades baseada no valor d (quanto menor o d maior a prioridade)

Algoritmo de Dijkstra

Exemplo:

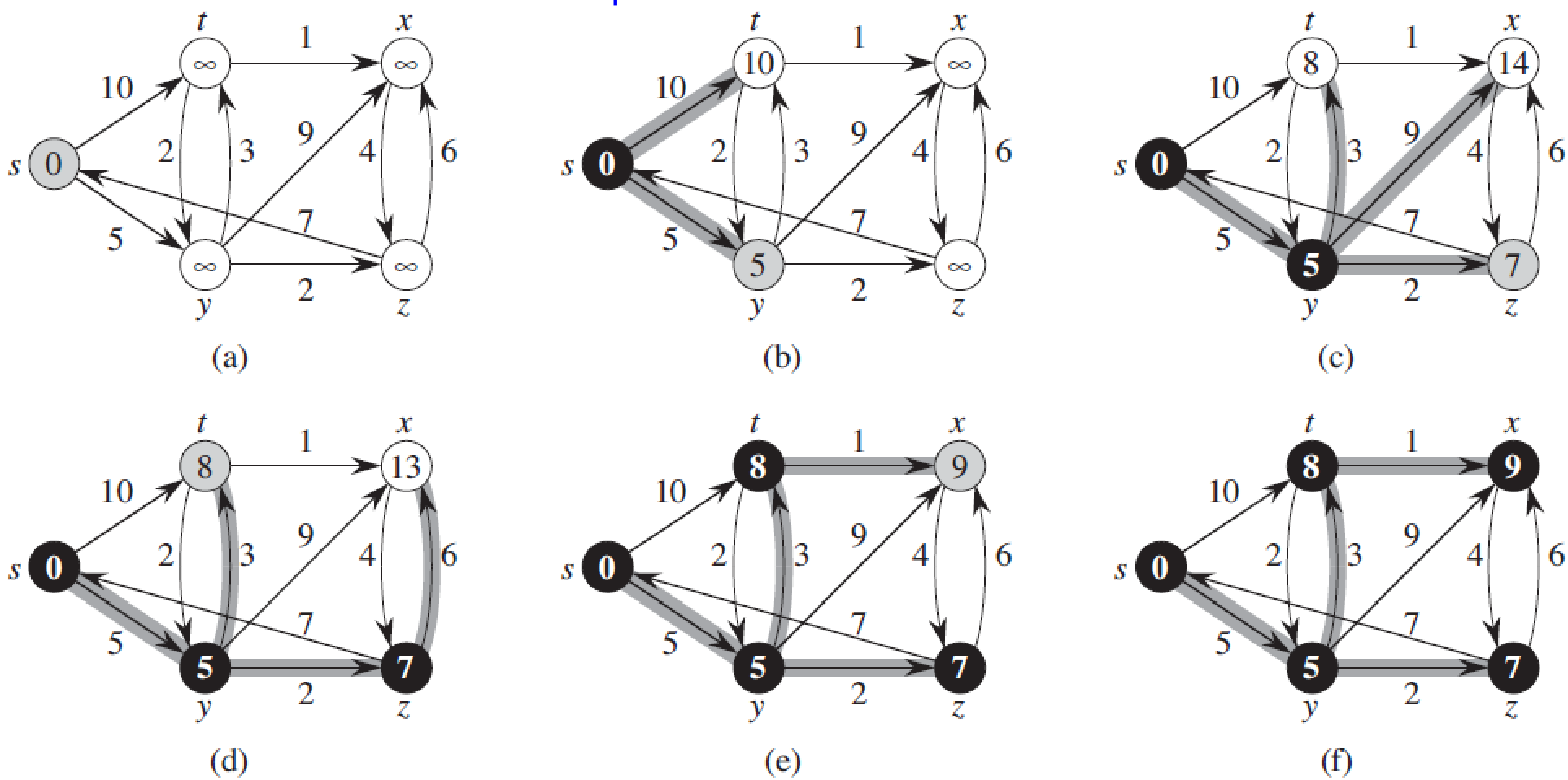


Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele ralaxadas

Algoritmo de Dijkstra

Exemplo:

Note que, como as arestas possuem peso >= 0, vértices que ainda estão em Q não terão d menor do que os dos vértices que já foram processados, garantindo que só uma passada sobre todos os vértices é o suficiente para o cálculo correto das distâncias.



Vértices brancos estão em Q, cinza é o que acaba de ser removido, pretos já tiveram todas as arestas que saem dele ralaxadas

Algoritmo de Dijkstra - complexidade

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Algoritmo de Dijkstra - complexidade

```
DIJKSTRA(G, w, s)
    INITIALIZE-SINGLE-SOURCE(G, s)
    S = ∅
    Q = G.V
    while Q ≠ ∅
        u = EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            RELAX(u, v, w)
```

Depende de como Q é implementada!!!

Algoritmo de Dijkstra - complexidade

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Seja uma estrutura de dados para Q (representaremos em vermelho a complexidade dependente dela)

L. 1: O(V)

L. 3: x

Loop L.4 executado |V| vezes

L. 5: y

L. 7-8: no total |A| chamadas a RELAX, cada uma demanda alterar d[v] em Q (que tem complexidade z)
total de RELAX: O(A*z)

Total: O (V+x+(V*y) + (A+V)* z)
ou
O (V+x+(V*y) + A* z) se todos os vértices forem alcançáveis a partir da origem

Assumindo-se uma implementação de grafos por listas de adjacência

Heaps

Implementações descritas no livro do Cormen, em particular:

Heap Fibonacci (cap 20)

Heap Binário (cap 6) – pincelado aqui. Ver detalhes no livro do Cormen!!!

- → normalmente quanto maior a chave maior a prioridade (para usá-lo no algoritmo de Dijkstra faremos o contrário => vocês precisarão adaptar as rotinas de manipulação do heap)

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 - Tipo Abstrato de Dados

- Operações:
  1. Constrói uma fila de prioridades a partir de um conjunto com n itens.
  2. Informa qual é o maior item do conjunto.
  3. Retira o item com maior chave.
  4. Insere um novo item.
  5. Aumenta o valor da chave do item i para um novo valor que é maior que o valor atual da chave.
  6. Substitui o maior item por um novo item, a não ser que o novo item seja maior.
  7. Altera a prioridade de um item.
  8. Remove um item qualquer.
  9. Ajunta duas filas de prioridades em uma única.

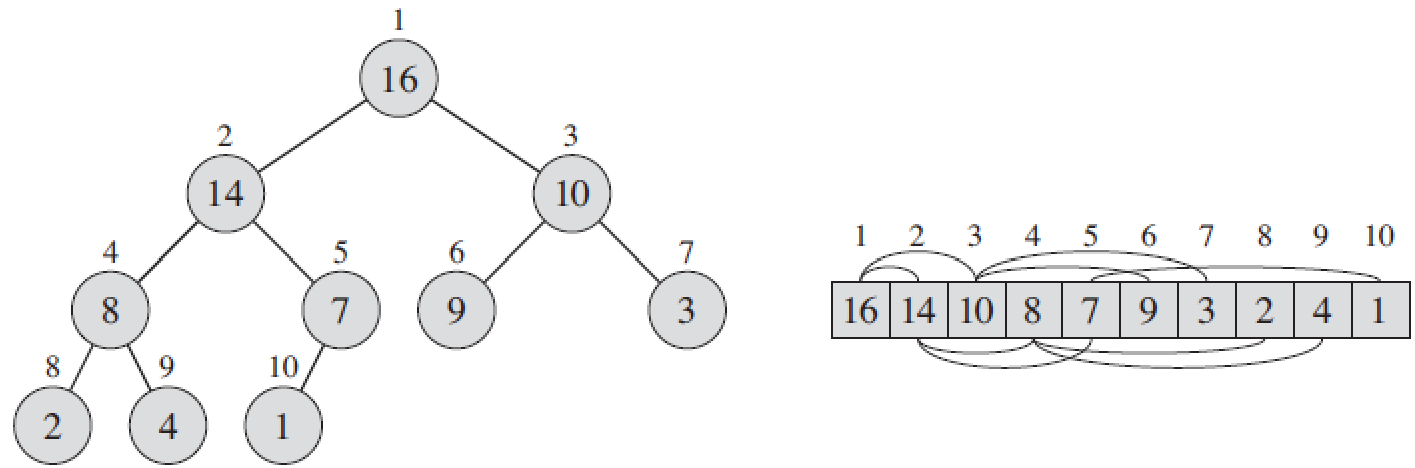
Heap

- As chaves na árvore satisfazem a condição do *heap*.
- A chave em cada nó é maior 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 array:

1	2	3	4	5	6	7
S	R	O	E	N	A	D

- 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 $i \text{ div } 2$.

Exemplos e códigos a seguir, do livro do Cormen, assumem um heap no qual quanto maior a chave maior a prioridade (MAX-HEAP). No caso de seu uso no algoritmo de Dijkstra para caminhos mais curtos deveria ser um MIN-HEAP.



(a) (b)

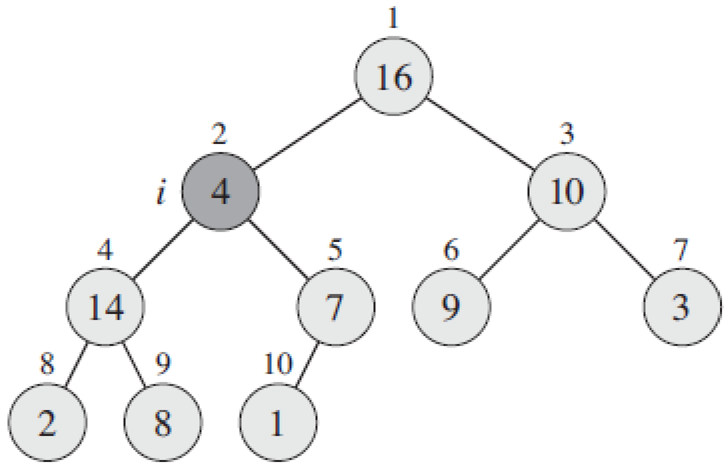
PARENT(i)
1 **return** $\lfloor i/2 \rfloor$

LEFT(i)
1 **return** $2i$

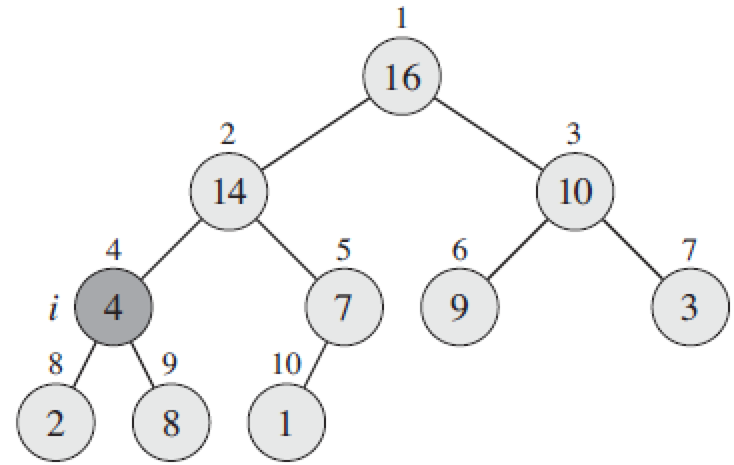
RIGHT(i)
1 **return** $2i + 1$

Notem que precisa começar no 1 !

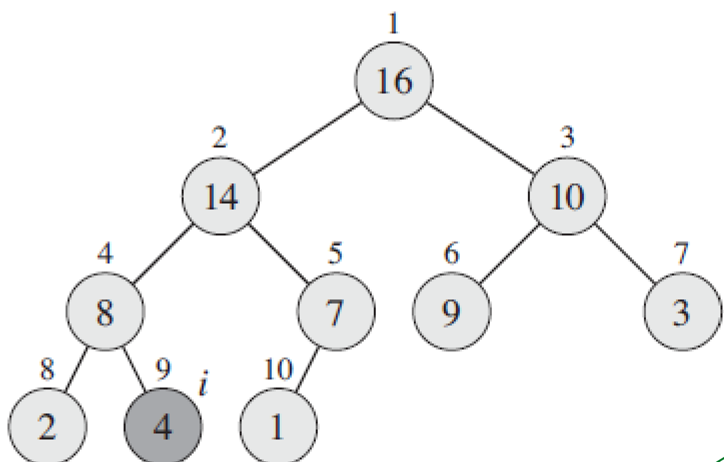
MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. Coloca o valor da atual posição i em um lugar adequado



(a)



(b)

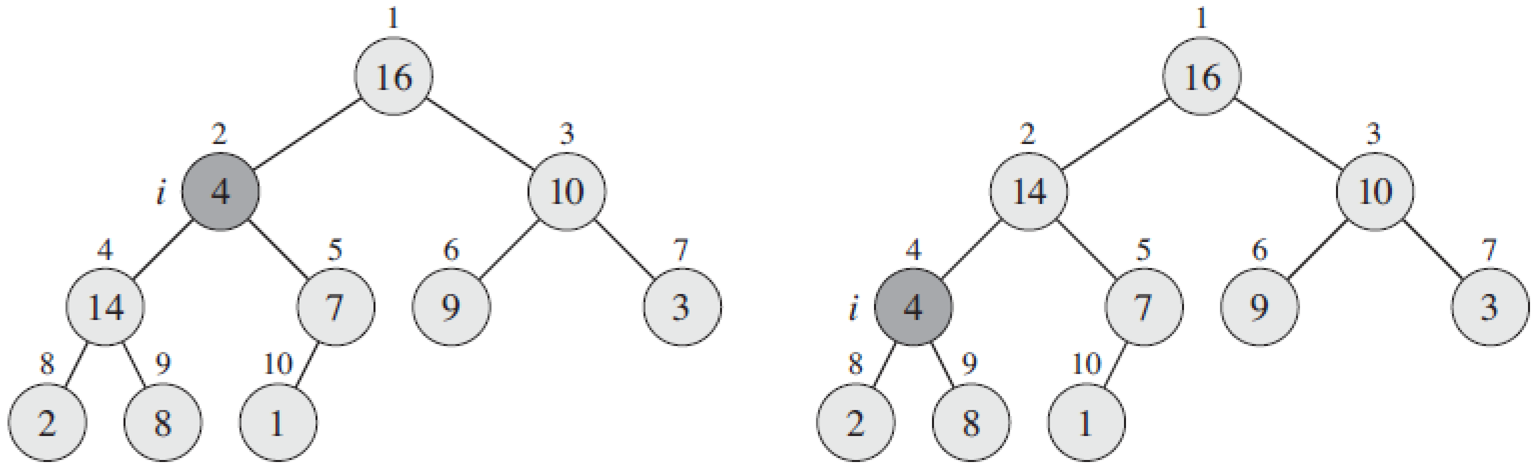


(c)

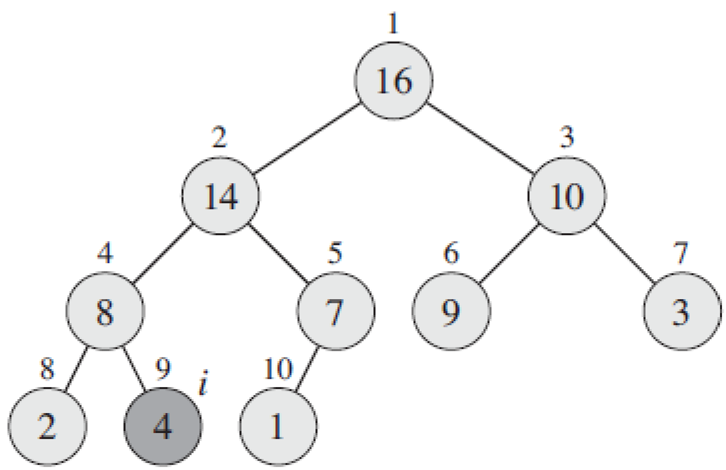
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Verifica quem é menor: i ou um de seus filhos (largest)

MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. Coloca o valor da atual posição i em um lugar adequado



(a) (b)

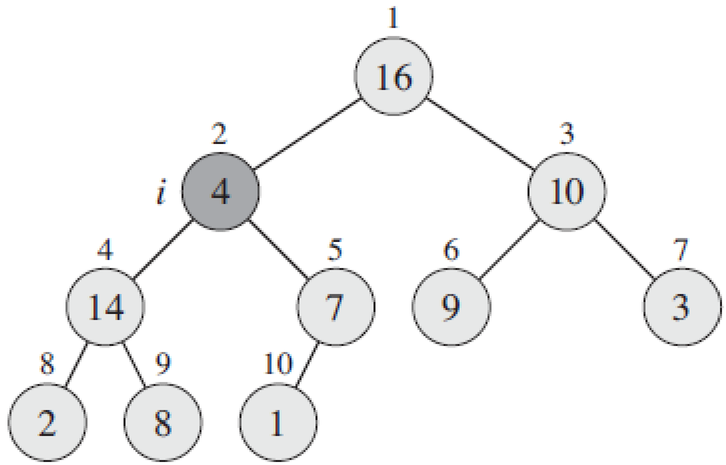


(c)

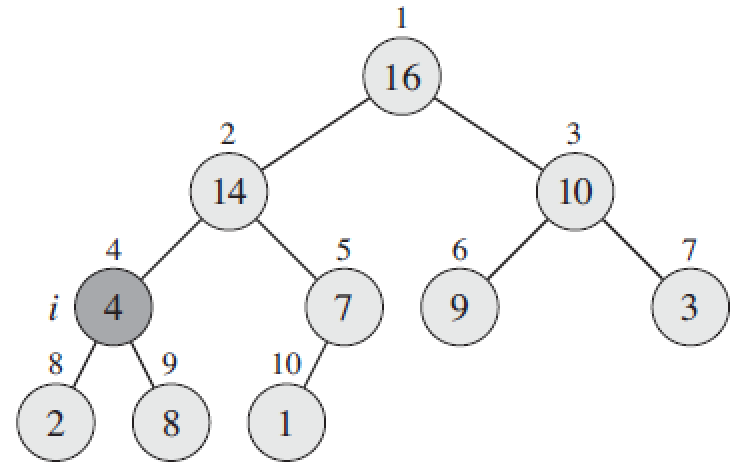
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Complexidade: ?

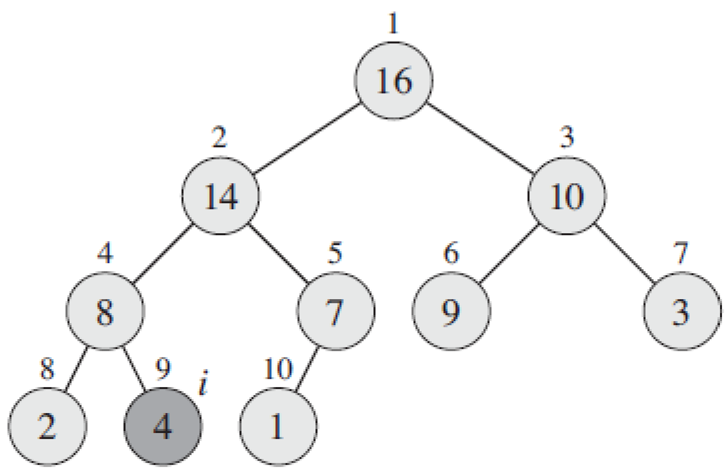
MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. Coloca o valor da atual posição i em um lugar adequado



(a)



(b)



(c)

MAX-HEAPIFY(A, i)

```
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Complexidade: O(lg n)

Construção do heap

BUILD-MAX-HEAP(A)

$A.heap\text{-}size = A.length$
for $i = \lfloor A.length/2 \rfloor$ **downto** 1
    MAX-HEAPIFY(A, i)

Nós do último nível satisfazem a condição do heap

Precisa então acertar o posicionamento dos nós do nível superior para cima

Construção do heap

```
BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)
```

Complexidade: ?

Construção do heap

```
BUILD-MAX-HEAP(A)
    A.heap-size = A.length
    for i = ⌊A.length/2⌋ downto 1
        MAX-HEAPIFY(A, i)
```

Complexidade: O(n)

(ver seção 6.3 do livro do Cormen - 3.ed)

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```

Complexidade: ?

HEAP-EXTRACT-MAX(A)

```
if A.heap-size < 1
    error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

Complexidade: O(lg n)

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

```
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

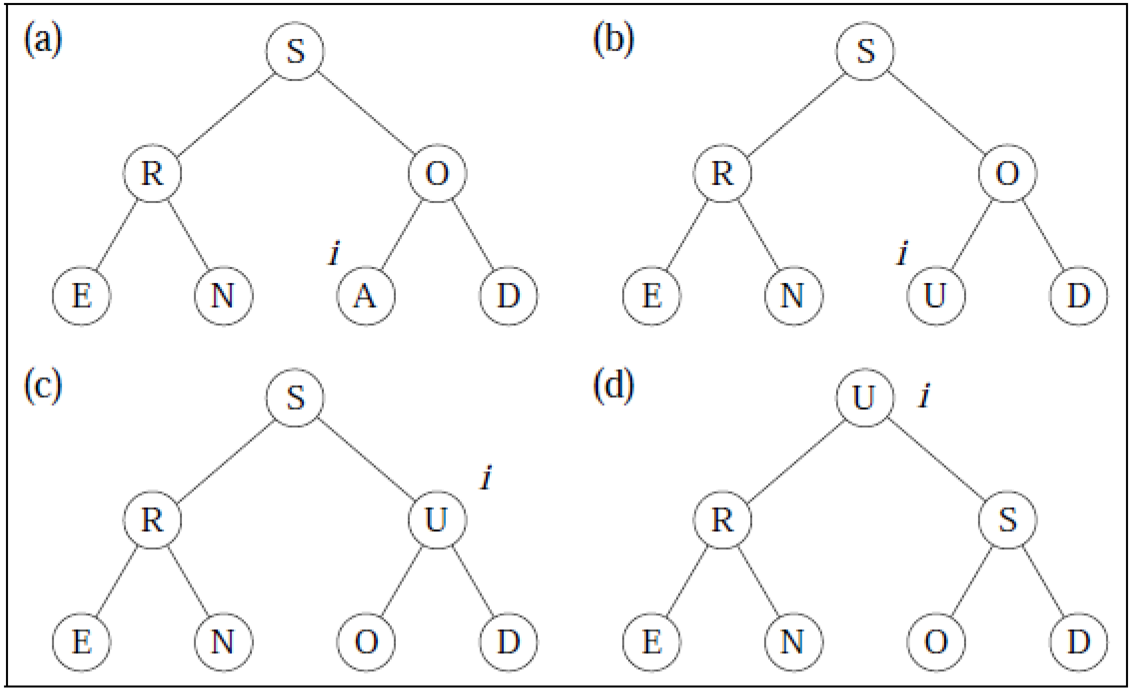
Complexidade: ?

```
HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
    error "new key is smaller than current key"
A[i] = key
while i > 1 and A[PARENT(i)] < A[i]
    exchange A[i] with A[PARENT(i)]
    i = PARENT(i)
```

Complexidade: O(lg n)

Heap

- Exemplo da operação de aumentar o valor da chave do item na posição i:



- O tempo de execução do procedimento AumentaChave em um item do *heap* é $O(\log n)$.

Algoritmo de Dijkstra - complexidade

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Considerando Q como um heap binário:

L. 1: O(V)

L. 3: O(V)

Loop L.4 executado |V| vezes

L. 5: no total O(V lg V)

L. 7-8: no total A chamadas a RELAX, cada uma com um DECREASE-KEY implícito: O(A lgV)

Total: O((A+V) lg V) ou

O(A lgV) se todos os vértices forem alcançáveis a partir da origem

Algoritmo de Dijkstra - complexidade

```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

Considerando Q como um heap binário:

L. 1: O(V)

L. 3: O(V)

Loop L.4 executado |V| vezes

L. 5: no total O(V lg V)

L. 7-8: no total A chamadas a RELAX, cada uma com um DECREASE-KEY implícito: O(A lgV)

Total: O((A+V) lg V) ou

O(A lgV) se todos os vértices forem alcançáveis a partir da origem

Usando heaps Fibonacci: O(V lg V + A)

Algoritmo Bellman-Ford

Resolve o caso geral (arestas podem ter pesos negativos)

Retorna falso se o grafo tiver um ciclo negativo

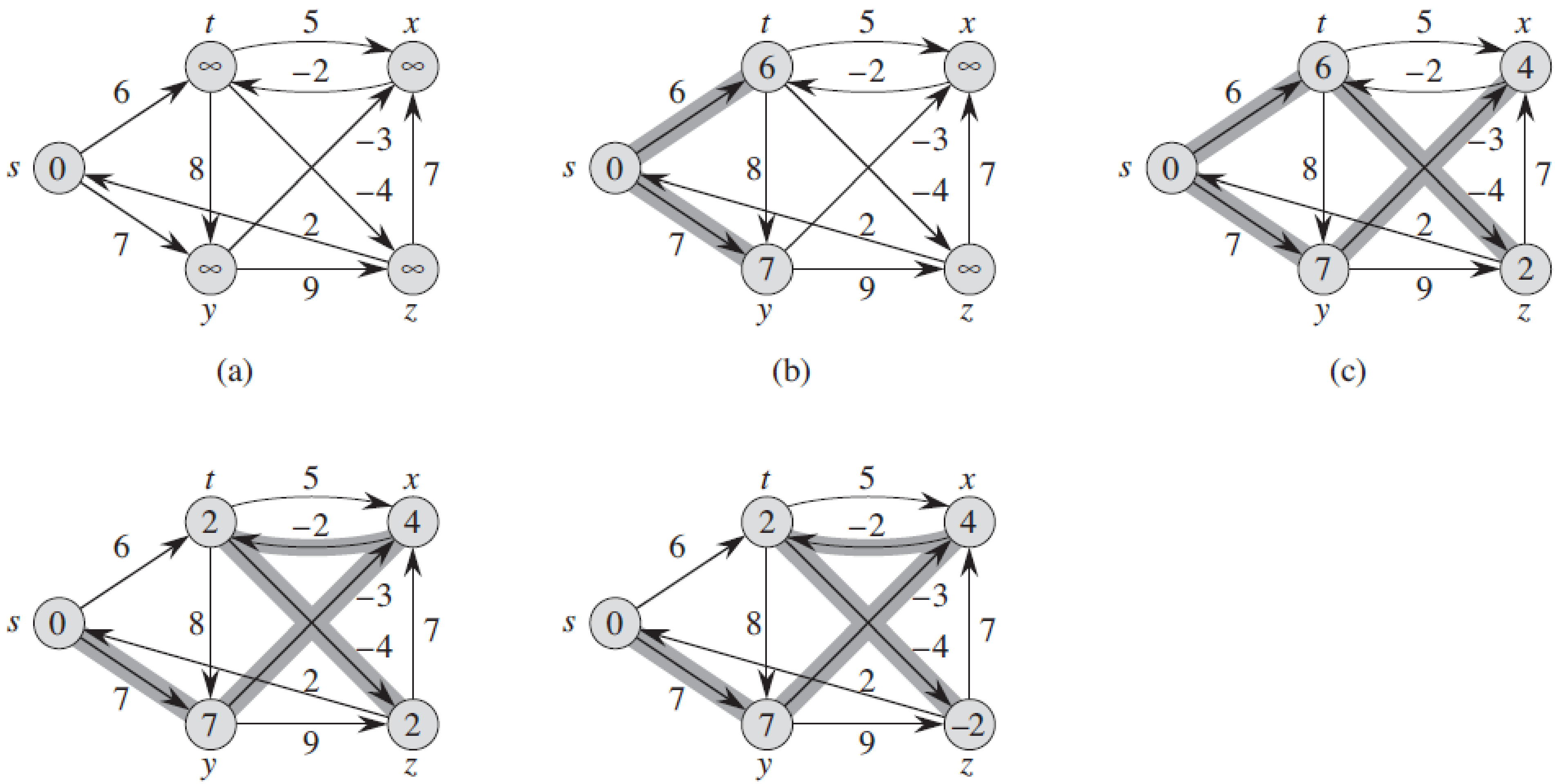
```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if d[v] > d[u] + w(u, v)
        return FALSE
return TRUE
```

Como o caminho de peso mínimo não tem ciclo, tem comprimento no máximo |V|-1

Logo, em |V|-1 rodadas, todas as arestas deste caminho são corretamente relaxadas para seus valores reais

Algoritmo Bellman-Ford

Exemplo:(algoritmo retorna TRUE)



(a) (b) (c) (d) (e)

Complexidade do Algoritmo Bellman-Ford

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| - 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if d[v] > d[u] + w(u, v)
7          return FALSE
8  return TRUE
```

L. 1: O(V)

L. 2-4: O(VA)

L. 5-7: O(A)

Total: O(VA)

Referências

Ziviani: seções 7.5 e 7.0

Cormen: seção 22.2 e cap 24