ACH2024

Aula 10 – Grafos:
Árvore Geradora Mínima
Algoritmo de Prim (cont.) e
Algoritmo de Kruskal

Profa. Ariane Machado Lima

Aula anterior

Árvore geradora mínima *(Minimum Spanning Tree)*

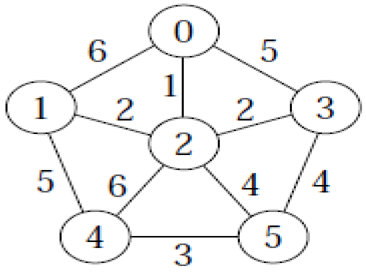
Veremos dois algoritmos distintos para resolver esse problema:

- Algoritmo de Prim
- Algoritmo de Kruskal

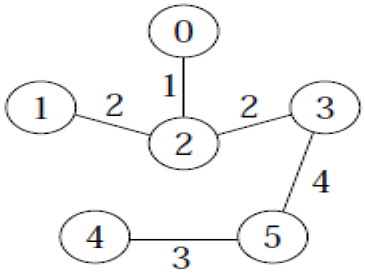
Árvore Geradora Mínima

- Como $G' = (V, T)$ é acíclico e conecta todos os vértices, T forma uma árvore chamada **árvore geradora** de G.
- O problema de obter a árvore T é conhecido como **árvore geradora mínima** (AGM).

Ex.: Árvore geradora mínima T cujo peso total é 12. T não é única, pode-se substituir a aresta $(3, 5)$ pela aresta $(2, 5)$ obtendo outra árvore geradora de custo 12.



(a)



(b)

AGM - Algoritmo Genérico

```
void GenericoAGM()
{  S = ∅;
   while(S não constitui uma árvore geradora mínima)
   { (u,v) = seleciona(A);
     if(aresta (u,v) é segura para S) S = S+ {(u,v)} }
   return S;
}
```

No final será o conjunto de arestas que formam a AGM

A principal diferença entre os algoritmos de Prim e de Kruskal é como eles definem mais especificamente o que é uma **aresta segura**.

- Uma estratégia **gulosa** permite obter a AGM adicionando uma aresta de cada vez.
- Invariante: Antes de cada iteração, S é um subconjunto de uma árvore geradora mínima.
- A cada passo adicionamos a S uma aresta (u, v) que não viola o invariante. (u, v) é chamada de uma **aresta segura**.
- Dentro do **while**, S tem que ser um subconjunto próprio da AGM T, e assim tem que existir uma aresta $(u, v) \in T$ tal que $(u, v) \notin S$ e (u, v) é seguro para S.

isto é, assim que se entra no while,

Algoritmo de Prim

Uma árvore, inicialmente vazia, cresce até chegar a ser uma AGM

A cada passo um vértice é acrescentado a essa árvore

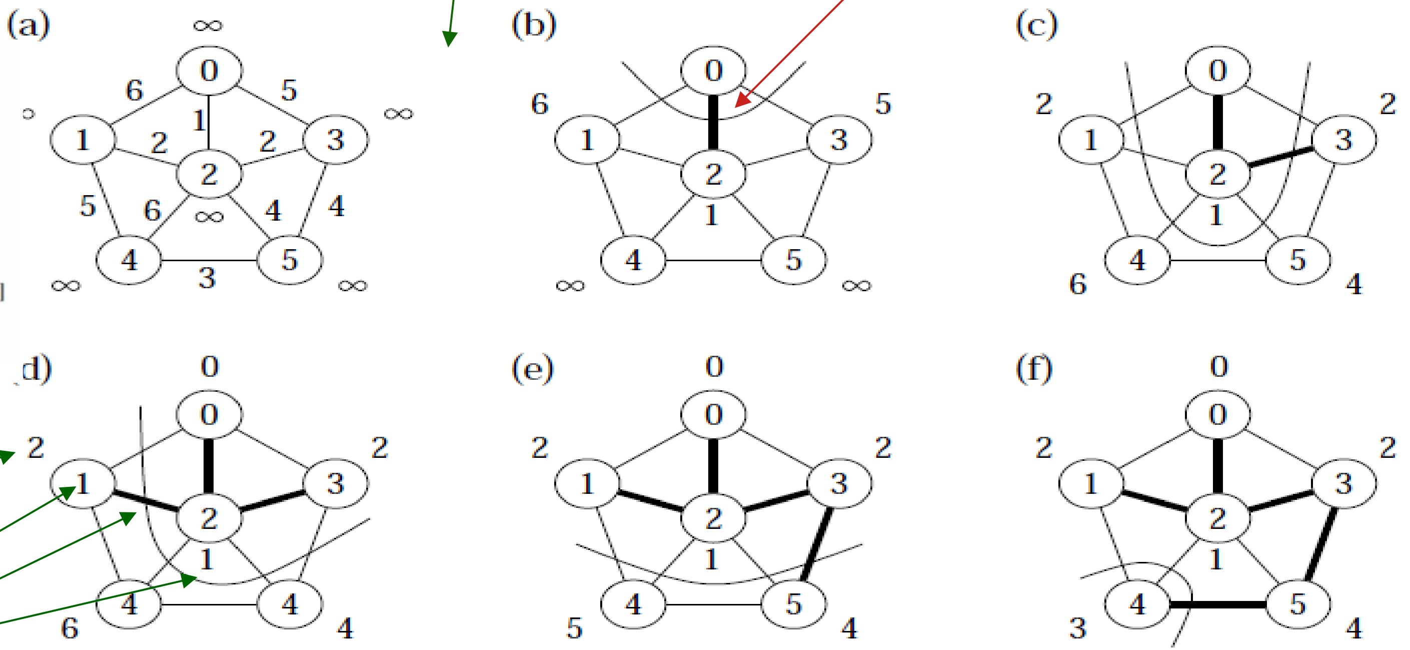
```
void GenericoAGM()
{  S = ∅;   ⟶  No final será o conjunto de arestas que formam a AGM
   while(S não constitui uma árvore geradora mínima)
   { (u,v) = seleciona(A);
     if(aresta (u,v) é segura para S) S = S+ {(u,v)} }
   return S;
}
```

Corte separa vértices que são extremos das arestas de S dos demais vértices do grafo (inicialmente só o vértice 0), **ou seja, vértices que estão fora de Q dos que estão dentro de Q**

Aresta leve a ser adicionada a S

Algoritmo de Prim: Exemplo

```
MST-PRIM(G, w, r)
  for each u ∈ V[G]
      do key[u] ← ∞
         π[u] ← NIL
  key[r] ← 0
  Q ← V[G]
  while Q ≠ ∅
      do u ← EXTRACT-MIN(Q)
         for each v ∈ Adj[u]
             do if v ∈ Q and w(u, v) < key[v]
                   then π[v] ← u
                        key[v] ← w(u, v)
```



key
rótulo do vértice
Π (antecessor)
corte

Complexidade

Depende da implementação de Q….

```
MST-PRIM(G, w, r)
1   for each u ∈ V[G]
2       do key[u] ← ∞
3          π[u] ← NIL
4   key[r] ← 0
5   Q ← V[G]
6   while Q ≠ Ø
7       do u ← EXTRACT-MIN(Q)
8          for each v ∈ Adj[u]
9              do if v ∈ Q and w(u, v) < key[v]
10                   then π[v] ← u
11                        key[v] ← w(u, v)
```

Se Q for uma lista linear simples não ordenada:

Linhas 1 a 5: O(V)

Loop da linha 6: V vezes

Linha 7: O(V)

Linha 6-7: $O(V^2)$

Linhas 8-11: O(A) no total (assumindo lista de adjacência)

Complexidade: $O(V) + O(V^2) + O(A) = O(V^2)$

Arg! Precisa melhorar….

Aula de hoje

- Usando Heaps no algoritmo de Prim
- Algoritmo de Kruskal

Filas de prioridades

Em muitas aplicações de grafos, a eficiência total do algoritmo depende da eficiência de outras estruturas de dados auxiliares

Uma delas é para o armazenamento e manipulação de filas de prioridades

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.

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.

Que estrutura de dados simples vêm à nossa cabeça?

Qual seria a complexidade dessas operações?

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:
- 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 div 2.

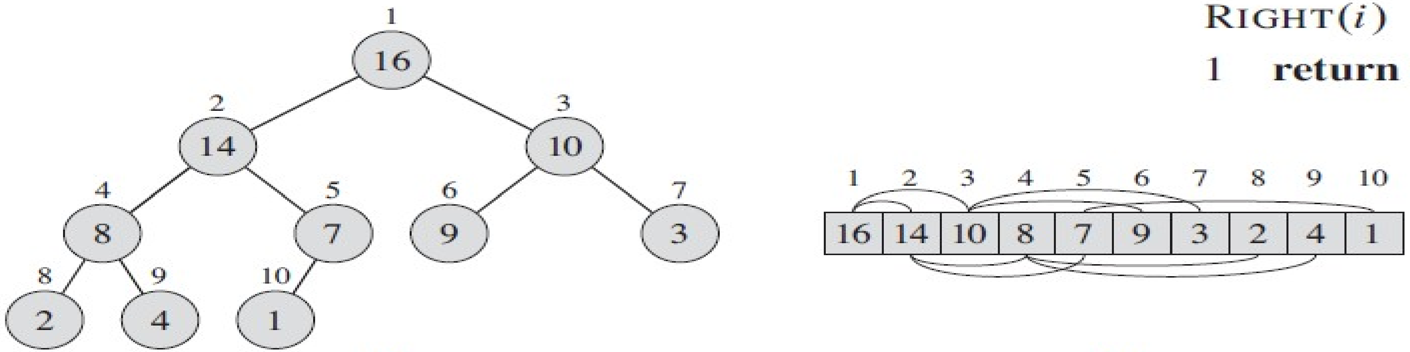
MAX-Heap

Notem que precisa começar no 1 !

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

LEFT(i)
    return $2i$

RIGHT(i)
    return $2i + 1$

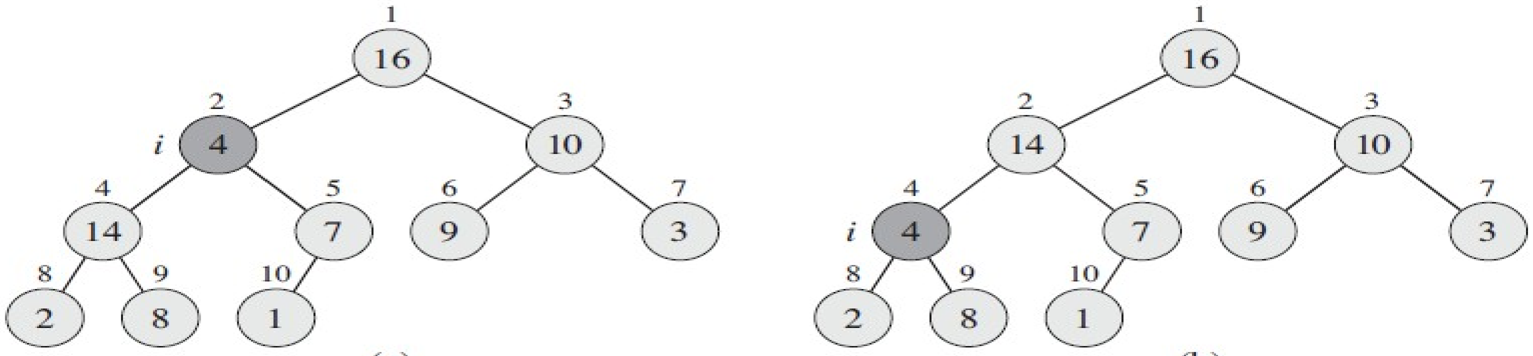


(a) (b)

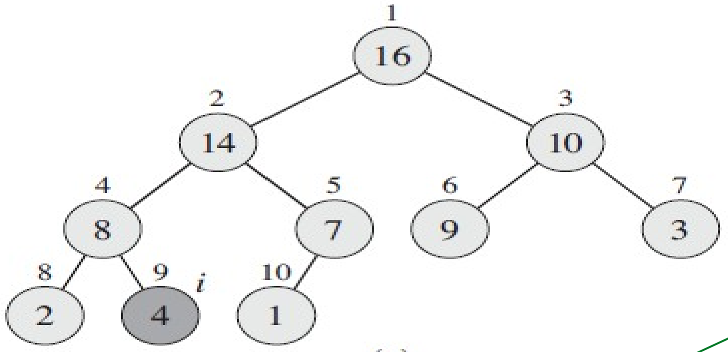
Visão LÓGICA do heap

O heap é um VETOR estático!

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



(a) (b)

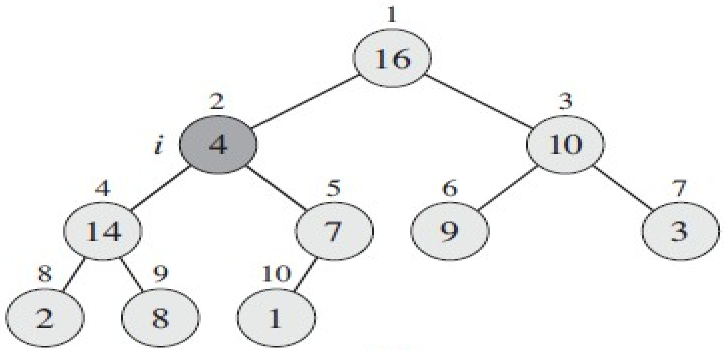


(c)

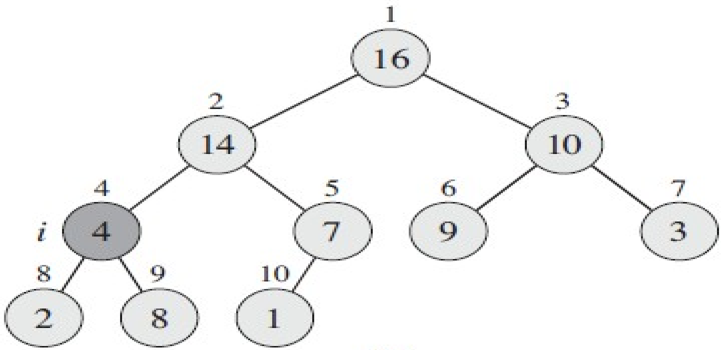
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
      largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
      largest = r
if largest ≠ i
      exchange A[i] with A[largest]
      MAX-HEAPIFY(A, largest)
```

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

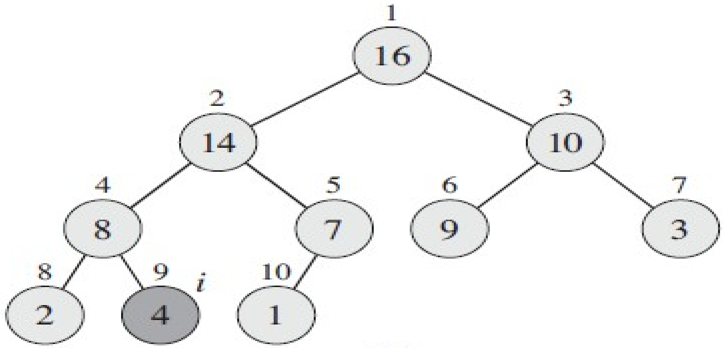
MAX-HEAPIFY$(A, 2)$, where $A.heap\text{-}size = 10$. Coloca o valor da atual posição i em um lugar adequado **para baixo**



(a)



(b)

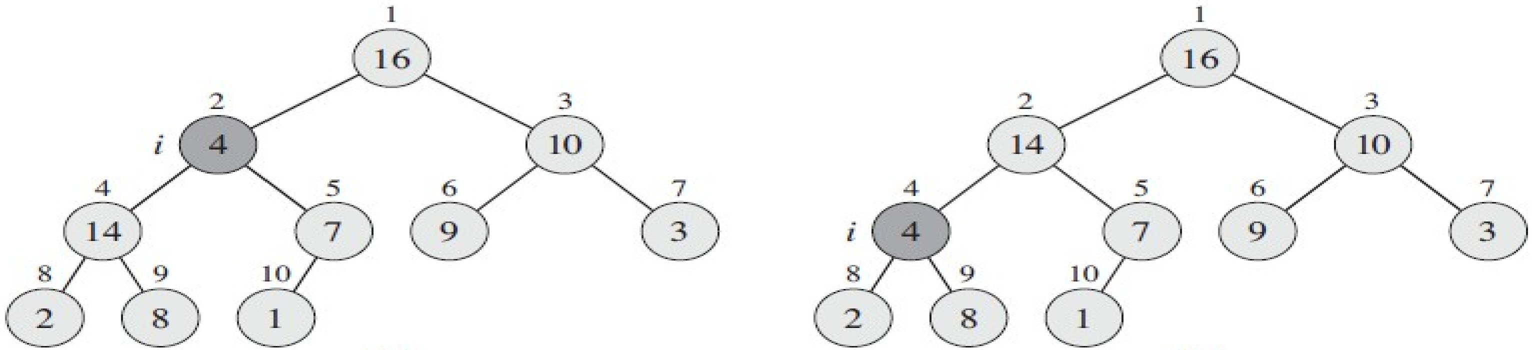


(c)

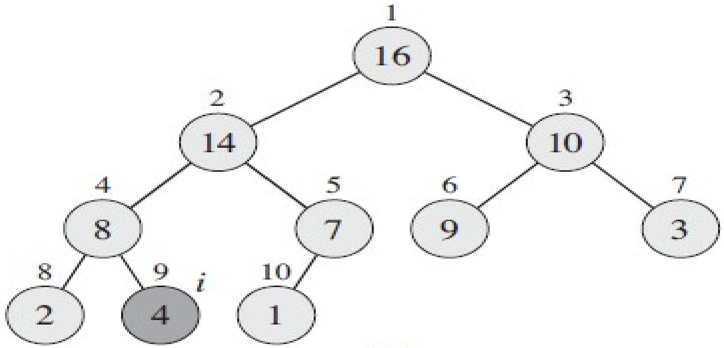
```
MAX-HEAPIFY(A, i)
l = LEFT(i)
r = RIGHT(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ≠ i
    exchange A[i] with A[largest]
    MAX-HEAPIFY(A, largest)
```

Complexidade: ?

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



(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)

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

Complexidade: O(n)

Construção do heap

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

- Analisando a complexidade:
  - cada chamada a Max-heapify tem T(n) = O(lg n) e existe O(n) chamadas. Então: *T(n) = O(n lg n).*
  - No entanto, é possível definir essa complexidade mais restritamente.
  - Se analisarmos a complexidade em função da altura da árvore, chegaremos a **O(n)**, pois afinal a complexidade do Max-heapify é O(h), sendo h a altura do nó I no qual é feita a primeira chamada do Max-heapify.

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)
```

- Analisando a complexidade:
  - a complexidade do Max-heapify é O(h), sendo h a altura do nó I no qual é feita a primeira chamada do Max-heapify.
  - Há nós com altura h variando de 0 até lg n
  - Há no máximo $\lceil n/2^{h+1} \rceil$ nós com uma certa altura h
  - Então a complexidade total é:

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

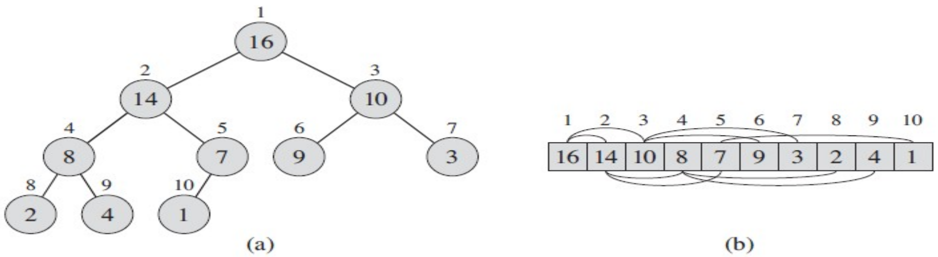
< 2, pois

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2 \, .$$

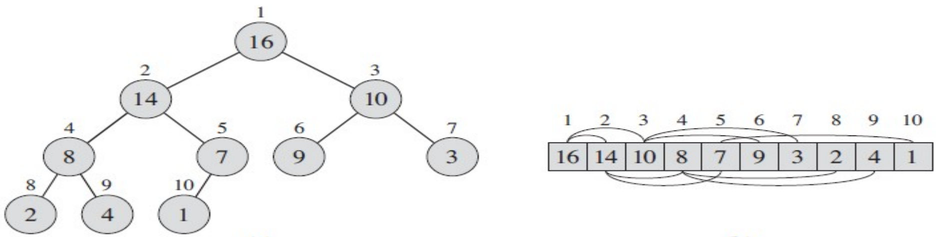
(x=1/2 na equação A.8 no livro do Cormen)

- Logo

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n) \, .$$

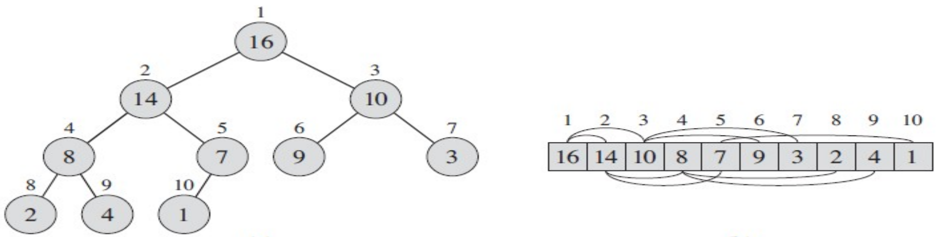


HEAP-EXTRACT-MAX(A)



(a) (b)

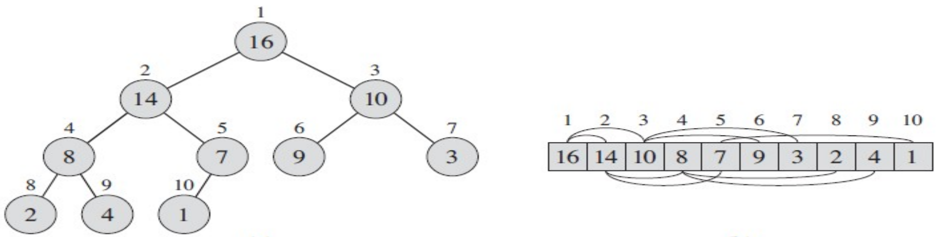
```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
      error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size − 1
MAX-HEAPIFY(A, 1)
return max
```



(a) (b)

```
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
```

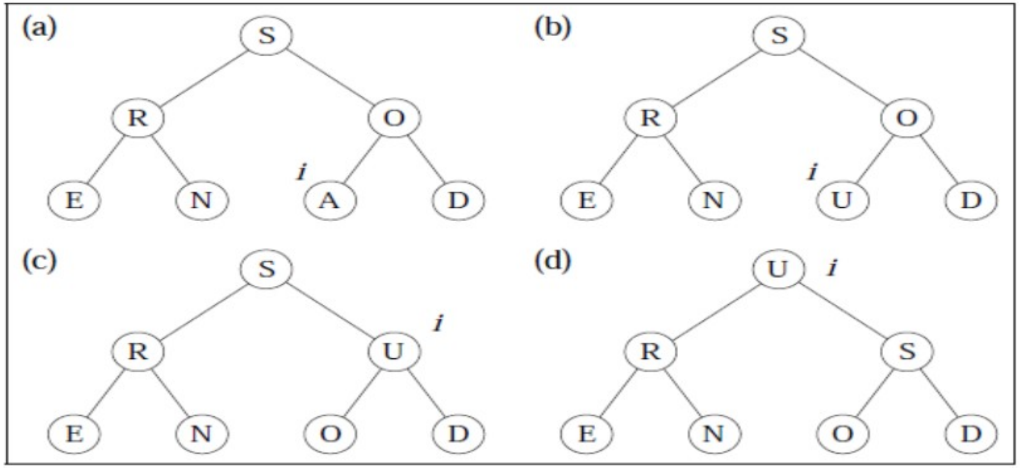
Complexidade: ?



(a) (b)

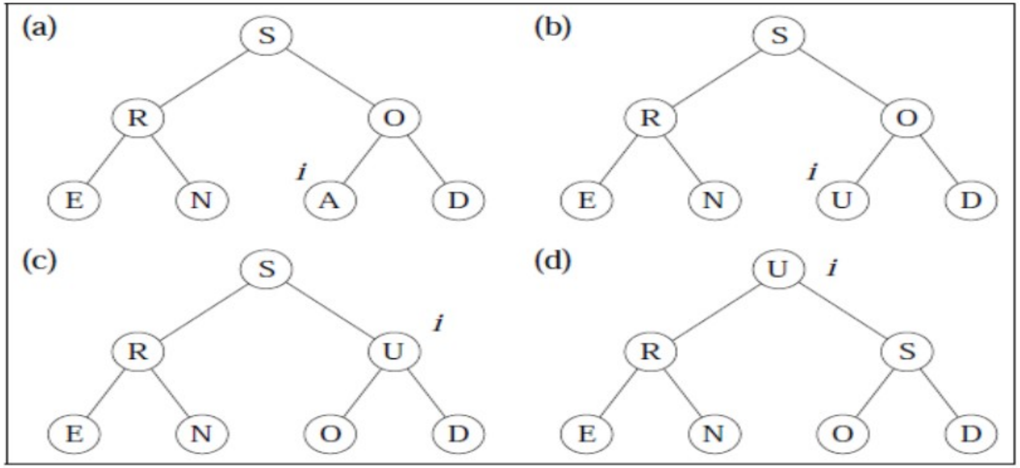
```
HEAP-EXTRACT-MAX(A)
if A.heap-size < 1
        error "heap underflow"
max = A[1]
A[1] = A[A.heap-size]
A.heap-size = A.heap-size - 1
MAX-HEAPIFY(A, 1)
return max
```

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)
```

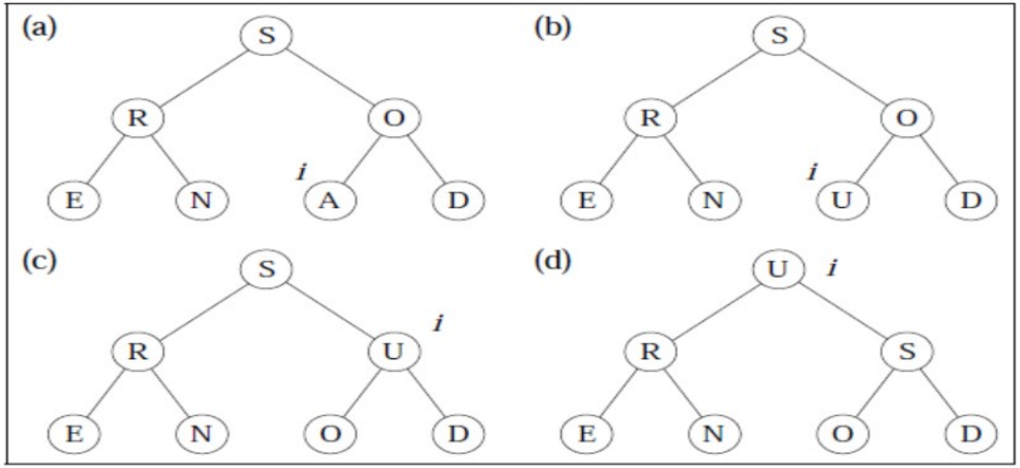
Isto é, potencialmente o elemento vai **subir** no heap




```
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)
```

Isto é, potencialmente o elemento vai **subir** no heap

Complexidade: ?



Isto é, potencialmente o elemento vai **subir** no heap

$\text{HEAP-INCREASE-KEY}(A, i, key)$

```
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 binário no algoritmo de Prim

Só que aqui precisaremos usar um **Heap mínimo** (pois o elemento de maior prioridade será o de menor chave)

```
MST-PRIM(G, w, r)
for each u ∈ V[G]
        do key[u] ← ∞
            π[u] ← NIL
key[r] ← 0
Q ← V[G]
while Q ≠ Ø
        do u ← EXTRACT-MIN(Q)
            for each v ∈ Adj[u]
                do if v ∈ Q and w(u, v) < key[v]
                    then π[v] ← u
                        key[v] ← w(u, v)
```

Construção do Heap (O(V))

Extrai do heap elemento de maior prioridade e reajuste o heap (O(lgV))

Vetor de bits (ou bools) para saber em O(1) se v está em Q

Aumenta prioridade de um elemento (O(lgV))

Complexidade

Depende da implementação de Q….

```
MST-PRIM(G, w, r)
1   for each u ∈ V[G]
2       do key[u] ← ∞
3          π[u] ← NIL
4   key[r] ← 0
5   Q ← V[G]
6   while Q ≠ Ø
7       do u ← EXTRACT-MIN(Q)
8          for each v ∈ Adj[u]
9              do if v ∈ Q and w(u, v) < key[v]
10                 then π[v] ← u
11                      key[v] ← w(u, v)
```

Se Q for um heap binário:

Linhas 1 a 5: O(V)
Loop da linha 6: V vezes
Linha 7: O(lg V)

Linha 6-7: O(V lg V)

Loop 8: O(A) no total
Linha 11: O(lg V)

Linhas 8-11: O(A lg V) no total (assumindo lista de adjacência)

Por que essa última igualdade?

Complexidade: O(V) + O(V lg V) +O(A lg V)
= O(A lg V)

Complexidade

Depende da implementação de Q….

```
MST-PRIM(G, w, r)
1   for each u ∈ V[G]
2        do key[u] ← ∞
3           π[u] ← NIL
4   key[r] ← 0
5   Q ← V[G]
6   while Q ≠ Ø
7        do u ← EXTRACT-MIN(Q)
8           for each v ∈ Adj[u]
9               do if v ∈ Q and w(u, v) < key[v]
10                     then π[v] ← u
11                          key[v] ← w(u, v)
```

Se Q for um heap binário:

Linhas 1 a 5: O(V)
Loop da linha 6: V vezes
Linha 7: O(lg V)

> Linha 6-7: O(V lg V)

Loop 8: O(A) no total
Linha 11: O(lg V)

> Linhas 8-11: O(A lg V) no total (assumindo lista de adjacência)

Por que essa última igualdade?
Assumindo que G é conexo…

Complexidade: O(V) + O(V lg V) +O(A lg V)
= O(A lg V)

Complexidade

Depende da implementação de Q….

```
MST-PRIM(G, w, r)
1   for each u ∈ V[G]
2        do key[u] ← ∞
3           π[u] ← NIL
4   key[r] ← 0
5   Q ← V[G]
6   while Q ≠ Ø
7        do u ← EXTRACT-MIN(Q)
8           for each v ∈ Adj[u]
9                do if v ∈ Q and w(u, v) < key[v]
10                     then π[v] ← u
11                          key[v] ← w(u, v)
```

Se Q for um heap binário:

Linhas 1 a 5: O(V)
Loop da linha 6: V vezes
Linha 7: O(lg V)

Linha 6-7: O(V lg V)

Loop 8: O(A) no total
Linha 11: O(lg V)

Linhas 8-11: O(A lg V) no total (assumindo lista de adjacência)

Bem melhor (que O(V^2)) se o grafo não for denso...

Complexidade: O(V) + O(V lg V) +O(A lg V)
= O(A lg V)

Complexidade

Depende da implementação de Q….

```
MST-PRIM(G, w, r)
 1  for each u ∈ V[G]
 2       do key[u] ← ∞
 3          π[u] ← NIL
 4  key[r] ← 0
 5  Q ← V[G]
 6  while Q ≠ Ø
 7       do u ← EXTRACT-MIN(Q)
 8          for each v ∈ Adj[u]
 9              do if v ∈ Q and w(u, v) < key[v]
10                    then π[v] ← u
11                         key[v] ← w(u, v)
```

Se Q for um heap binário:

Linhas 1 a 5: O(V)
Loop da linha 6: V vezes
Linha 7: O(lg V)

Linha 6-7: O(V lg V)

Loop 8: O(A) no total
Linha 11: O(lg V)

Linhas 8-11: O(A lg V) no total (assumindo lista de adjacência)

Complexidade: O(V) + O(V lg V) +O(A lg V)
= O(A lg V)

Se usar heap Fibonacci, O(A + V lg V),
que é ainda melhor caso quando |V| < |A|

Cormen cap 19 (3ª. ed)

Algoritmo de Kruskal

Algoritmo de Kruskal

Uma floresta A contém inicialmente todos os vértices isolados (cada vértice é um componente conectado)

A cada passo, é adicionada uma aresta segura:

- a aresta de menor peso que conecta dois componentes conectados DISTINTOS

Algoritmo de Kruskal

Uma floresta A contém inicialmente todos os vértices isolados (cada vértice é um componente conectado)

A cada passo, é adicionada uma aresta segura:

- a aresta de menor peso que conecta dois componentes conectados DISTINTOS
- até que a floresta se torne uma árvore (ou seja, que haja apenas um componente conectado)

Por eficiência, cada componente conectado é representado por uma estrutura de dados eficiente que implementa conjuntos disjuntos.

Algoritmo de Kruskal

```
MST-KRUSKAL(G, w)
    A = Ø
    for each vertex v ∈ G.V
        MAKE-SET(v)
    sort the edges of G.E into nondecreasing order by weight w
    for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
    return A
```

Cria um conjunto disjunto contendo apenas v

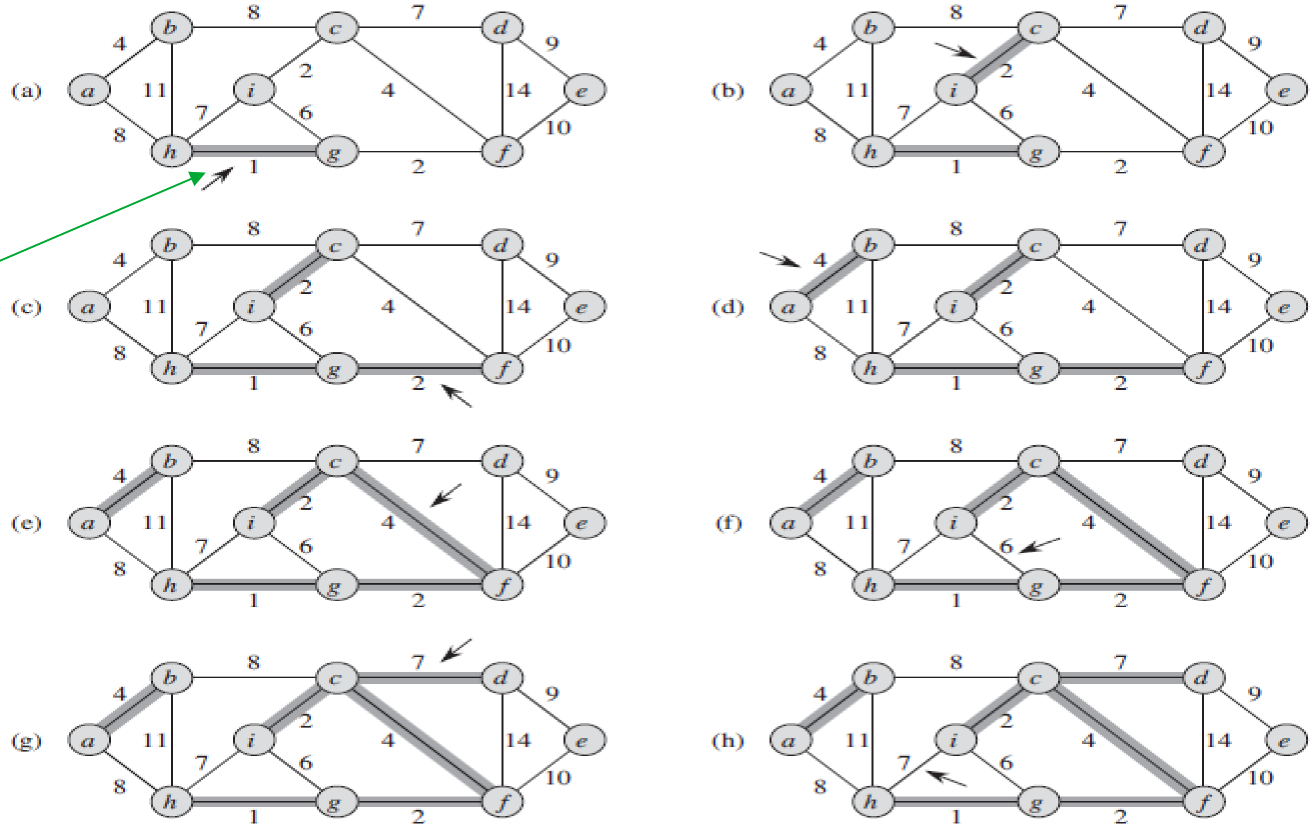
Encontra o conjunto daquele vértice

Une os conjuntos de u e de v em um só

Algoritmo de Kruskal

```
MST-KRUSKAL(G, w)
A = Ø
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

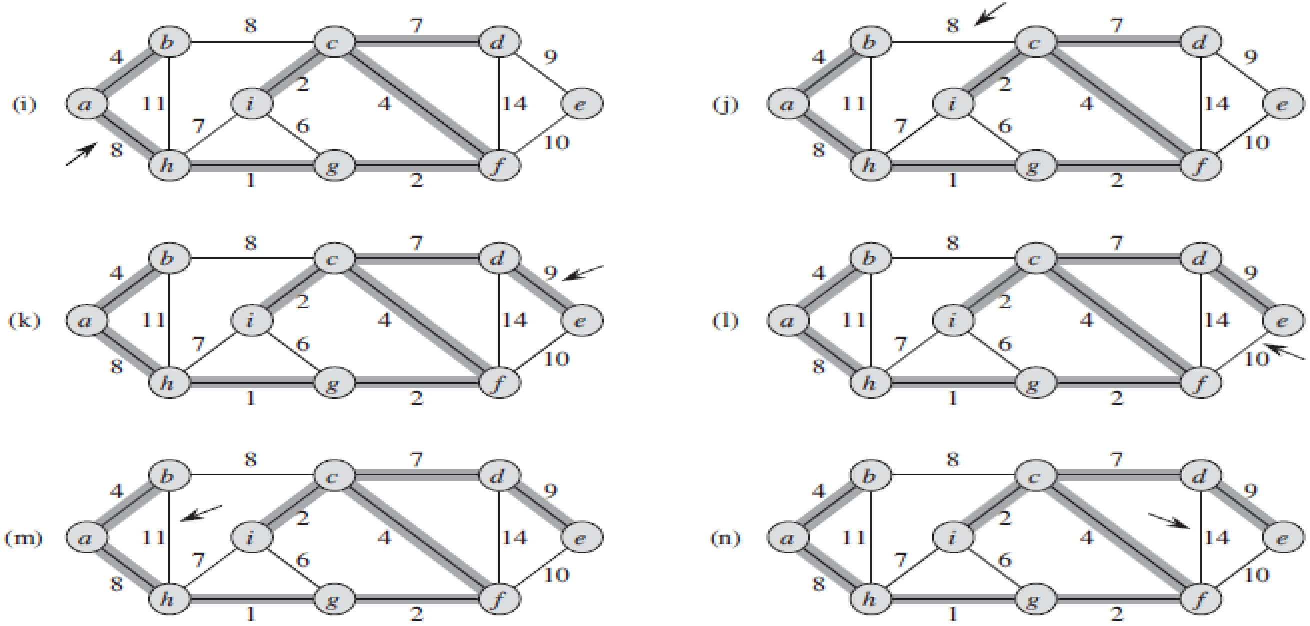
Essas arestas mais grossas conecta vértices de um mesmo componente conectado



Profa. Ariane Machado Lima

Algoritmo de Kruskal

```
MST-KRUSKAL(G, w)
    A = ∅
    for each vertex v ∈ G.V
        MAKE-SET(v)
    sort the edges of G.E into nondecreasing order by weight w
    for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
        if FIND-SET(u) ≠ FIND-SET(v)
            A = A ∪ {(u, v)}
            UNION(u, v)
    return A
```



Conjuntos disjuntos

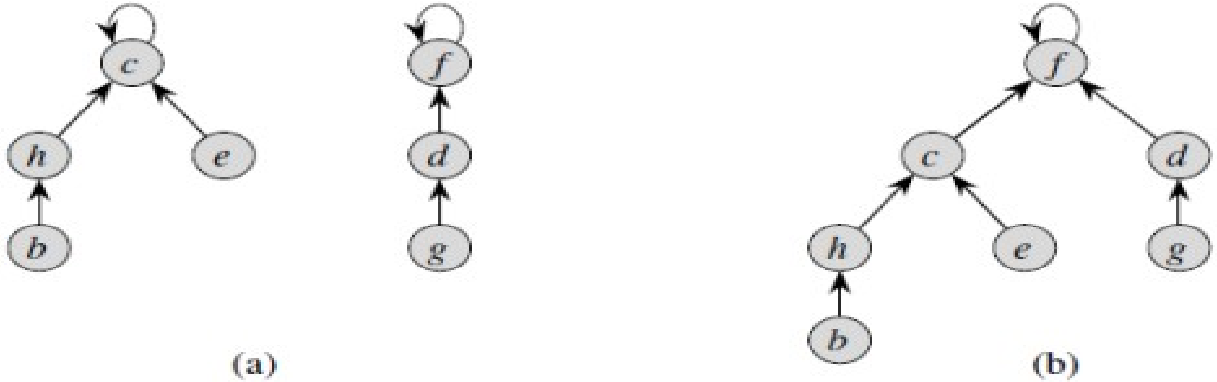
A complexidade do algoritmo de Kruskal depende da eficiência da estrutura de dados para conjuntos disjuntos (cap 21 do livro do Cormen).

Também conhecidos como “Union-Find”

Pseudo-código para florestas de conjuntos disjuntos com heurísticas de união ponderada pelo posto e compressão de caminho

Referência: CORMEN, H.T.; LEISERSON, C.E.; RIVEST, R.L. Introduction to Algorithms, MIT Press, McGraw-Hill, 1999, pp. 505-509.

Floresta de conjuntos disjuntos: cada árvore corresponde a um conjunto. O elemento situado na raiz da árvore é o representante do conjunto.



Representação de uma floresta de conjuntos disjuntos.
(a) Duas árvores representando dois conjuntos disjuntos: {c, h, e, b} e {f, d, g};
(b) Árvore resultante da chamada da união dos dois conjuntos.

Notação:

***x*:**	denota um elemento
***p*[x]:**	pai do nó referente ao elemento x.
***rank*[x]:**	limitante superior para a altura de x (comprimento do caminho entre x e a folha descendente mais distante). Também chamado de "posto"

Resumo das funções:

MAKE-SET(x): Cria um conjunto unitário contendo apenas o elemento x.

UNION (x, y): Une os conjuntos aos quais x e y pertencem. Assume-se que esses conjuntos sejam disjuntos.

LINK (x, y): Supondo que x e y sejam as raízes de duas árvores distintas (ou seja, representantes de seus respectivos subconjuntos), esta função torna x um filho de y ou vice-versa.

FIND_SET (x): Retorna o representante do elemento x (ou seja, o elemento situado na raiz da árvore em que x se encontra).

```
MAKE-SET(x)
p[x] ← x
rank[x] ← 0
```

rank[f] = 0

Complexidade: ?

```
MAKE-SET(x)
p[x] ← x
rank[x] ← 0
```

rank[f] = 0

Complexidade: O(1)

```
MAKE-SET(x)
p[x] ← x
rank[x] ← 0
```

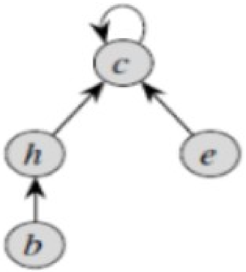
Complexidade: O(1)

rank[f] = 0

Uma **possível** implementação do Find-SET:

```
FIND-SET(x)
se x != p[x]
        retorna FIND-SET(p[x])
retorna x
```

Complexidade: ?



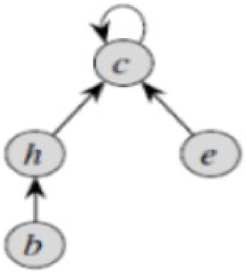
```
MAKE-SET(x)
p[x] ← x
rank[x] ← 0
```

rank[f] = 0

Complexidade: O(1)

Uma **possível** implementação do Find-SET:

```
FIND-SET(x)
se x != p[x]
        retorna FIND-SET(p[x])
retorna x
```



Complexidade: O(n)? (se a árvore for uma linguiça...)

MAKE-SET(x)

$p[x] \leftarrow x$

$rank[x] \leftarrow 0$

Complexidade: O(1)

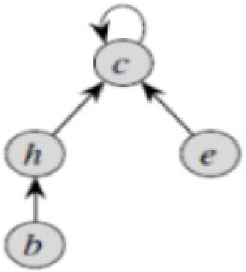
Uma **possível** implementação do Find-SET:

FIND-SET(x)

se x != p[x]

retorna FIND-SET(p[x])

retorna x



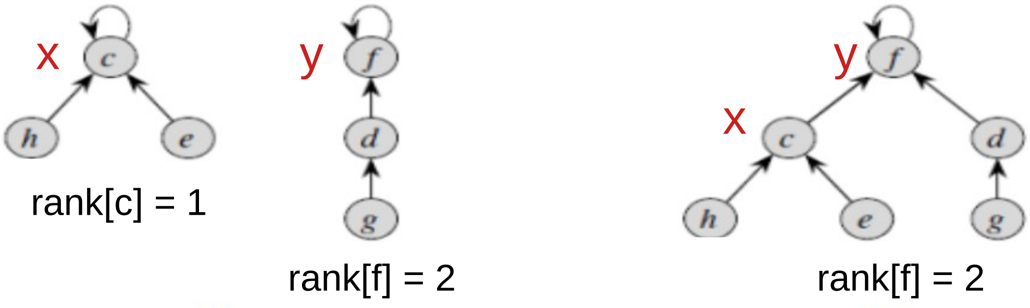
Complexidade: O(n)? Depende de como faço a união...

UNION(x, y)

LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y) → Procura não deixar a árvore mais alta do que poderia

```
if rank[x] > rank[y]
    then p[y] ← x
    else p[x] ← y
         if rank[x] = rank[y]
            then rank[y] ← rank[y] + 1
```



(a) (b)

Complexidade: ?

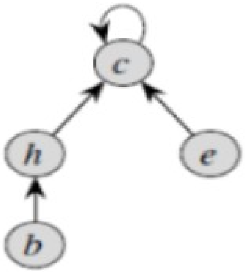
```
MAKE-SET(x)
    p[x] ← x
    rank[x] ← 0
```

rank[f] = 0

Complexidade: O(1)

Uma **possível** implementação do Find-SET:

```
FIND-SET(x)
    se x != p[x]
        retorna FIND-SET(p[x])
    retorna x
```



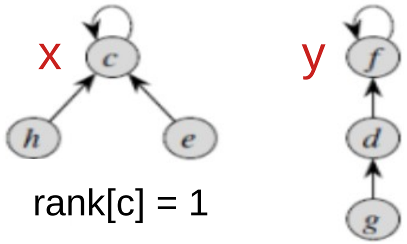
Complexidade: O(n)? Depende de como faço a união...

```
UNION(x, y)
    LINK(FIND-SET(x), FIND-SET(y))
```

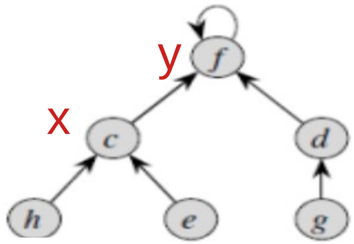
```
LINK(x, y)
    if rank[x] > rank[y]
        then p[y] ← x
        else p[x] ← y
             if rank[x] = rank[y]
                then rank[y] ← rank[y] + 1
```

→ Procura não deixar a árvore mais alta do que poderia

Complexidade: O(1)



(a)



(b)

MAKE-SET(x)

$p[x] \leftarrow x$

$rank[x] \leftarrow 0$

rank[f] = 0

Complexidade: O(1)

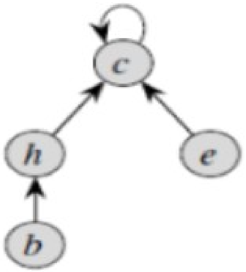
Uma **possível** implementação do Find-SET:

FIND-SET(x)

se x != p[x]

retorna FIND-SET(p[x])

retorna x



Complexidade: O(n)? Depende de como faço a união...

UNION(x, y)

LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y) → Procura não deixar a árvore mais alta do que poderia

if $rank[x] > rank[y]$

  then $p[y] \leftarrow x$

  else $p[x] \leftarrow y$

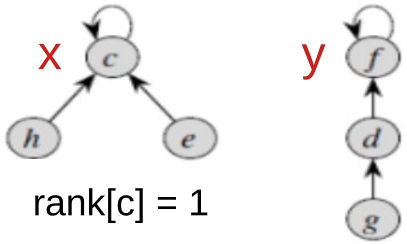
   if $rank[x] = rank[y]$

    then $rank[y] \leftarrow rank[y] + 1$

Complexidade: O(1)

Heurística de união pelo posto (rank)

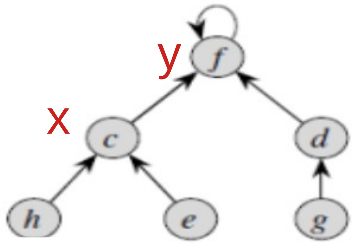
Isso acelera os find-sets...



rank[c] = 1

rank[f] = 2

(a)



rank[f] = 2

(b)

MAKE-SET(x)

$p[x] \leftarrow x$

$rank[x] \leftarrow 0$

rank[f] = 0

Complexidade: O(1)

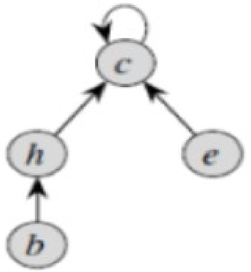
Uma **possível** implementação do Find-SET:

FIND-SET(x)

se x != p[x]

retorna FIND-SET(p[x])

retorna x



Complexidade: O(n)? Depende de como faço a união...

UNION(x, y)

LINK(FIND-SET(x), FIND-SET(y))

Complexidade: ?

LINK(x, y) → Procura não deixar a árvore mais alta do que poderia

if $rank[x] > rank[y]$

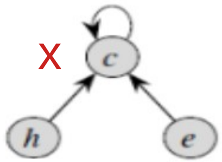
then $p[y] \leftarrow x$

else $p[x] \leftarrow y$

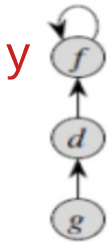
if $rank[x] = rank[y]$

then $rank[y] \leftarrow rank[y] + 1$

Complexidade: O(1)

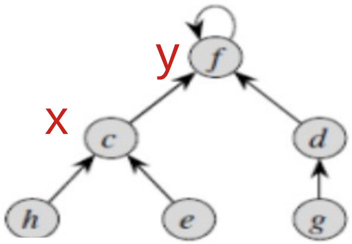


rank[c] = 1



rank[f] = 2

(a)



rank[f] = 2

(b)

Profa. Ariane Machado Lima

MAKE-SET(x)

$p[x] \leftarrow x$

$rank[x] \leftarrow 0$

Complexidade: O(1)

rank[f] = 0

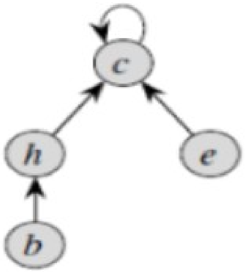
Uma **possível** implementação do Find-SET:

FIND-SET(x)

se x != p[x]

retorna FIND-SET(p[x])

retorna x



Complexidade: O(n)? Depende de como faço a união...

UNION(x, y)

LINK(FIND-SET(x), FIND-SET(y))

Complexidade: Depende do FIND-SET...

LINK(x, y) → Procura não deixar a árvore mais alta do que poderia

if $rank[x] > rank[y]$

then $p[y] \leftarrow x$

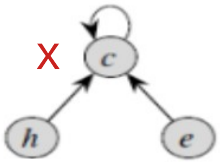
else $p[x] \leftarrow y$

if $rank[x] = rank[y]$

then $rank[y] \leftarrow rank[y] + 1$

Complexidade: O(1)

x

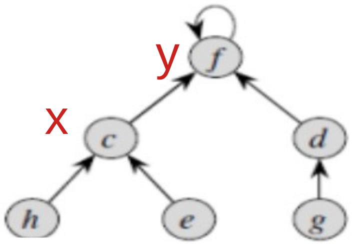


rank[c] = 1

y

rank[f] = 2

(a)

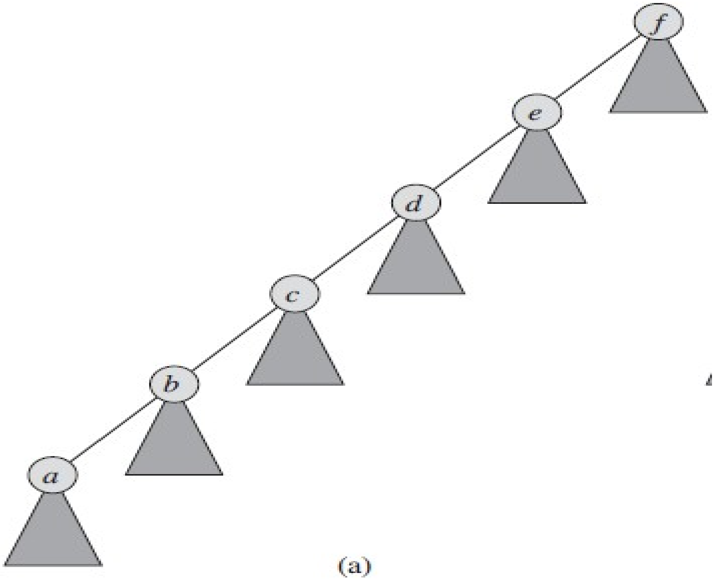


rank[f] = 2

(b)

Sempre que eu ligo duas raízes com o mesmo rank eu aumento o rank da raiz do novo conjunto

Após sucessivos aumentos, a árvore pode ficar bem desbalenceada e com uma altura grande



(a)

Uma **possível** implementação do Find-SET:

FIND-SET(x)

se x != p[x]

retorna FIND-SET(p[x])

retorna x

Pode-se aproveitar as buscas para melhorar o balanceamento da árvore!

VERSÃO ANTIGA...

```
FIND-SET(x)
    se x ≠ p[x]
        retorna FIND-SET(p[x])
    retorna x
```

```
FIND-SET(x)
1   if x ≠ p[x]          ie, se x não for a raiz
2       then p[x] ← FIND-SET(p[x])
3   return p[x]
```

VERSÃO NOVA! (Heurística de compressão de caminho)

Efeito de um FIND-SET(a)

```
FIND-SET(x)
if x ≠ p[x]
    then p[x] ← FIND-SET(p[x])
return p[x]
```

VERSÃO NOVA!
(Heurística de compressão de caminho)

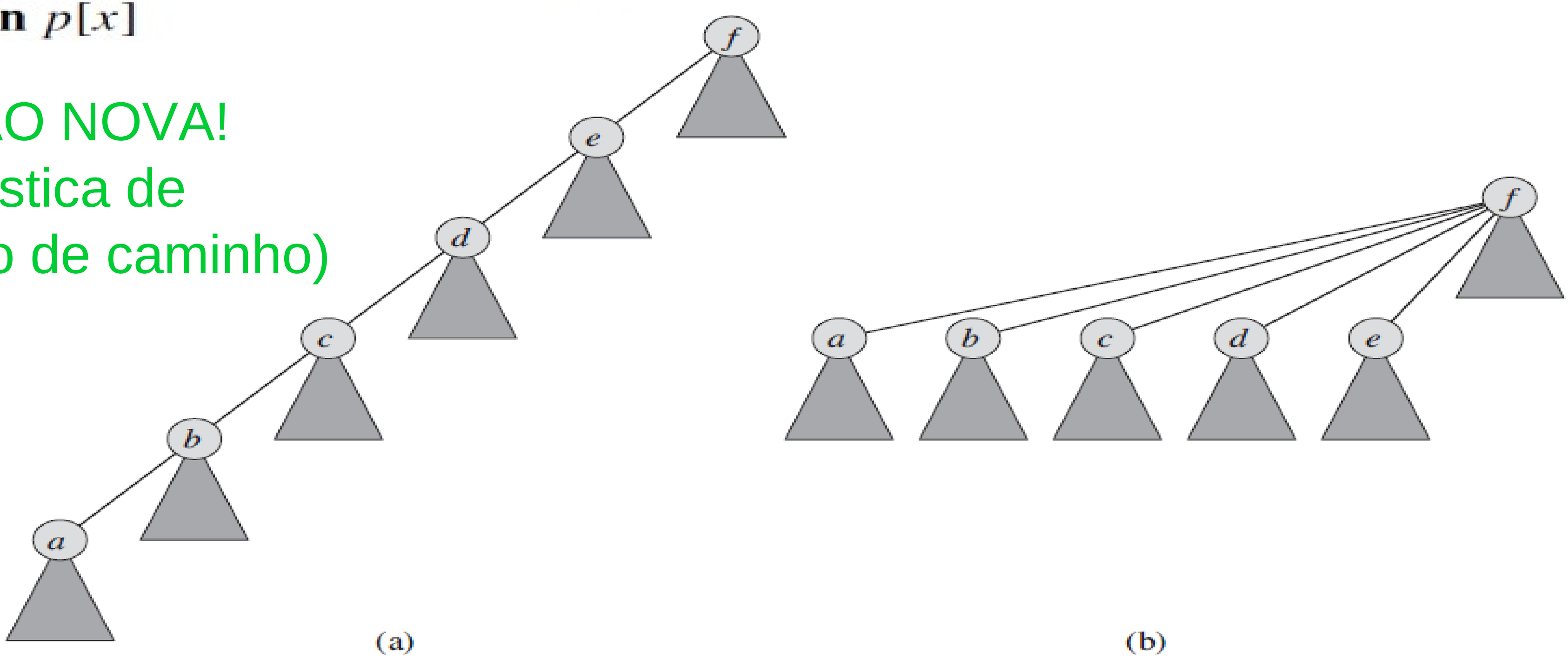


Figure 21.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

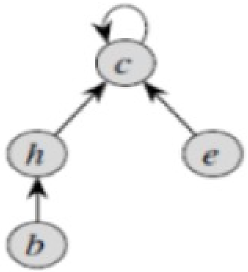
```
MAKE-SET(x)
    p[x] ← x
    rank[x] ← 0
```

rank[f] = 0

Complexidade: O(1)

Versão nova do Find-SET:

```
FIND-SET(x)
    se x != p[x]
        p[x] ← FIND-SET(p[x])
    retorna p[x]
```



Complexidade: Depende das uniões e chamadas do FIND-SET...

```
UNION(x, y)
    LINK(FIND-SET(x), FIND-SET(y))
```

Complexidade: Depende do FIND-SET...

```
LINK(x, y)
    if rank[x] > rank[y]
        then p[y] ← x
        else p[x] ← y
            if rank[x] = rank[y]
                then rank[y] ← rank[y] + 1
```

Procura não deixar a árvore mais alta do que poderia

Complexidade: O(1)

rank[c] = 1

rank[f] = 2

(a)

rank[f] = 2

(b)

Complexidade do Union-Find (análise amortizada)

Considerando as duas heurísticas juntas (união por posto e compressão de caminho), a complexidade depende do conjunto de chamadas às operações…

Para m chamadas no total, das quais n são chamadas a MAKE-SET (portanto no máximo n-1 chamadas a UNION, e o restante de chamadas a FIND-SET):

- Complexidade total: O(m α(n)), sendo α(n) uma função que cresce muito lentamente (α(n) <= 4 para a grande maioria das aplicações práticas)

(Cormen 3ª ed. - seções 21.3 e 21.4)

Algoritmo de Kruskal - Complexidade

```
MST-KRUSKAL(G, w)
1  A = Ø
2  for each vertex v ∈ G.V
3      MAKE-SET(v)
4  sort the edges of G.E into nondecreasing order by weight w
5  for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
6      if FIND-SET(u) ≠ FIND-SET(v)
7          A = A ∪ {(u, v)}
8          UNION(u, v)
9  return A
```

E = A: conjunto de arestas

L. 4: O(A lg A)

L. 2-3 + loop L.5: |V| MAKE-SET's + O(A) FIND-SET's e UNION's =

O((V+A) α(V)), sendo α(V) é uma função que cresce muito lentamente, menos que o log(V)

Como G é conectado → |A| >= |V| -1 → O(A α(V))

Como α(V) = O(lg V) = O(lg A), complexidade total: O(A lg A) + O(A α(V)) = O(A lg A)

Como $|A| < |V|^2$ → $\lg |A| < \lg (|V|^2)$ → $\lg |A| < 2 \lg |V|$ → $\lg |A| = O(\lg V)$

→ Complexidade total O(A lg V) (a mesma do alg. de Prim)

Observação

Embora os algoritmos de Prim e Kruskal tenham a mesma complexidade assintótica, o algoritmo de Kruskal tende a executar mais lentamente para grafos densos, por conta da ordenação de arestas.

Referências

Livro do Cormen (3^a ed):

- cap 23 (AGM)
- seções 6.1 a 6.3 (Heaps binários)
- cap 21 (Union-Find)

Livro do Ziviani seção 7.8