

# Grafos – Aula 9

## Roteiro

- Corretude de Dijkstra
- Fila de prioridades e Heap
- Complexidade de Dijkstra
- Tempo de execução

# Algoritmo de Dijkstra

## ■ Ideias

- Manter dois conjuntos de vértices (explorado e não-explorado)
  - Manter comprimento do menor caminho conhecido até o momento para cada vértice
  - Avaliar vértice de menor caminho (para o conjunto explorado)
  - Atualizar distâncias
- Como tornar a ideia em algoritmo?

# Algoritmo de Dijkstra

1. `Dijkstra(G, s)`
2. Para cada vértice  $v$
3. `dist[v] = infinito`
4. Define conjunto  $S = \emptyset$  // inicia vazio
5. `dist[s] = 0`
6. Enquanto  $S \neq V$
7. Selecione  $u$  em  $V-S$ , tal que `dist[u]` é mínima
8. Adicione  $u$  em  $S$
9. Para cada vizinho  $v$  de  $u$  faça
10. Se `dist[v] > dist[u] + w(u,v)` então
11. `dist[v] = dist[u] + w(u,v)`
12. Retorna `dist[]`

■  $S$  = conjunto explorado

■  $V-S$  = conjunto não-explorado ( $V$  = conjunto de vértices)

■ `dist[v]` = comprimento do menor caminho conhecido pelo algoritmo de  $s$  até  $v$  (inicialmente, é infinito)

■  $w(u,v)$  = peso da aresta  $(u,v)$

# Corretude do Algoritmo

- Provar que algoritmo sempre produz resultado correto
  - calcula a distância de  $s$  a qualquer outro vértice

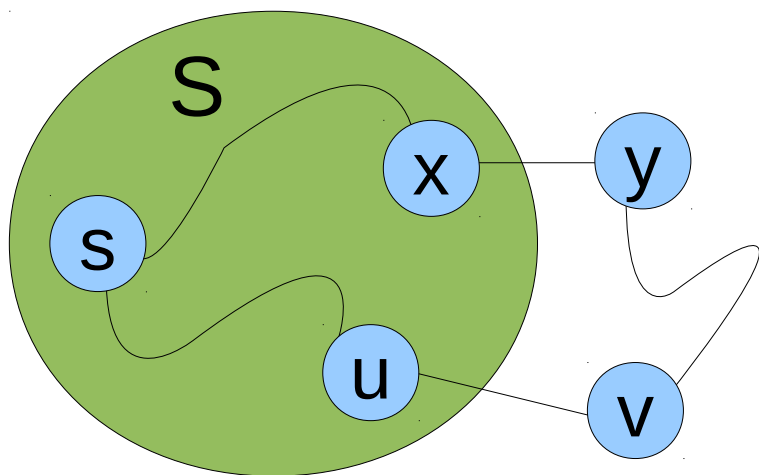
## Teorema

- Considere um vértice  $u$  pertencente ao conjunto  $S$  em qualquer instante do algoritmo. Temos que “ $\text{dist}[u]$ ” é a distância entre  $s$  e  $u$ .

# Prova (1/3)

- Prova por indução no tamanho de  $S$
- Caso base:  $|S| = 1$ 
  - $S = \{s\}$ ,  $\text{dist}[s] = 0$ , devido inicialização
- Hipótese:  $|S| = k$ 
  - para  $|S| = k$ , assumamos que  $\text{dist}[u]$  é igual a distância entre  $s$  e  $u$
- Caso geral:  $|S| = k + 1$ 
  - $k+1$  adiciona  $v$  à  $S$ , dando origem a  $P_v$
  - temos que mostrar que  $P_v$  é mínimo

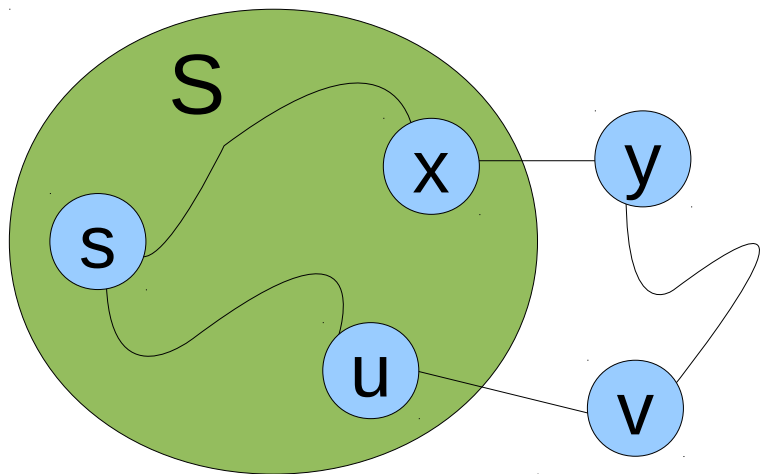
# Prova (2/3)



- $v$  é o novo vértice, do passo  $k+1$
- seja  $(u,v)$  última aresta no caminho  $P_v$
- pela hipótese,  $P_u$  é caminho mínimo  $s-u$

- Considere outro caminho  $s-v$ ,  $P$ , que não passa por  $u$
- Precisamos provar que  $P$  é maior (ou igual) a  $P_v$
- $P$  passa pela aresta  $(x,y)$ , com  $x$  em  $S$  e  $y$  fora de  $S$  (para algum  $x$  e  $y$  qualquer)

# Prova (3/3)



- No passo  $k+1$ , algoritmo escolhe  $v$  para adicionar a  $S$  (e não  $y$ )
- Então, caminho  $s \rightarrow y$  é maior ou igual a  $P_v$
- Como  $\text{dist}(y, v) \geq 0$ , caminho  $P$  será maior ou igual a  $P_v$
- Logo,  $P_v$  é caminho mínimo e  $\text{dist}[v]$  será distância até  $v$ 
  - $\text{dist}[v]$  não é alterada pelo algoritmo após vértice entrar no conjunto  $S$

# Complexidade

```
1. Dijkstra(G, s)
2.   Para cada vértice v
3.     dist[v] = infinito
4.   Define conjunto S = ∅ // inicia vazio
5.   dist[s] = 0
6.   Enquanto S ≠ V
7.     Selecione u em V-S, tal que dist[u] é mínima
8.     Adicione u em S
9.     Para cada vizinho v de u faça
10.      Se dist[v] > dist[u] + w(u,v) então
11.        dist[v] = dist[u] + w(u,v)
12. Retorna dist[]
```

- Percorre todas arestas do grafo (linha 6 + linha 9)
- Depende do tempo para escolher u (linha 7)



# Complexidade

- Algoritmo simples
  - Percorre vértices e encontra  $\text{dist}[u]$  mínimo
  - Complexidade do algoritmo:  $O(n^2)$

**Outra ideia?**

- Fila de prioridades
  - Chave: distâncias, conteúdo: id do vértice
  - Extrair o mínimo (vértice com menor distância)
  - Atualizar chaves (distâncias)

# Fila de Prioridade

- Estrutura de dados poderosa (*priority queue*)
- Mantém um conjunto de elementos  $S$
- Cada elemento possui uma chave (número de prioridade)
- Permite inserir, remover e modificar elementos de  $S$  através da chave
- Complexidade de *pior caso* para remoção, inserção e modificação:  $O(\log n)$ 
  - caso médio para algumas operações pode ser  $O(1)$ , dependendo da implementação (ex. *Fibonacci Heap*)

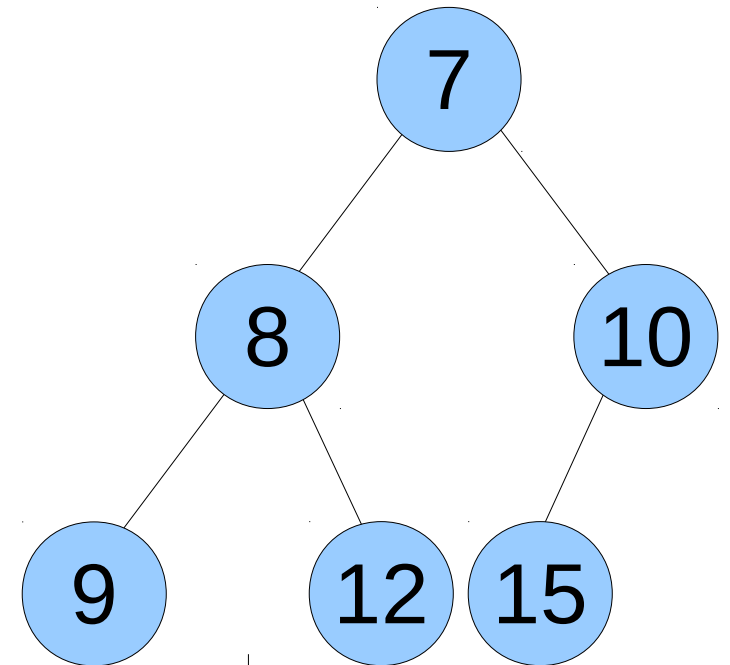
# Heap

Heap: Implementação de fila de prioridade

- Estrutura de dados em forma de árvore
- Armazena conjunto de elementos, todos associados a uma chave
- Chave dos elementos define a árvore
- *Heap order*
  - 1) Min-heap:  $\text{chave}(u) \leq \text{chave}(v)$ , quando  $u$  é pai de  $v$  (usado para remover menor chave primeiro)
  - 2) Max-heap:  $\text{chave}(u) \geq \text{chave}(v)$ , quando  $u$  é pai de  $v$  (usado para remover maior chave primeiro)

# Heap Binário

- Estrutura organizada em uma árvore binária cheia
  - fácil implementação
- Operações básicas
  - adicionar, remover, atualizar chave
- Não serve para fazer busca (não está ordenada)



Max-heap ou  
Min-heap?  
(valor mostrado é a  
prioridade)

Saindo da raiz, como  
chegar na chave 12?

# Inserção e Remoção

## Inserção

- 1) adicionar elemento na última posição da árvore
- 2) trocar posição com pai até garantir *heap-order* (subir)

## Atualização

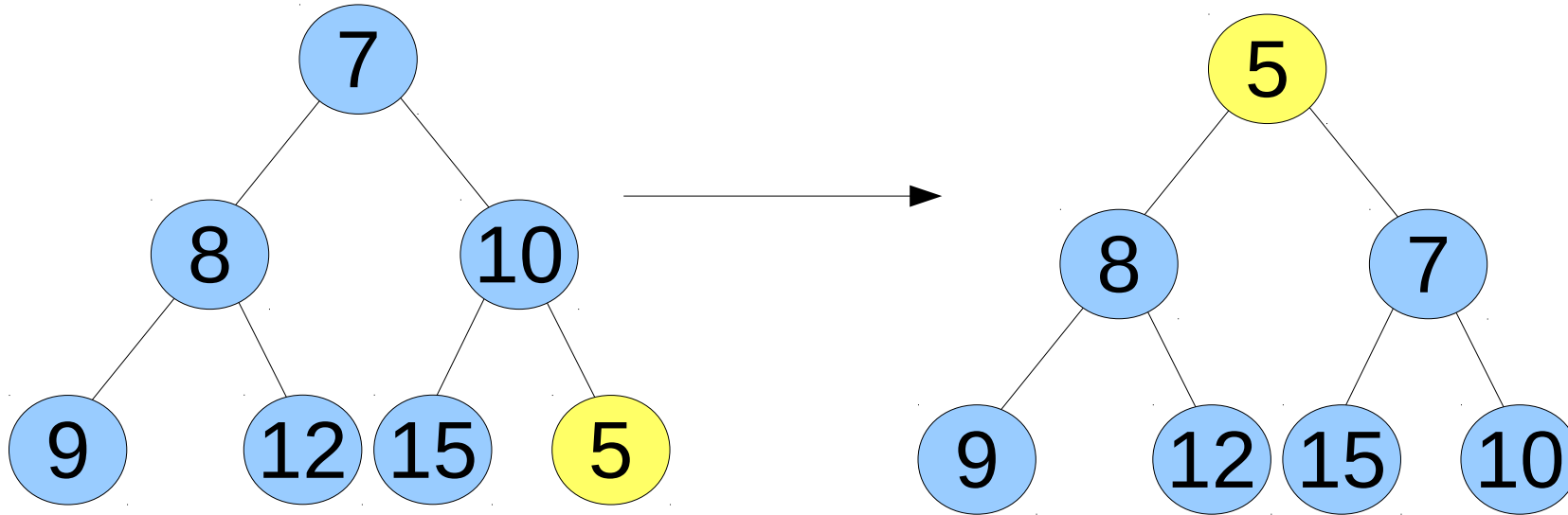
- Chave muda de valor acessando diretamente o elemento
- Subir ou descer, de acordo até garantir *heap-order*

## Remoção

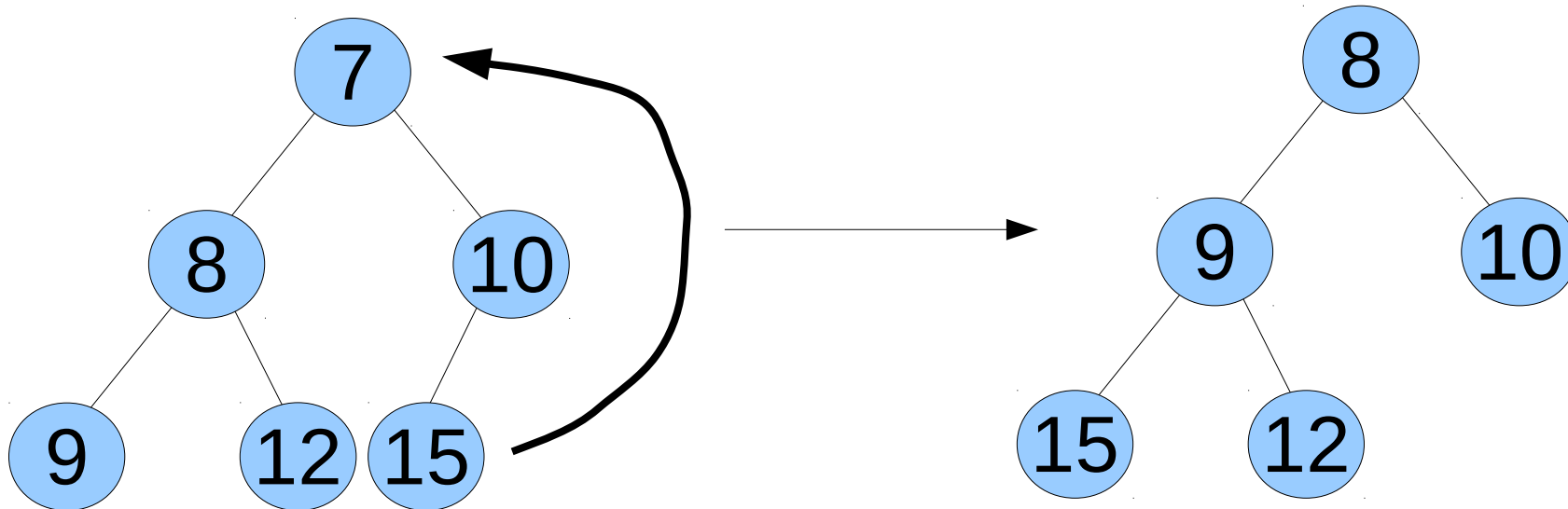
- 1) remover raiz e mover último elemento para raiz
- 2) trocar de posição com menor filho até garantir *heap-order* (descer)

# Exemplos

## ■ Adicionar chave 5

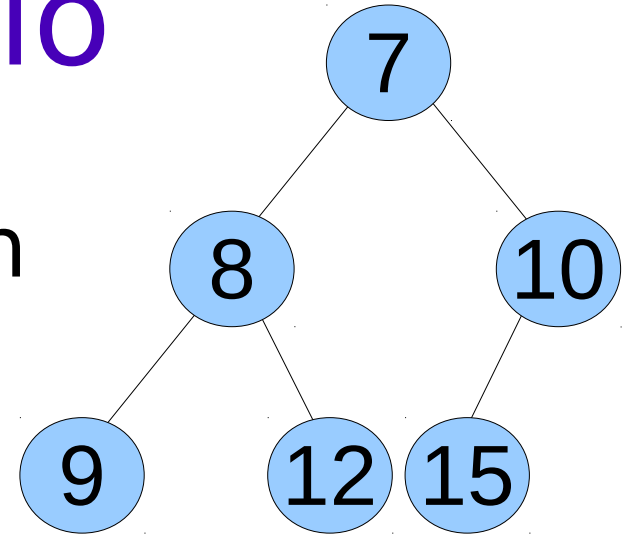


## ■ Remover raiz



# Heap Binário

- Considere heap binário com  $n$  elementos
- Altura da árvore:  $\log_2 n$
- Complexidade das operações de inserção, remoção e atualização?
- Subir ou descer na árvore
  - pior caso percorre todos os níveis,  $\log_2 n$
- Complexidade:  $O(\log n)$
- Pode ser implementado com um vetor!



# Dijkstra com Heap

- Armazenar  $\text{dist}[v]$  também em um heap
  - chave é distância, conteúdo é id do vértice
- Heap tem  $n$  elementos
- Operações no heap

```
1. Dijkstra(G, s)
2.   Para cada vértice v
3.     dist[v] = infinito
4.   Define conjunto S = ∅ // inicia vazio
5.   dist[s] = 0
6.   Enquanto S ≠ V
7.     Selecione u em V-S, tal que dist[u] é mínima
8.     Adicione u em S
9.     Para cada vizinho v de u faça
10.      Se dist[v] > dist[u] + w(u,v) então
11.        dist[v] = dist[u] + w(u,v)
12. Retorna dist[]
```

inserção



remoção



atualização



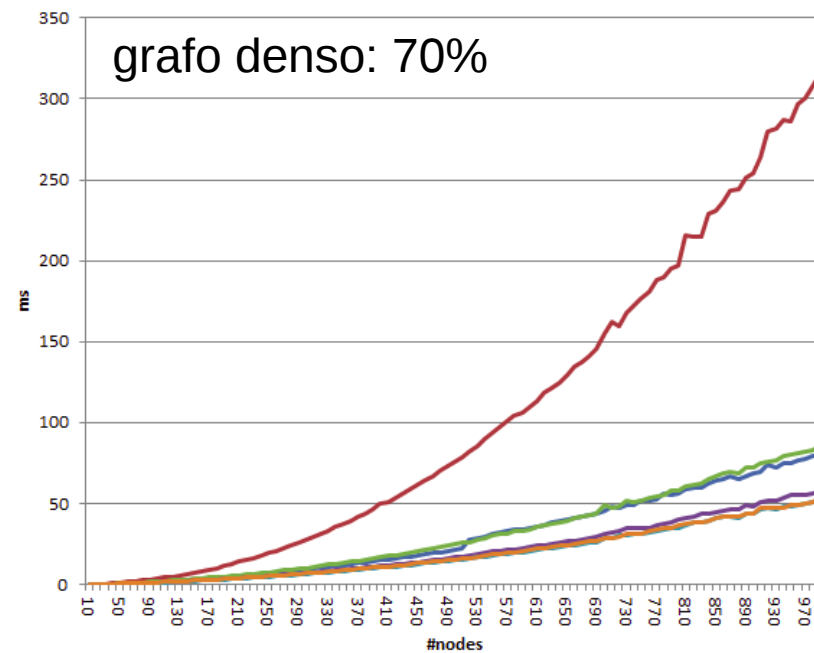
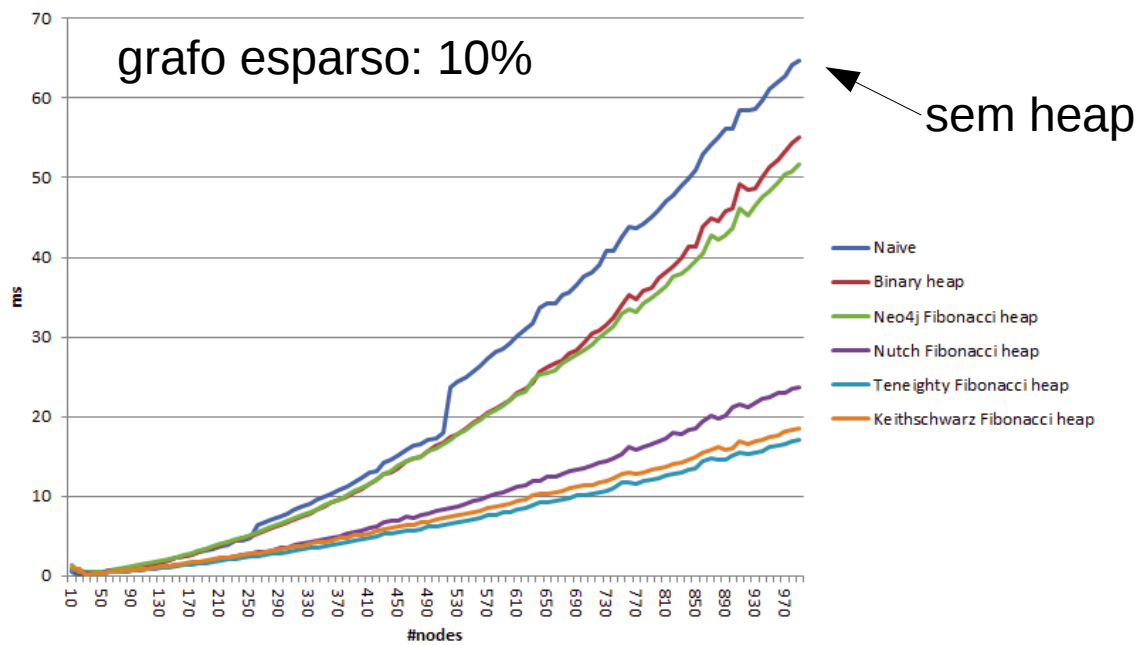


# Complexidade

- Total de operações no heap
  - $n$  operações de inserção e remoção
  - $m$  operações de atualização, no pior caso considerando que cada aresta gera uma atualização
- Cada operação tem custo  $\log n$
- Complexidade:  $O((n+m)\log n)$ 
  - que é igual a  $O(m \log n)$ , se grafo for conexo
  - se  $m \sim n^2$  temos  $O(n^2 \log n)$ : pior que sem o heap!
- Vantagem para grafos esparsos,  $m \ll n^2$

# Tempo de Execução

## ■ Diferentes implementações do heap



## ■ Diferentes implementações do heap

■ “Naive” (sem heap) é o pior no grafo esparso

■ “Binary heap” é pior que “Naive” no grafo denso

■ mas outros heaps são melhores que “Naive”