

# Árvores

## Programação de computadores II

Prof. Renan Augusto Starke

Instituto Federal de Santa Catarina – IFSC  
Campus Florianópolis  
`renan.starke@ifsc.edu.br`

5 de dezembro de 2016



**INSTITUTO FEDERAL**  
**SANTA CATARINA**

Ministério da Educação  
Secretaria de Educação Profissional e Tecnológica  
**INSTITUTO FEDERAL DE SANTA CATARINA**

# Tópicos da aula

- 1 Introdução
- 2 Definições básicas
- 3 Árvores N-árias
- 4 Árvores binárias
- 5 Percorrendo árvores
- 6 Heap
- 7 Exercícios
- 8 Algoritmo de Huffman

- 1 Introdução
- 2 Definições básicas
- 3 Árvores N-árias
- 4 Árvores binárias
- 5 Percorrendo árvores
- 6 Heap
- 7 Exercícios
- 8 Algoritmo de Huffman

- ▶ Estender o conceito e aplicações de grafos para:
  - Árvores
  - Árvores binárias
  - Heaps
- ▶ Entender a ordenação *Heap-sort*
- ▶ Utilizar estruturas de dados conhecidos para representação e manipulação de grafos

- 1 Introdução
- 2 Definições básicas**
- 3 Árvores N-árias
- 4 Árvores binárias
- 5 Percorrendo árvores
- 6 Heap
- 7 Exercícios
- 8 Algoritmo de Huffman

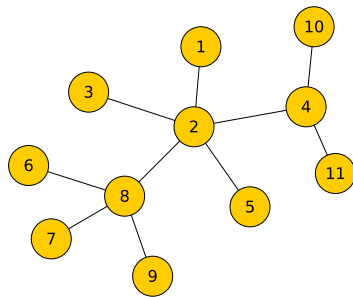
## Árvore

Um grafo  $G = (V, E)$  é uma árvore se  $G$  for **acíclico e não direcionado**.

Propriedades, sendo  $G = (V, E)$  não orientado:

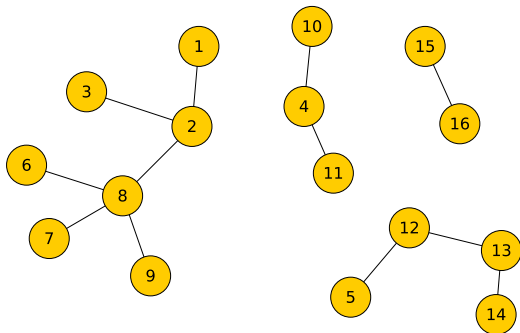
- ▶  $G$  é uma árvore livre.
- ▶ Dois vértices quaisquer em  $G$  estão conectados por um caminho simples único.
- ▶  $G$  é conectado mas, se qualquer aresta for removida de  $E$ , o grafo resultante será desconectado.
- ▶  $G$  é conectado, e  $|E| = |V| - 1$
- ▶  $G$  é acíclico, e  $|E| = |V| - 1$
- ▶  $G$  é acíclico, mas se qualquer aresta for adicionada a  $E$ , haverá um ciclo.

# Exemplos



Árvore

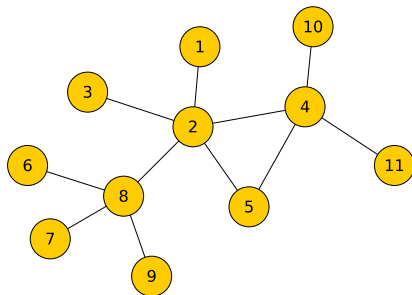
# Exemplos



Floresta



# Exemplos



Grafo comum

## Árvore

Uma árvore  $T$  é um conjunto finito de vértices:

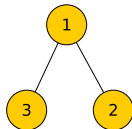
$$T = \{r\} \cup T_1 \cup T_2 \cup \dots \cup T_n$$

Onde:

- 1 Um vértice específico do conjunto,  $r$ , é a raiz da árvore.
- 2 Demais nós são particionados em  $n \geq 0$  subconjuntos,  $T_1, T_2, \dots, T_n$ , onde cada um é uma árvore.

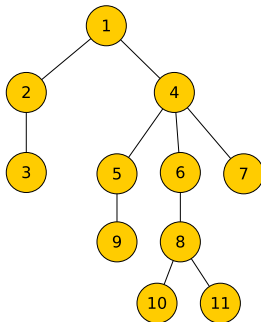


$$T_a = \{1\}$$



$$T_b = \{1, \{2\}, \{3\}\}$$

# Exemplos



$$T_c = \{1, \{2, \{3\}\}, \\ \{4, \{5, \{9\}\}, \\ \{6, \{8, \{10\}, \{11\}\}, \\ \{7\}\}$$

# Terminologia:

Considere uma árvore  $T = \{r, T_1, T_2, \dots, T_n\}$ ,  $n \geq 0$ :

- ▶ O **grau** de um vértice é o número de sub-árvores associados a este vértice. O grau de um árvore  $T$  é  $n$ .
- ▶ Um vértice de grau zero não contém nenhuma sub-árvore. Este vértice é chamado de **folha**.
- ▶ Cada raiz  $r_i$  de uma sub-árvore  $T_i$  é chamada de **filha** de  $r$ .
- ▶ A raiz de um vértice  $r$  de uma árvore  $T$  é o pai de todas as raízes  $r_i$  de todas as sub-árvores  $T_i$ ,  $1 \leq i \leq n$ .
- ▶ Duas raízes  $r_i$  e  $r_j$  de duas árvores distintas  $T_i$  e  $T_j$  de uma árvore  $T$  são chamadas de **irmãs**.

## Caminho

Dada uma árvore  $T$  contendo o conjunto de vértices  $V$ , um caminho em  $T$  é definido com uma sequência não vazia de nós:

$$P = \{v_1, v_2, \dots, v_k\}$$

Onde:

- 1  $v_i \in V$ , para  $1 \leq i \leq k$  de tal forma que o  $n$ -ésimo vértice na sequência,  $v_i$  é o pai do  $(i + 1)$ -ésimo vértice na sequência  $v_{i+1}$ .
- 2 O comprimento do caminho  $P$  é  $k - 1$ .

- 1 Introdução
- 2 Definições básicas
- 3 Árvores N-árias**
- 4 Árvores binárias
- 5 Percorrendo árvores
- 6 Heap
- 7 Exercícios
- 8 Algoritmo de Huffman



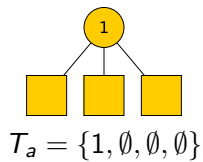
No caso de árvores N-árias, cada vértice (sub-árvore) deve ter exatamente o mesmo grau (número de filhos).

## Árvore N-ária

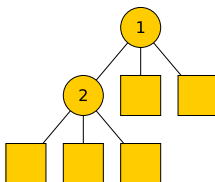
Uma árvore N-ária é um conjunto de vértices com as seguintes propriedades:

- 1  $T$  por der vazia:  $T = \emptyset$
- 2 O conjunto consiste em uma raiz,  $r$ , com exatamente  $N$  sub-árvores distintas.
  - Os vértices remanescentes são particionados em  $N \geq 0$  subconjuntos,  $T_0, T_1, \dots, T_{N-1}$ , cada um com uma sub-árvore de grau  $N$ .

# Exemplo - árvore ternária (*N-ary tree*)

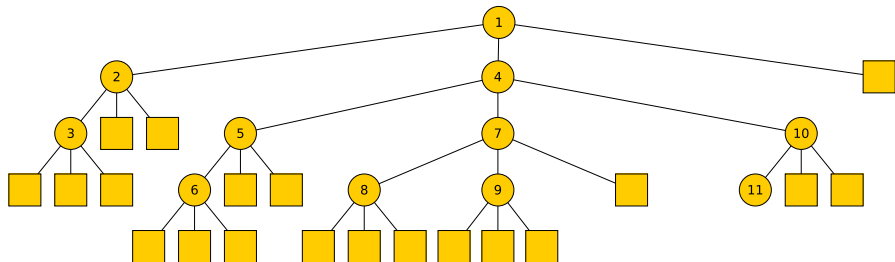


# Exemplo - árvore ternária (*N-ary tree*)



$$T_b = \{1, \{2, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}$$

# Exemplo - árvore ternária (*N-ary tree*)



$$T_c = \{1, \{2, \{3, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \\ \{4, \{5, \{6, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}, \\ \{7, \{8, \emptyset, \emptyset, \emptyset\}, \{9, \emptyset, \emptyset, \emptyset\}, \emptyset\} \\ \{10, \{11, \emptyset, \emptyset, \emptyset\}, \emptyset, \emptyset\}\}, \emptyset\}$$

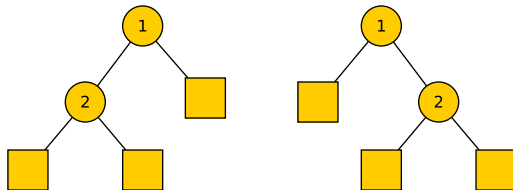
- 1 Introdução
- 2 Definições básicas
- 3 Árvores N-árias
- 4 Árvores binárias**
- 5 Percorrendo árvores
- 6 Heap
- 7 Exercícios
- 8 Algoritmo de Huffman

# Árvores binárias

## Árvore binária

Uma árvore binária é uma árvore  $N$ -ária com grau 2 ( $N = 2$ ) com um conjunto de vértices com as seguintes propriedades:

- 1  $T$  pode ser vazia:  $T = \emptyset$
- 2 O conjunto consiste em uma raiz,  $r$ , com exatamente 2 sub-árvores distintas:  $T_D$  e  $T_E$ 
  - $T = \{r, T_D, T_E\}$



**Cuidado: estas árvores são diferentes.**

- 1 Introdução
- 2 Definições básicas
- 3 Árvores N-árias
- 4 Árvores binárias
- 5 Percorrendo árvores**
- 6 Heap
- 7 Exercícios
- 8 Algoritmo de Huffman

Existem duas maneiras principais para percorrer árvores (já vimos ambos em grafos):

- ▶ Profundidade (*depth-first transversal*)
  - pré-ordem
  - em-ordem
  - pós-ordem
  
- ▶ Largura (*breadth-first transversal*)





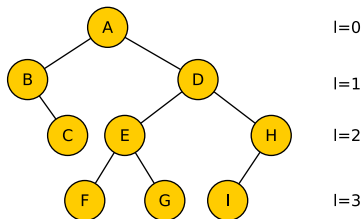
# Profundidade – pré-ordem

```
1 preordem.iterativo(vertices)
2   if (vertices == null)
3       return;
4
5   pilha = cria_pilha();
6   push(vertices, pilha);
7
8   while (!pilha.vazia(pilha))
9       v = pop(pilha);
10
11   visite(v);
12
13   if (vertices.esquerda)
14       push(v.esquerda, pilha)
15   if (vertices.direita)
16       push(v.direita, pilha)
```

# Profundidade – pós-ordem

Algoritmo:

- 1 Percorra a sub-árvore da esquerda em ordem posterior
- 2 Percorra a sub-árvore da direita em ordem posterior
- 3 Visite o vértice



*Saida =*  
*C, B, F, G, E, I, H, D, A*

```
1 posordem.recursivo(vertices)
2   if (vertices == null)
3     return;
4
5   posordem.recursivo(vertices.esquerda);
6   posordem.recursivo(vertices.direita);
7   visite(vertices);
```

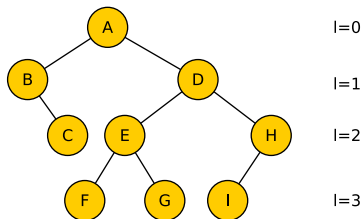
# Profundidade – pós-ordem

```
1 posordem.iterativo(vertice)
2
3 pilha = cria_pilha();
4 ultimoVisitado = null;
5
6 while (!pilha.vazia(pilha) || vertice != null) {
7
8     if (vertice != null)
9         push(vertice, pilha)
10        vertice = vertice.esquerda;
11    else
12        verticeTopo = topo(pilha);
13
14    if (verticeTopo.esquerda != null &&
15        ultimoVisitado != verticeTopo.direita)
16        vertice = verticeTopo.direita;
17    else
18        visite(verticeTopo)
19        ultimoVisitado = pop(pilha);
20 }
```

# Profundidade – em-ordem

Algoritmo:

- 1 Percorra a sub-árvore da esquerda
- 2 Visite o vértice
- 3 Percorra a sub-árvore da direita



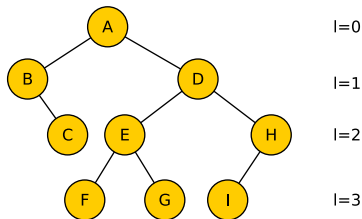
*Saida =*  
*B, C, A, F, E, G, D, I, H*

```
1 emordem_recurativo(vertices)
2   if (vertices == null)
3     return;
4
5   emordem_recurativo(vertices.esquerda);
6   visite(vertices);
7   emordem_recurativo(vertices.direita);
```

# Profundidade – pós-ordem

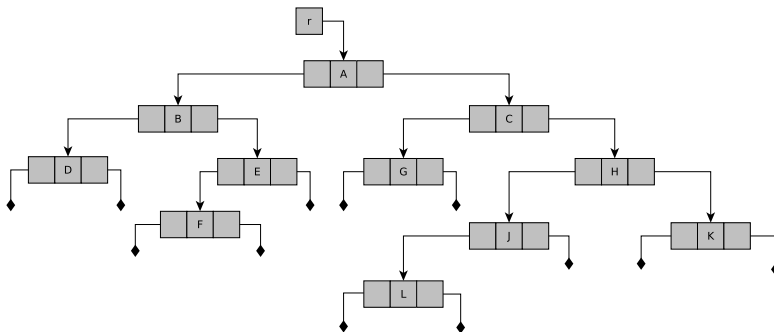
```
1 emordem_iterativo(vertices)
2
3 pilha = cria_pilha();
4
5 while (!pilha.vazia(pilha) || vertice != null)
6
7     if (vertice != null)
8         push(vertice, pilha)
9         vertice = vertice.esquerda;
10    else
11        vertice = pop(pilha);
12        visite(vertice)
13        vertice = vertice.direita;
```

Este algoritmo é basicamente a busca em largura de grafos, visitando os nós (sub-árvores) em camadas.



*Saida =*  
*A, B, D, C, E, H, F, G, I*

# Implementação de árvores binárias – lista encadeada





- 1 Introdução
- 2 Definições básicas
- 3 Árvores N-árias
- 4 Árvores binárias
- 5 Percorrendo árvores
- 6 Heap**
- 7 Exercícios
- 8 Algoritmo de Huffman

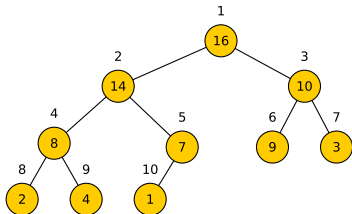
# Heap

## Heap

O heap é uma estrutura de dados que pode ser visto como uma árvore binária praticamente completa. Existem dois tipos de heaps:

- ▶ máximo: para todo vértice  $i$  diferente da raiz:  
 $\text{pai}(i) \geq i$
- ▶ mínimo: para todo vértice  $i$  diferente da raiz:  
 $\text{pai}(i) \leq i$

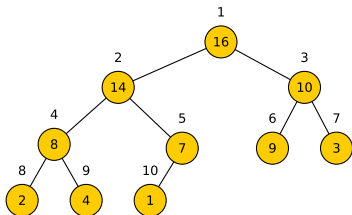
Heap máximo (árvore)



Vetor



# Propriedades um heap



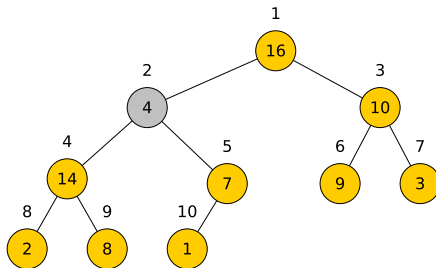
|    |    |    |   |   |   |   |   |   |    |
|----|----|----|---|---|---|---|---|---|----|
| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1  |
| 1  | 2  | 3  | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

```
1 pai(i)
2   return floor(i/2);
3   //se vetor inicia em indice 0
4   return (i-1)/2
5
6 esquerda(i)
7   return 2*i;
8   //se vetor inicia em indice 0
9   return 2*i + 1)
10
11 direita(i)
12   return 2*i + 1;
13   //se vetor inicia em indice 0
14   return (2*i + 2)
```

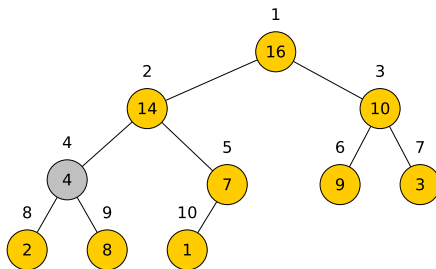
# Construção de um heap

```
1  max_heapify(heap_t *heap, int i) {
2      e = esquerda(i);
3      d = direita(i);
4
5      //cuidado com indices e,d e i para vetores que utilizam indice zero
6      //ajutar comparacao de e <= tam_heap(heap) para e < tam_heap(heap)
7      //para vetores que utilizam indice zero
8      if (e <= tam_heap(heap) && (obter_dado(heap,e) > obter_dado(heap, i))
9          maior = e;
10     else
11         maior = i;
12
13     //ajutar comparacao de d <= tam_heap(heap) para d < tam_heap(heap)
14     //para vetores que utilizam indice zero
15     if (d <= tam_heap(heap) && (obter_dado(heap,d) > obter_dado(heap, maior))
16         maior = d;
17
18     //Se houver algum filho da árvore maior que o pai, troca-se
19     if (maior != i){
20         swap(heap, i, maior);
21         max_heapify(heap, maior);
22     }
23 }
```

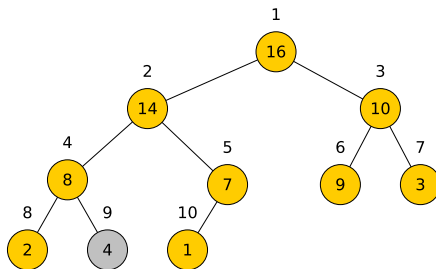
# Exemplo – max\_heapify



# Exemplo – max\_heapify



# Exemplo – max\_heapify



# Construção de um heap

```
1  build_heap(heap_t *heap, int vetor[]) {  
2      heap->tamanho_heap = tamanho_array();  
3      copia_dados(heap, vetor);  
4  
5      //Um heap é construído a partir da metade do array  
6      //Se vetor utiliza o índice 0  
7      //for (i = heap->tamanho/2-1; i >= 0; i--)  
8      for (i = floor(tamanho_array() / 2); i > 0; i--)  
9          max_heapify(heap, i)  
10 }
```

- ▶ Complexidade da construção:  **$O(n)$** .
- ▶ Pode-se construir um heap a partir de um vetor não ordenado em **tempo linear**.

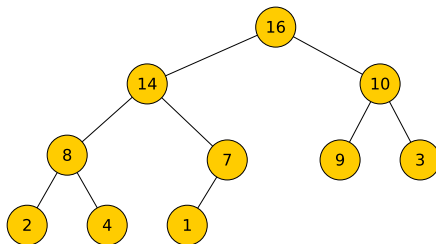


# Ordenação por Heap (heap-sort)

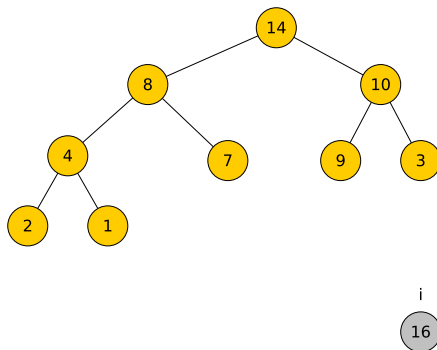
```
1  heapsort(int vetor[]) {  
2      build_heap(heap, vetor);  
3  
4      //para vetores que inciam com indice 1  
5      for (i = tamanho_vetor(); i > 2; i--) {  
6          //para vetores que inciam com indice 1  
7          swap(vetor[1], A[i])  
8  
9          set_tamanho_heap(heap, tamanho_heap(heap) - 1);  
10  
11         //para vetores que inciam com indice 1  
12         max_heapify(heap, 1);  
13     }  
14 }
```

- Complexidade:  $O(n \log n)$ .

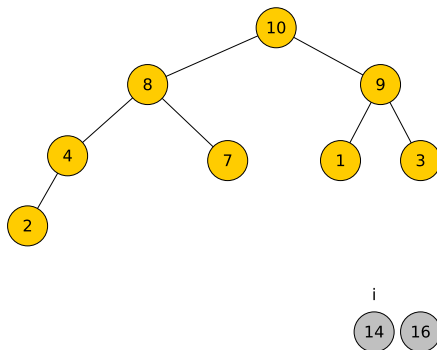
# Exemplo - heapsort



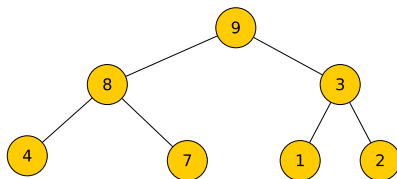
# Exemplo - heapsort



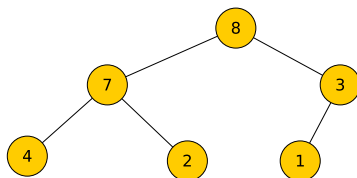
# Exemplo - heapsort



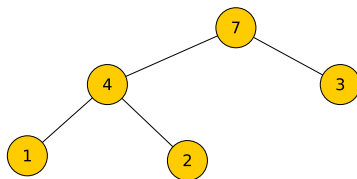
# Exemplo - heapsort



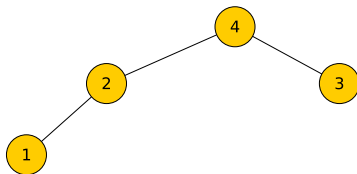
# Exemplo - heapsort



# Exemplo - heapsort

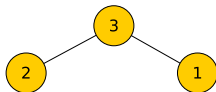


# Exemplo - heapsort

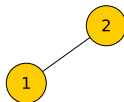




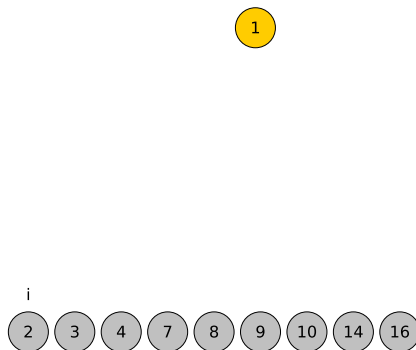
# Exemplo - heapsort



# Exemplo - heapsort



# Exemplo - heapsort



# Exemplo - heapsort



- 1 Introdução
- 2 Definições básicas
- 3 Árvores N-árias
- 4 Árvores binárias
- 5 Percorrendo árvores
- 6 Heap
- 7 Exercícios**
- 8 Algoritmo de Huffman

- ▶ Utilizando a implementação de grafos com listas encadeadas como base:
  - Represente uma árvore binária e exporte o grafo para a linguagem DOT (use a função fornecida)
  - Implemente as 4 funções que percorrem árvores de maneira recursiva e iterativa

- 1 Introdução
- 2 Definições básicas
- 3 Árvores N-árias
- 4 Árvores binárias
- 5 Percorrendo árvores
- 6 Heap
- 7 Exercícios
- 8 Algoritmo de Huffman**

# Algoritmo de Huffman

- ▶ Supõe-se que há uma mensagem de  $n = 4$  símbolos, que são representados pelos seguintes códigos binários:

| Símbolo | Código |
|---------|--------|
| A       | 010    |
| B       | 100    |
| C       | 000    |
| D       | 111    |

- ▶ Uma mensagem ABACCDCA:
  - 010 100 010 000 000 111 010 : total de 21 bits
- ▶ Lembre-se que um caractere é formado por 8-bits



# Algoritmo de Huffman

- ▶ Se mudarmos para 2 bits por símbolo:

| Símbolo | Código |
|---------|--------|
| A       | 00     |
| B       | 01     |
| C       | 10     |
| D       | 11     |

- ▶ Uma mensagem ABACCD A:

- 00 01 00 10 10 11 00 : total de 14 bits

# Algoritmo de Huffman

- ▶ Analisando a mensagem ABACCDAA
  - Letras B e D aparecem somente uma vez
  - Letra A aparece três vezes
  - Letra C aparece duas vezes
- ▶ Construindo uma codificação que atribui códigos menores para símbolos de maior frequência, pode-se reduzir ainda mais:

| Símbolo | Código |
|---------|--------|
| A       | 0      |
| B       | 110    |
| C       | 10     |
| D       | 111    |

- 0 110 0 10 10 111 0: total de 13 bits

# Algoritmo de Huffman

- ▶ Árvore binária de Huffman
- ▶ Folhas:
  - Símbolo e frequência
- ▶ Arestas:
  - Construção do código

