# Árvores

## Programação de computadores II

Prof. Renan Augusto Starke

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

14 de maio de 2018

INSTITUTO FEDERAL
SANTA CATARINA

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

# Tópicos da aula

# Tópico

# Objetivos

- 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

# Tópico

# Definição básica

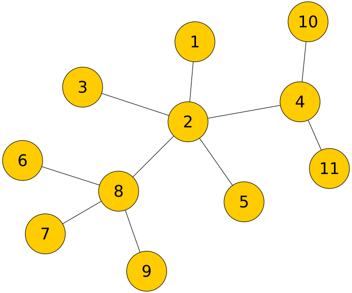
## Árvore

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

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

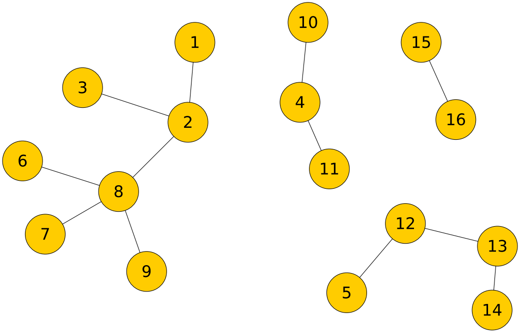
- G é uma árvore livre.
- Dois vértices quaisquer em $G$ estão conectados por um caminho simples único.
- $G$ é conectado mas, se qualquer aresta for removida de $E$, o grafo resultante será desconectado.
- $G$ é conectado, e $|E| = |V| - 1$
- $G$ é acíclico, e $|E| = |V| - 1$
- $G$ é acíclico, mas se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

# Exemplos



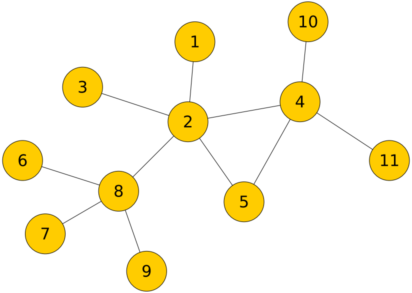

Árvore

# Exemplos




Floresta

# Exemplos




Grafo comum

# Definição matemática

## Árvore

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

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

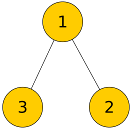
Onde:

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

# Exemplos

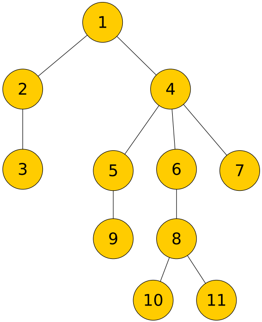

$$T_a = \{1\}$$

# Exemplos




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

# Exemplos




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

# Terminologia:

Considere uma árvore $T = \{r, T_1, T_2, ...T_3\}, n \geq 0$:

- 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 $T_i, 1 < i \leq n$.
- 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

## Caminho

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

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

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 $k - 1$.

# Tópico

# Árvores N-árias

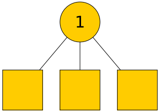
No caso de árvores N-árias, cada vértice (sub-árvore) deve ter exatamente o mesmo grau (número de filhos).

## Árvore N-ária

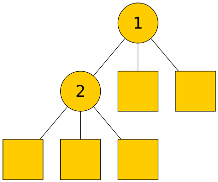
Uma árvore N-ária é um conjunto de vértices com as seguintes propriedades:

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, $T_0, T_1, ..., T_{N-1}$, cada um com uma sub-árvore de grau $N$.

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

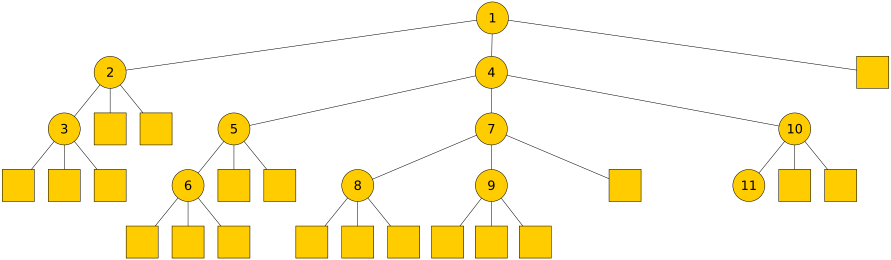



$$T_a = \{1, \emptyset, \emptyset, \emptyset\}$$




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

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




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

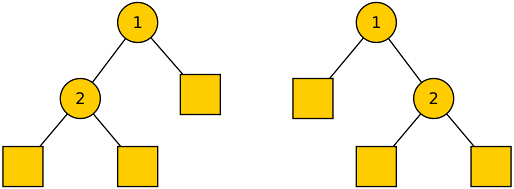
# Tópico

# 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.**

# Tópico

# Percorrendo árvores

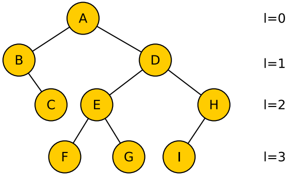
Existem duas maneiras principais para percorres árvores (já vimos ambos em grafos):

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

# Profundidade – pré-ordem

Algoritmo:

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




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

```
preordem_recursivo(vertice)
  if (vertice == null)
    return;

  visite(vertice);
  preordem_recursivo(vertice.esquerda);
  preordem_recursivo(vertice.direita);
```

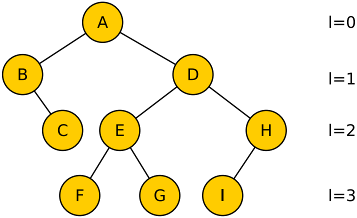
# Profundidade – pré-ordem

```
preordem_iterativo(vertice)
  if (vertice = null)
    return;

  pilha = cria_pilha();
  push(vertice, pilha);

  while (!pilha_vazia(pilha))
    v = pop(pilha);

    visite(v);

    if (vertice.esquerda)
      push(v.direita, pilha)
    if (vertice.direita)
      push(v.esquerda, 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




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

```
posordem_recursivo(vertice)
  if (vertice == null)
    return;

  posordem_recursivo(vertice.esquerda);
  posordem_recursivo(vertice.direita);
  visite(vertice);
```

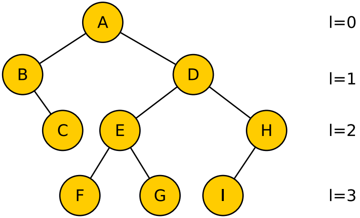
# Profundidade – pós-ordem

```
posordem_iterativo(vertice)

  pilha = cria_pilha();
  ultimoVisitado = null;

  while (!pilha_vazia(pilha) || vertice != null) {

    if (vertice != null)
      push(vertice, pilha)
      vertice = vertice.esquerda;
    else
      verticeTopo = topo(pilha);

    if (verticeTopo.esquerda != null &&
            ultimoVisitado != verticeTopo.direita)
      vertice = verticeTopo.direita;
    else
      visite(verticeTopo)
      ultimoVisitado = pop(pilha);
   }
```

# Profundidade – em-ordem

Algoritmo:

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




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

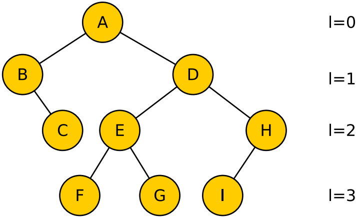
```
emordem_recursivo(vertice)
  if (vertice == null)
    return;

  emordem_recursivo(vertice.esquerda);
  visite(vertice);
  emordem_recursivo(vertice.direita);
```

# Profundidade – pós-ordem

```
emordem_iterativo(vertice)

  pilha = cria_pilha();

  while (!pilha_vazia(pilha) || vertice != null)

    if (vertice != null)
      push(vertice, pilha)
      vertice = vertice.esquerda;
    else
      vertice = pop(pilha);
      visite(vertice)
      vertice = vertice.direita;
```

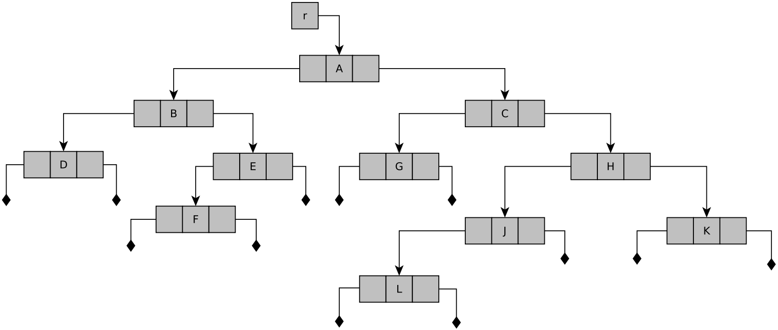
# Largura

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



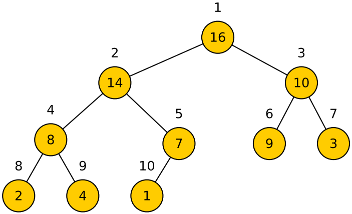
# Tópico

## 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: $dado[pai(i)] \geq dado[i]$
- mínimo: para todo vértice $i$ diferente da raiz: $dado[pai(i)] \leq dado[i]$

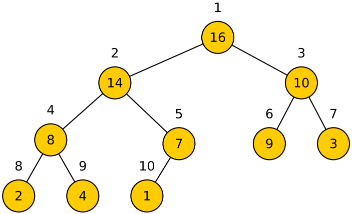
Heap máximo (árvore)




Vetor

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

# Propriedades um heap



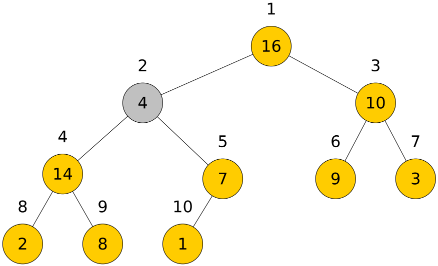

| 16 | 14 | 10 | 8 | 7 | 9 | 3 | 2 | 4 | 1 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

```
pai (i)
  return floor(i/2);
  //se vetor inicia em indice 0
  return (i−1)/2

esquerda(i)
  return 2∗i;
  //se vetor inicia em indice 0
  return 2∗i + 1)

direita(i)
  return 2∗i + 1;
   //se vetor inicia em indice 0
   return (2∗i + 2)
```

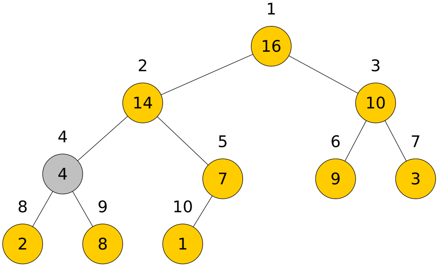
# Construção de um heap

```c
max_heapify(heap_t *heap, int i) {
 e = esquerda(i);
 d = direita(i);

 //cuidado com indices e,d e i para vetores que utilizam indize zero
 //ajustar comparacao de e <= tam_heap(heap) para e < tam_heap(heap)
 //para vetores que utilizam indize zero
 if (e <= tam_heap(heap) && (obter_dado(heap,e) > obter_dado(heap, i))
   maior = e;
 else
   maior = i;

 //ajutar comparacao de d <= tam_heap(heap) para d < tam_heap(heap)
 //para vetores que utilizam indize zero
 if (d <= tam_heap(heap) && (obter_dado(heap,d) > obter_dado(heap, maior))
   maior = d;

 //Se houver algum filho da árvore maior que o pai, troca-se
 if (maior != i){
   swap(heap, i, maior);
   max_heapify(heap, maior);
 }
}
```

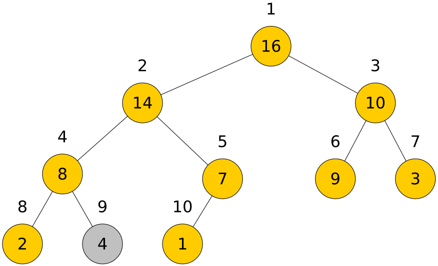
# Exemplo – max_heapify



# Exemplo – max_heapify



# Exemplo – max_heapify



```
build_heap(heap_t *heap, int vetor[]) {
 heap->tamanho_heap = tamanho_array();
 copia_dados(heap, vetor);

 //Um heap é construído a partir da metade do array
 //Se vetor utiliza o indice 0
 //for (i = heap->tamanho/2-1; i >= 0; i--)
 for (i = floor(tamanho_array() / 2); i > 0; i--)
   max_heapify(heap, i)
}
```

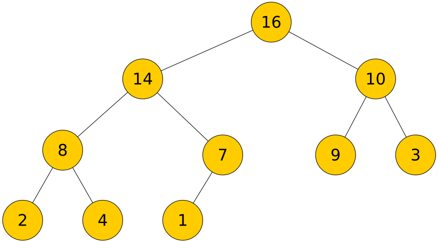
- Complexidade da construção: **O(n)**.
- Pode-se construir um heap a partir de um vetor não ordenado em **tempo linear**.

# Ordenação por Heap (heap-sort)

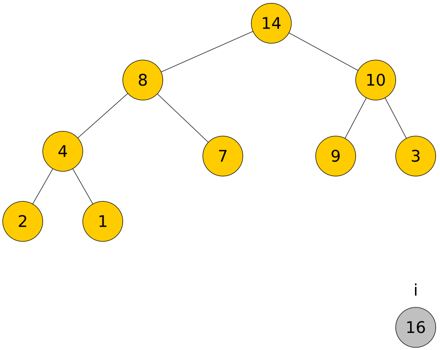
```
heapsort(int vetor[]) {
  build_heap(heap, vetor);

  //para vetores que inciam com indice 1
  for (i = tamanho_vetor(); i >= 2; i--) {
   //para vetores que inciam com indice 1
   swap(A[1], A[i])

   set_tamanho_heap(heap, tamanho_heap(heap) - 1);

   //para vetores que inciam com indice 1
   max_heapify(heap, 1);
  }
 }
```

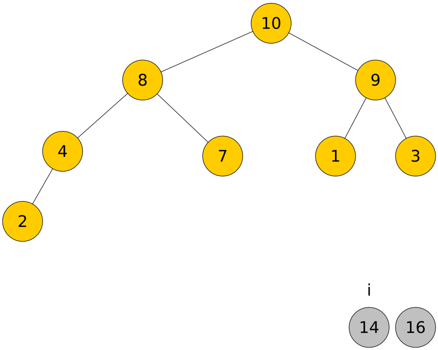
- Complexidade: $O(n \log n)$.

# Exemplo - heapsort

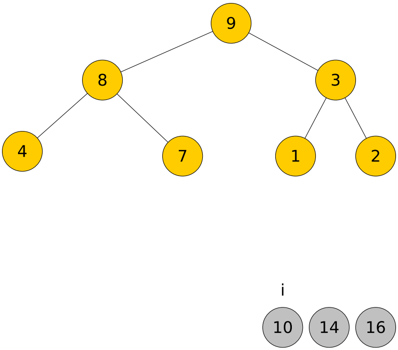


# Exemplo - heapsort

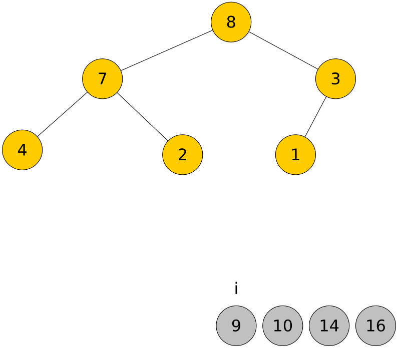



10
8
9
4
7
1
3
2
i
14
16

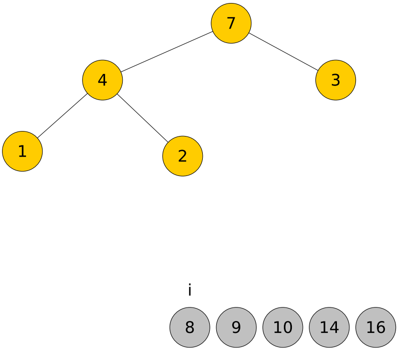
# Exemplo - heapsort



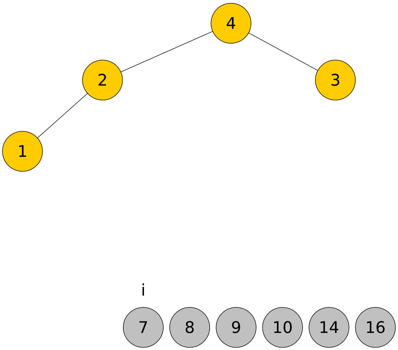
# Exemplo - heapsort



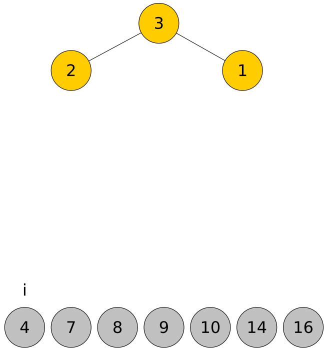
# Exemplo - heapsort



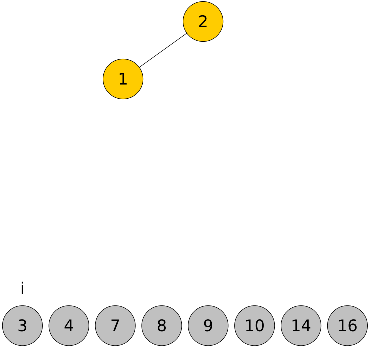
# Exemplo - heapsort



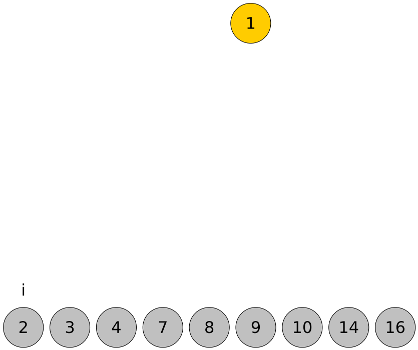
# Exemplo - heapsort



# Exemplo - heapsort



# Exemplo - heapsort



# Exemplo - heapsort

1 2 3 4 7 8 9 10 14 16

# Tópico

# Exercícios

- 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

# Tópico

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

- Analisando a mensagem ABACCDA
  - Letras B e D aparecem somente uma vez
  - Letra A aparece três vezes
  - Letra C apare duas vezes
- Construindo um codificação que atribui códigos menores para símbolos de maior frequência, pose-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

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