INF 1010
Estruturas de Dados Avançadas

Listas de Prioridades e *Heaps*

Listas de Prioridades

Em muitas aplicações, dados de uma coleção são acessados por ordem de prioridade

A prioridade associada a um dado pode ser qualquer coisa: tempo, custo, etc. Só precisa ser ordenável

Nesse contexto, as operações que devem ser eficientes são:

- Seleção do elemento com maior (ou menor) prioridade
- Remoção do elemento de maior (ou menor) prioridade
- Inserção de um novo elemento

Complexidade

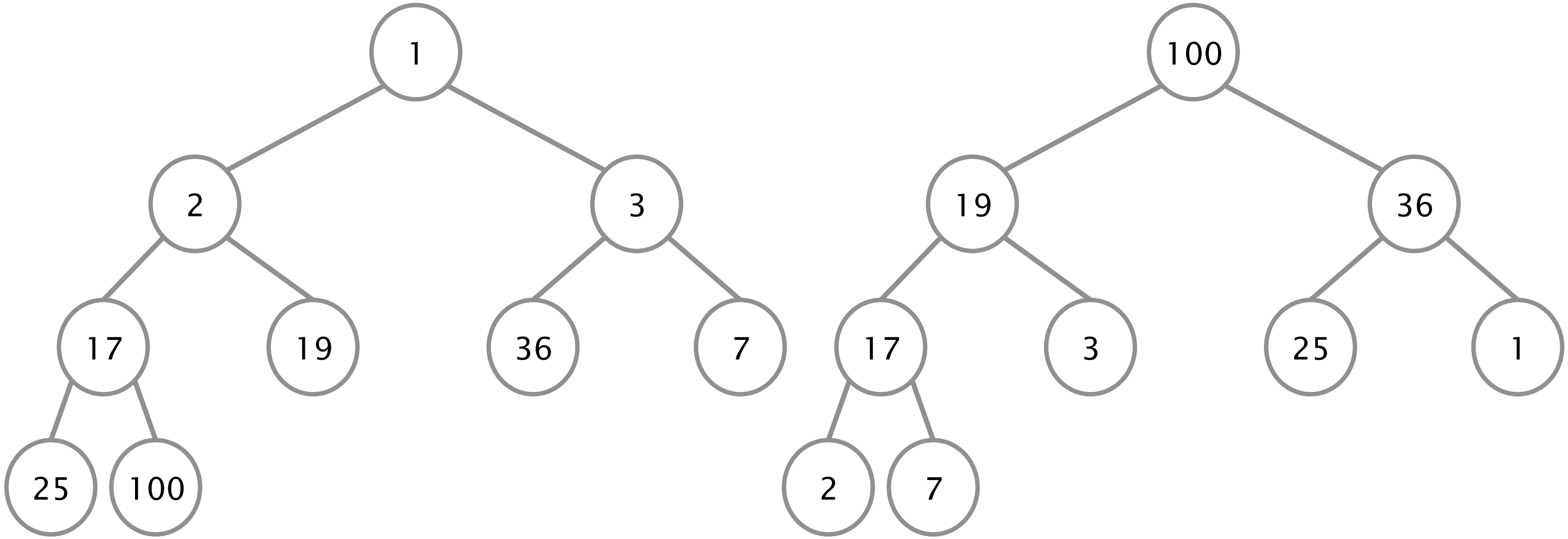
Operação	Lista	Lista Ordenada	Árvore (Balanceada)	Heap
Seleção	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
Inserção	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$
Remoção	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Alteração (de prioridade)	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$
Construção	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

O que é um heap (binário)

Árvore binária completa ou quase completa

Min heap: Cada nó é **menor** que seus filhos

Max heap: Cada nó é **maior** que seus filhos



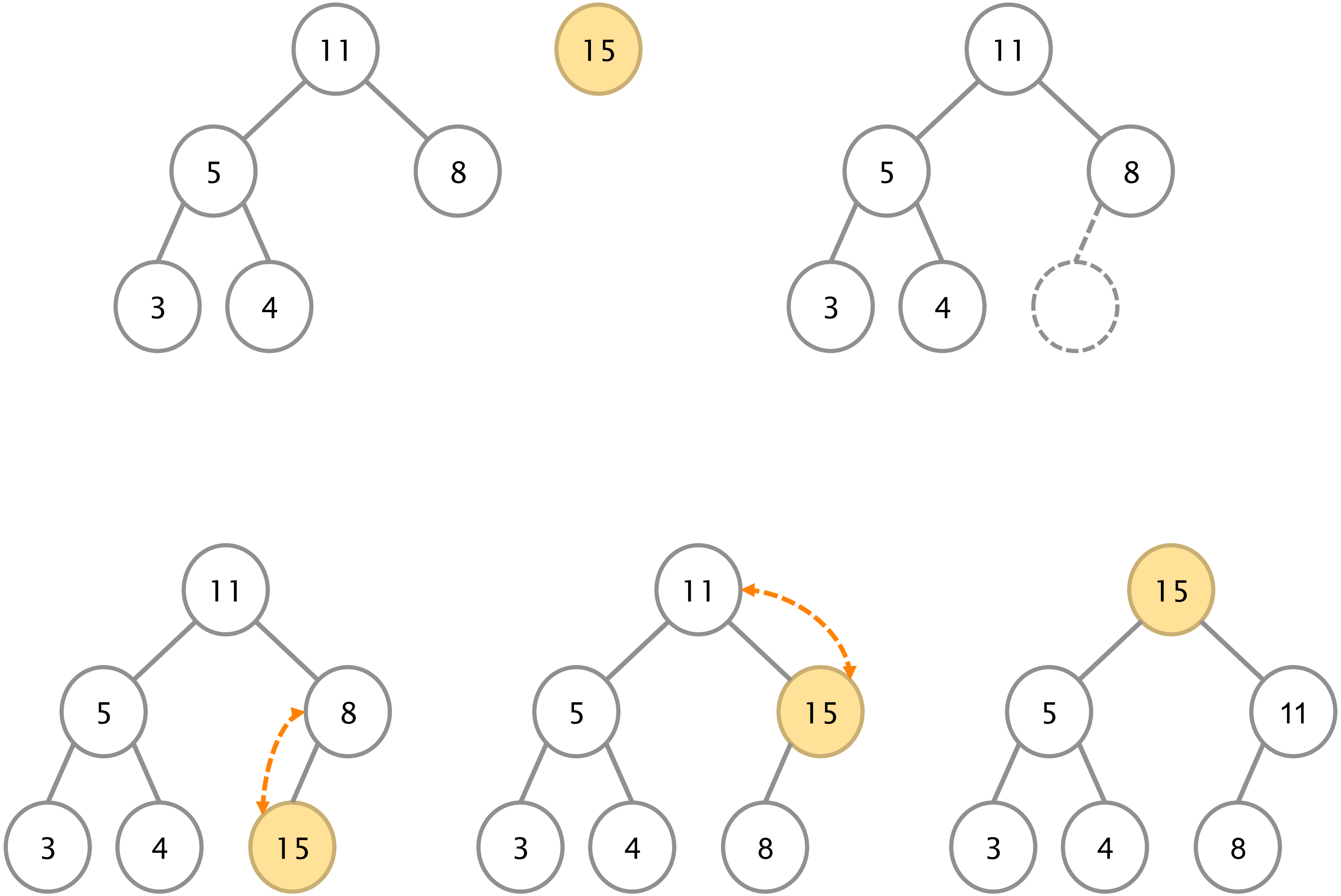
Observações

- A estrutura de dados Heap não deve ser confundida com o Heap normalmente utilizado para designar a memória de alocação dinâmica.
- Não existe restrição para o número de filhos de um nó de uma árvore heap, porém o convencional são dois nós filhos.

Inserir um elemento

1. Insira o elemento no final do heap
2. Compare ele com seu pai:
   - Se estiver em ordem, a inserção terminou.
   - Se não estiver, troque com o pai e repita o passo 2 até terminar ou chegar à raiz.

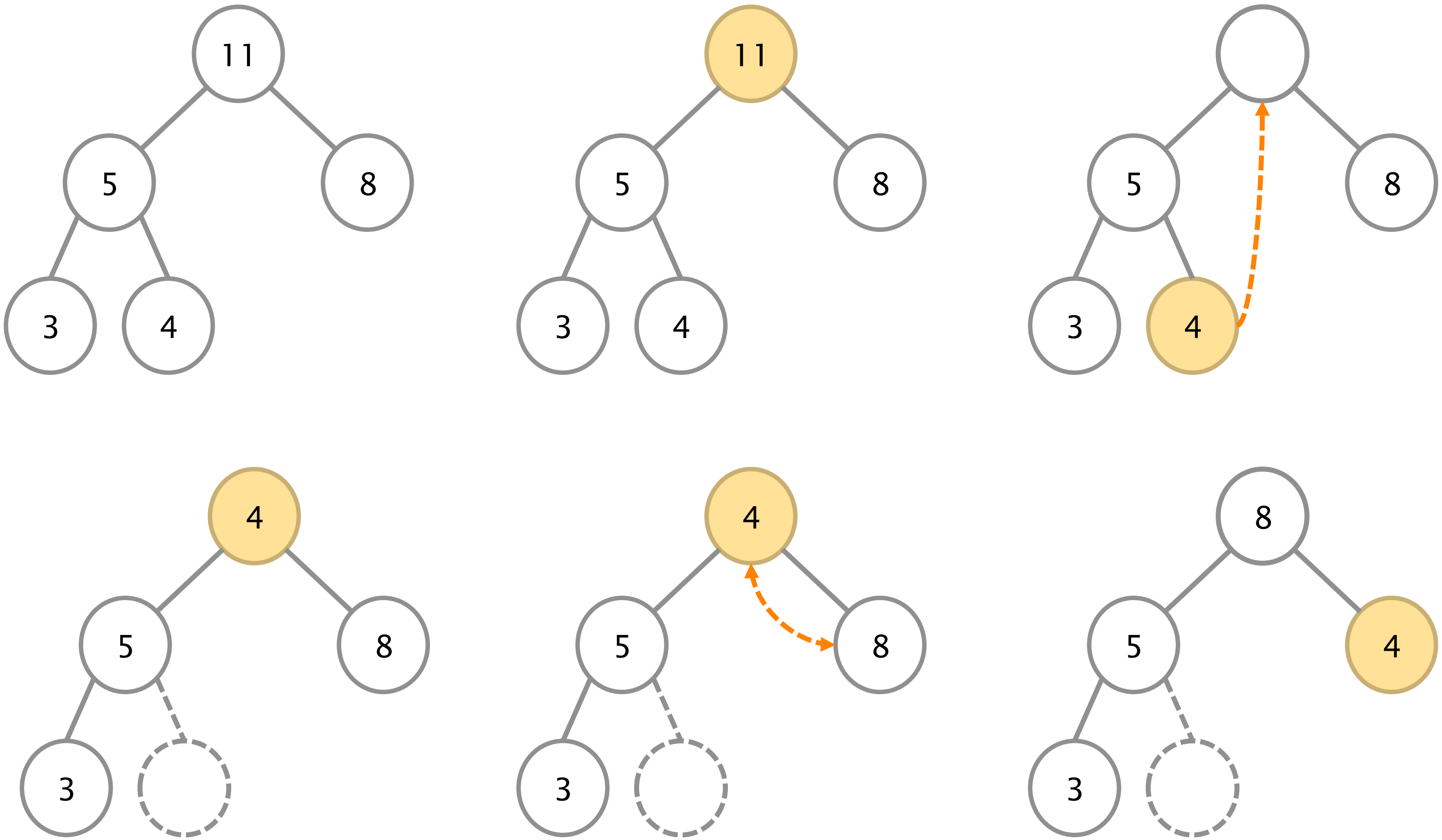
Exemplo de inserção



Remoção (do topo)

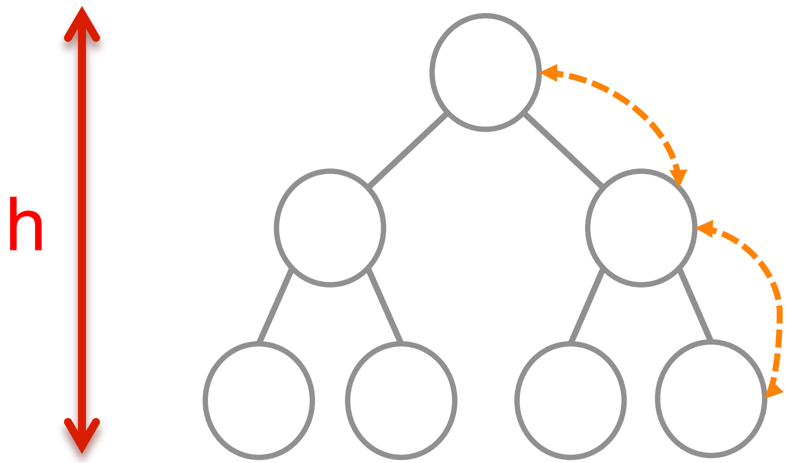
1. Coloque na raiz o último elemento
2. Compare ele com seus filhos:
   - Se estiver em ordem, a remoção terminou.
   - Se não estiver, troque com o maior filho e repita o passo 2 até terminar ou chegar numa folha.

Exemplo de remoção



Complexidade

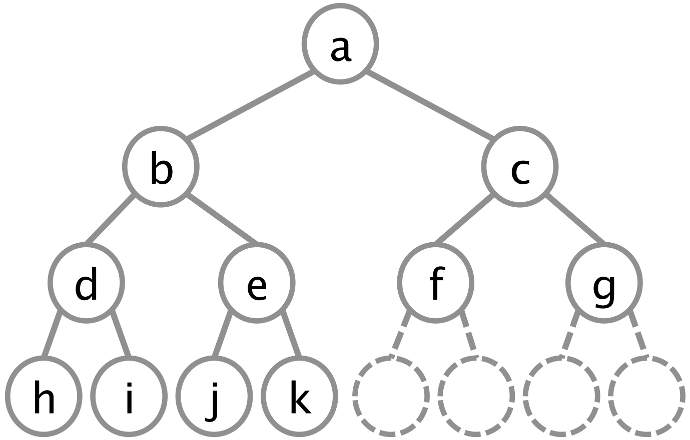
Tanto a inserção como a remoção faz trocas recursivas dos pais com os filhos. No pior caso toda a altura da árvore seria percorrida: O(h), como $h=\lfloor \log_2 n \rfloor$: $O(\log n)$



Implementando heap com vetor

Dado um nó armazenado no índice i, é possível computar o índice:

- do nó filho esquerdo de i : **2*i +1**
- do nó filho direito de i : **2*i + 2**
- do nó pai de i : **(i-1)/2**



índice	0	1	2	3	4	5	6	7	8	9	10	...
nó	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**...**
nível	0	1		2				3				

Para armazenar uma árvore de altura *h* precisamos de um vetor de $2^{(h+1)}-1$ (número de nós de uma árvore cheia de altura *h*).

Os *n* nós da árvore estão nas *n* primeiras posições do vetor.

Implementação de um TAD Heap

```
typedef struct _heap Heap;

Heap* heap_cria(int max);
void  heap_insere(Heap* heap, float prioridade);
float heap_remove(Heap* heap);
void heap_print(Heap *heap);
void heap_print_indent(Heap *heap);
```

Implementação de um TAD Heap

```
struct _heap {
    int max;          /* tamanho maximo do heap */
    int pos;          /* proxima posicao disponivel no vetor */
    float* prioridade; /* vetor das prioridades */
};

Heap* heap_cria(int max){
    Heap* heap=(Heap*)malloc(sizeof(Heap));
    heap->max=max;
    heap->pos=0;
    heap->prioridade=(float*)malloc(max*sizeof(float));
    return heap;
}
```

Insere

```
void heap_insere(Heap* heap, float prioridade)
{
    if ( heap->pos < heap->max )
    {
        heap->prioridade[heap->pos]=prioridade;
        corrige_acima(heap,heap->pos);
        heap->pos++;
    }
    else
        printf("Heap CHEIO!\n");
}
```

Insere

```
static void troca(int a, int b, float* v) {
    float f = v[a];
    v[a] = v[b];
    v[b] = f;
}

static void corrige_acima(Heap* heap, int pos) {
    while (pos > 0){
        int pai = (pos-1)/2;
        if (heap->prioridade[pai] < heap->prioridade[pos])
            troca(pos,pai,heap->prioridade);
        else
            break;
        pos=pai;
    }
}
```

Remove

```
float heap_remove(Heap* heap)
{
  if (heap->pos>0) {
    float topo=heap->prioridade[0];
    heap->prioridade[0]=heap->prioridade[heap->pos-1];
    heap->pos--;
    corrige_abaixo(heap);
    return topo;
  }
  else {
     printf("Heap VAZIO!");
     return -1;
  }
}
```

Remove

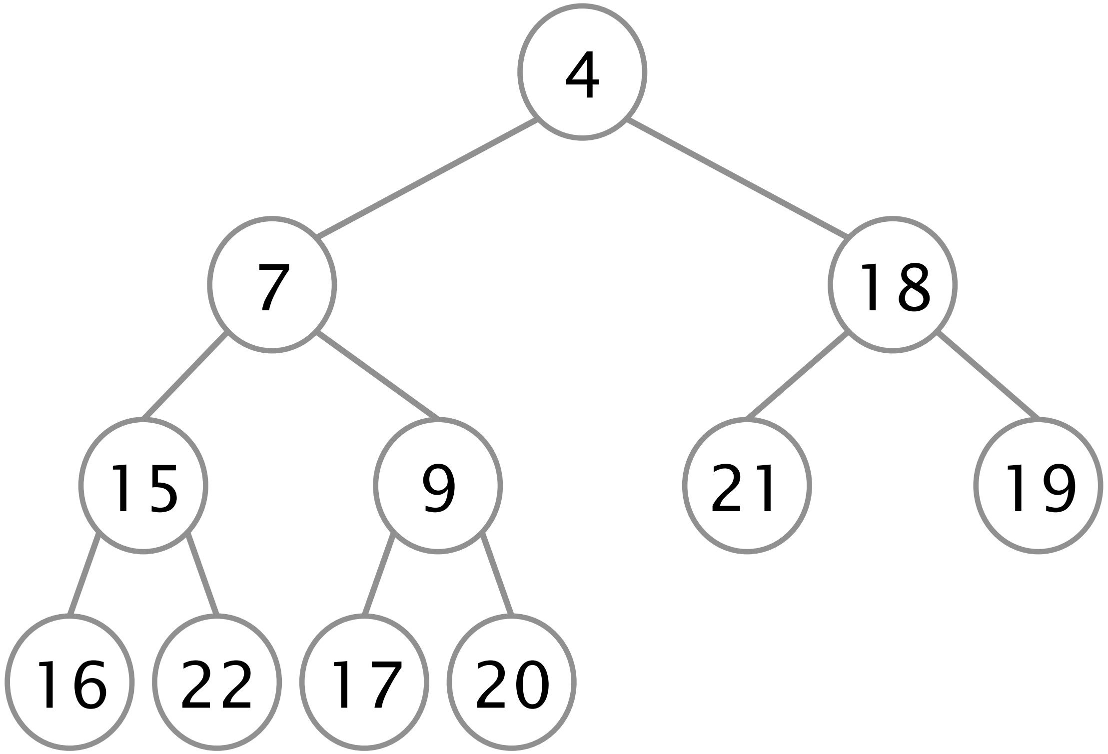
```
static void corrige_abaixo(Heap* heap){
  int pai=0;
  while (2*pai+1 < heap->pos){
    int filho_esq=2*pai+1;
    int filho_dir=2*pai+2;
    int filho;
    if (filho_dir >= heap->pos) filho_dir=filho_esq;
    if (heap->prioridade[filho_esq]>heap->prioridade[filho_dir])
        filho=filho_esq;
    else
        filho=filho_dir;
    if (heap->prioridade[pai]<heap->prioridade[filho])
            troca(pai,filho,heap->prioridade);
    else
            break;
    pai=filho;
  }
}
```

Alterando a Prioridade em um Heap

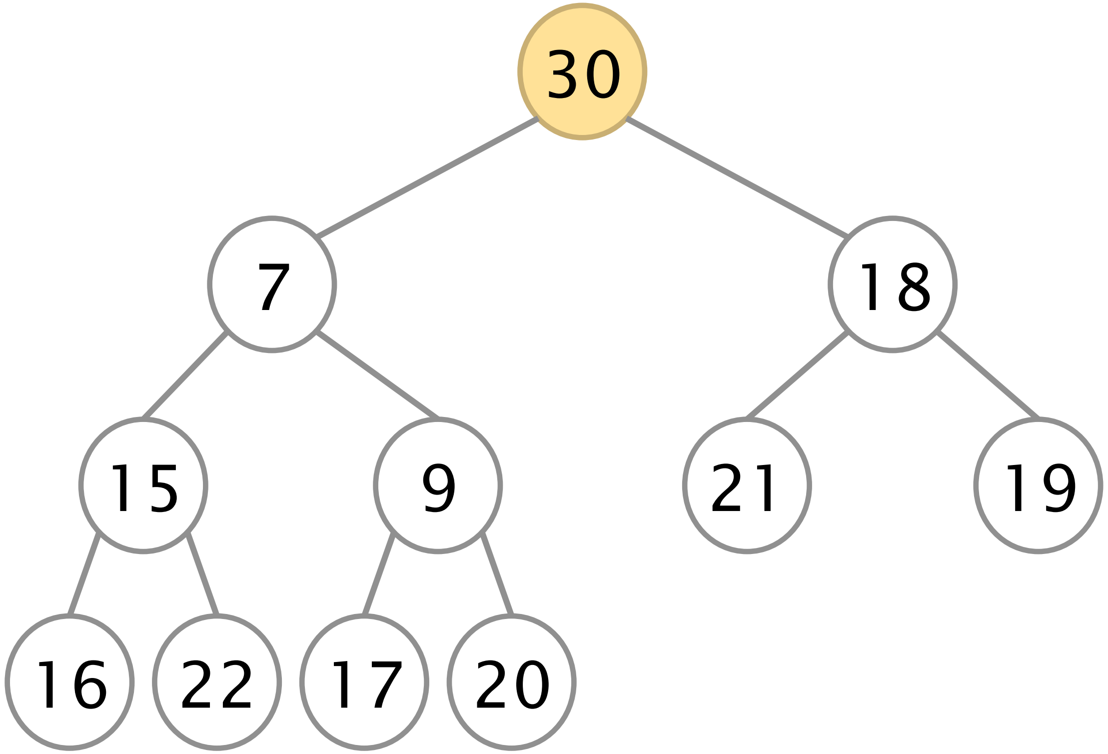
Se um nó tem seu valor alterado, a manutenção das propriedades do heap pode requerer que o nó "migre" na árvore

- para cima (se ele aumentar a prioridade)
- para baixo (se ele reduzir a prioridade)

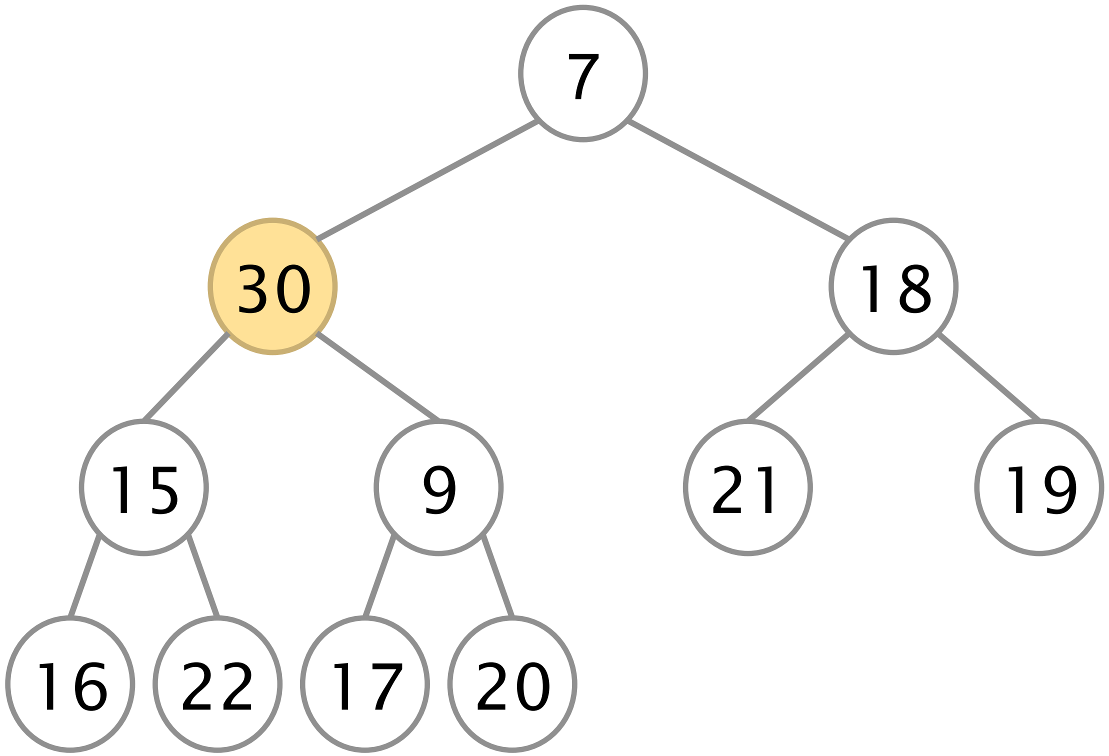
Migração de valores num Min Heap



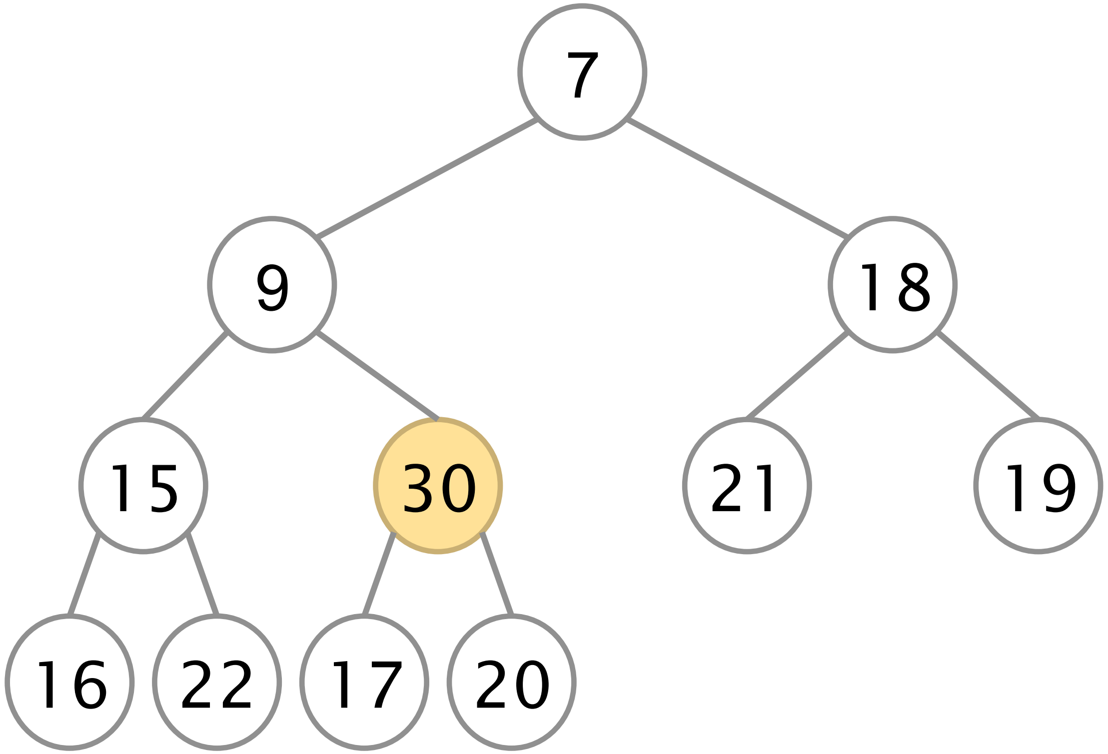
Migração de valores num Min Heap



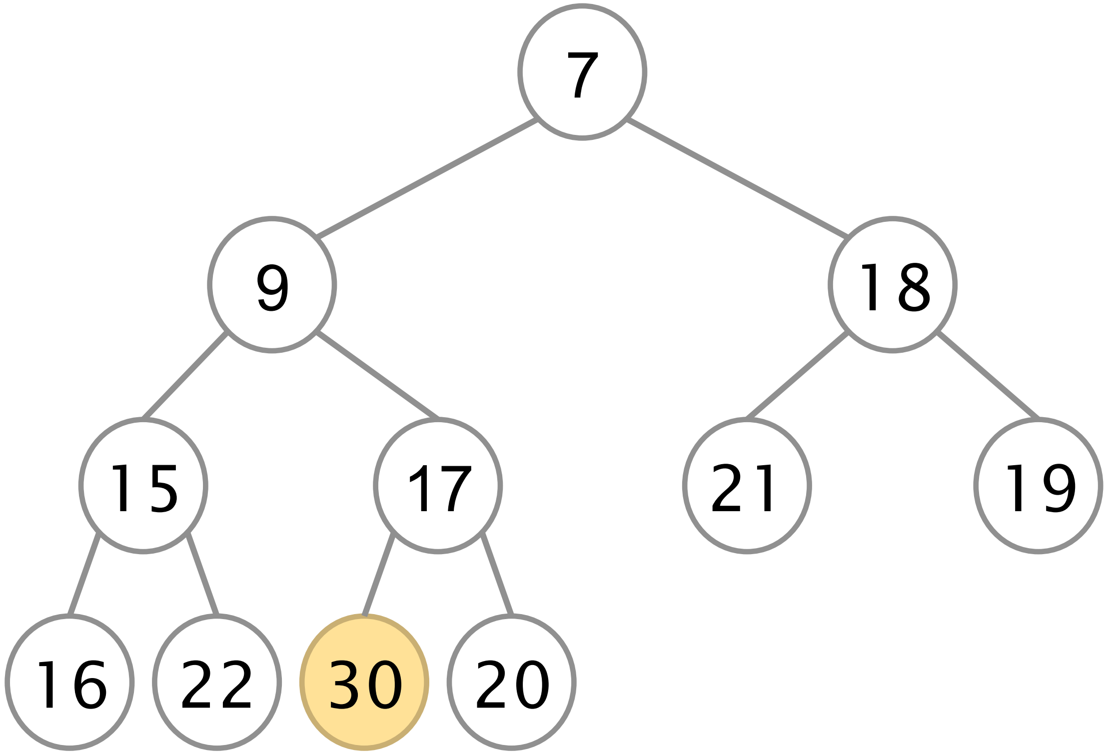
Migração de valores num Min Heap



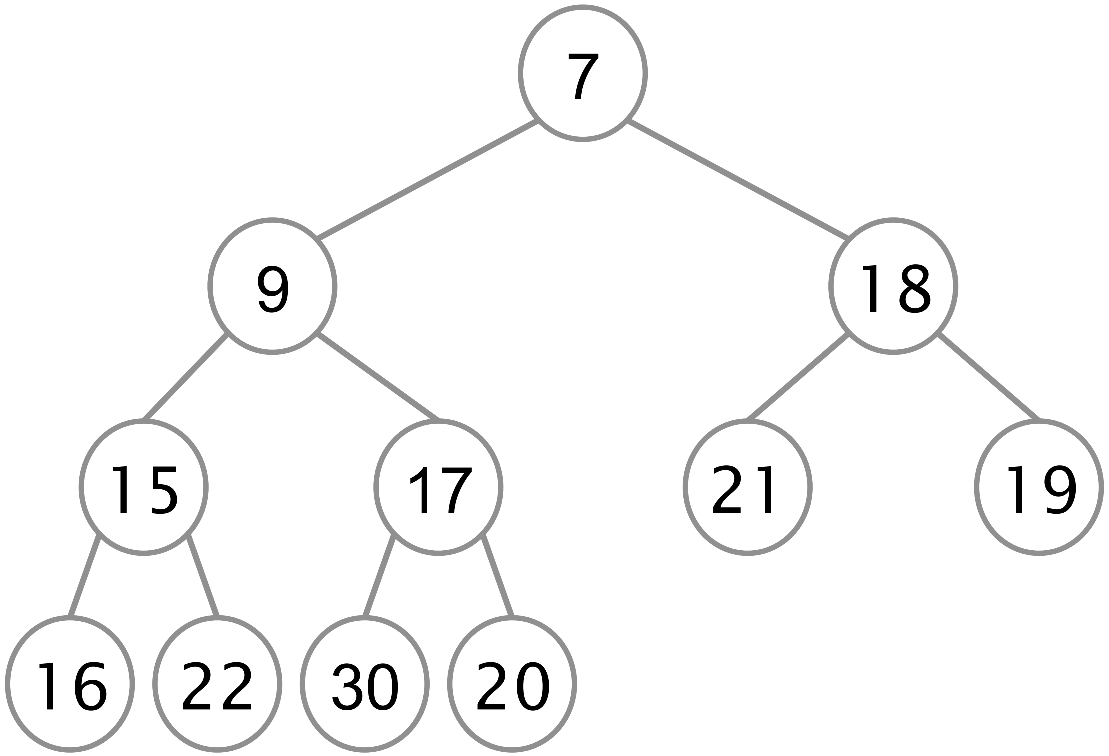
Migração de valores num Min Heap



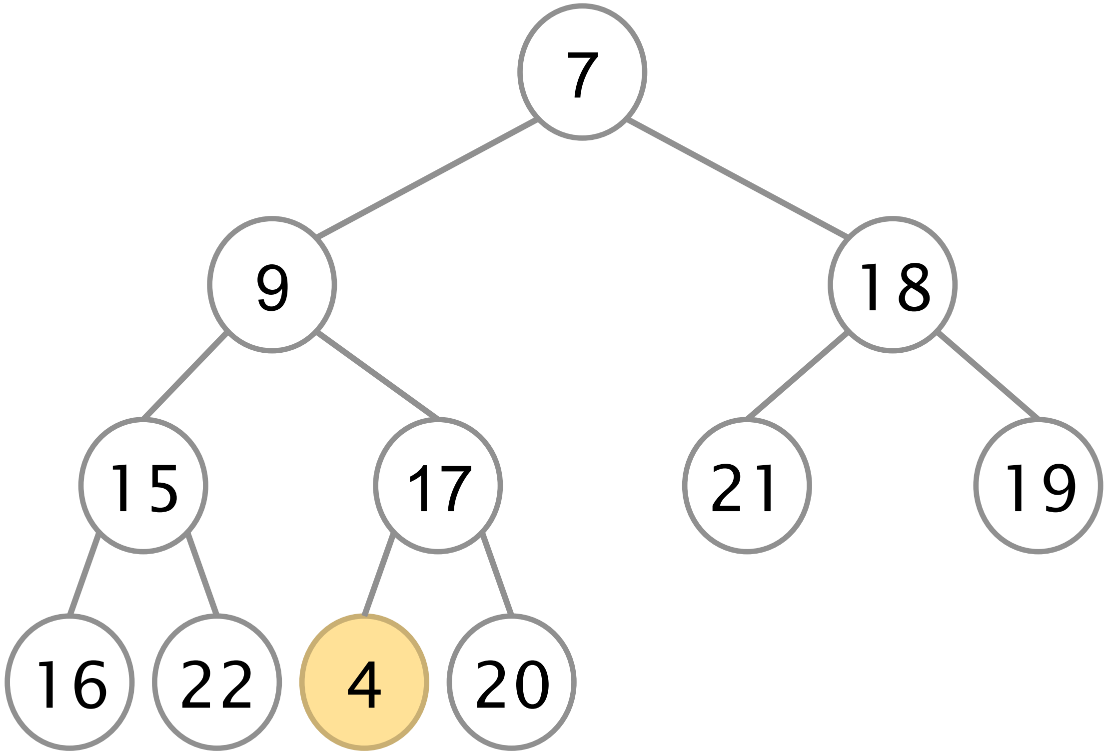
Migração de valores num Min Heap



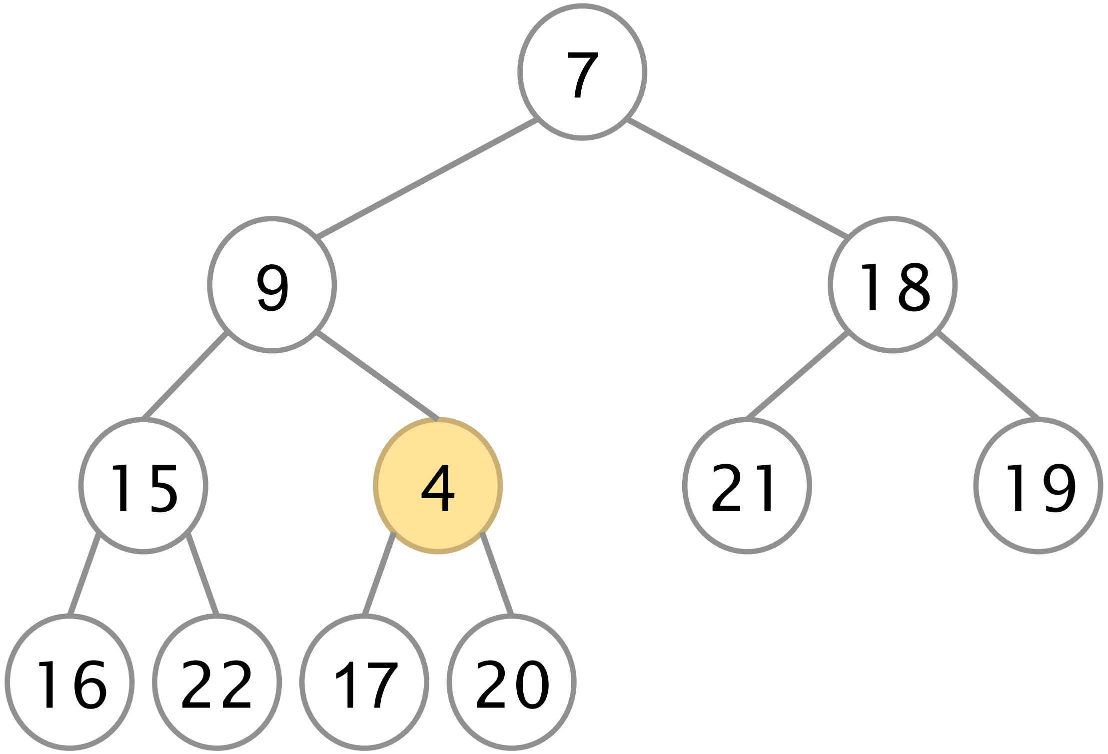
Migração de valores num Min Heap (2)



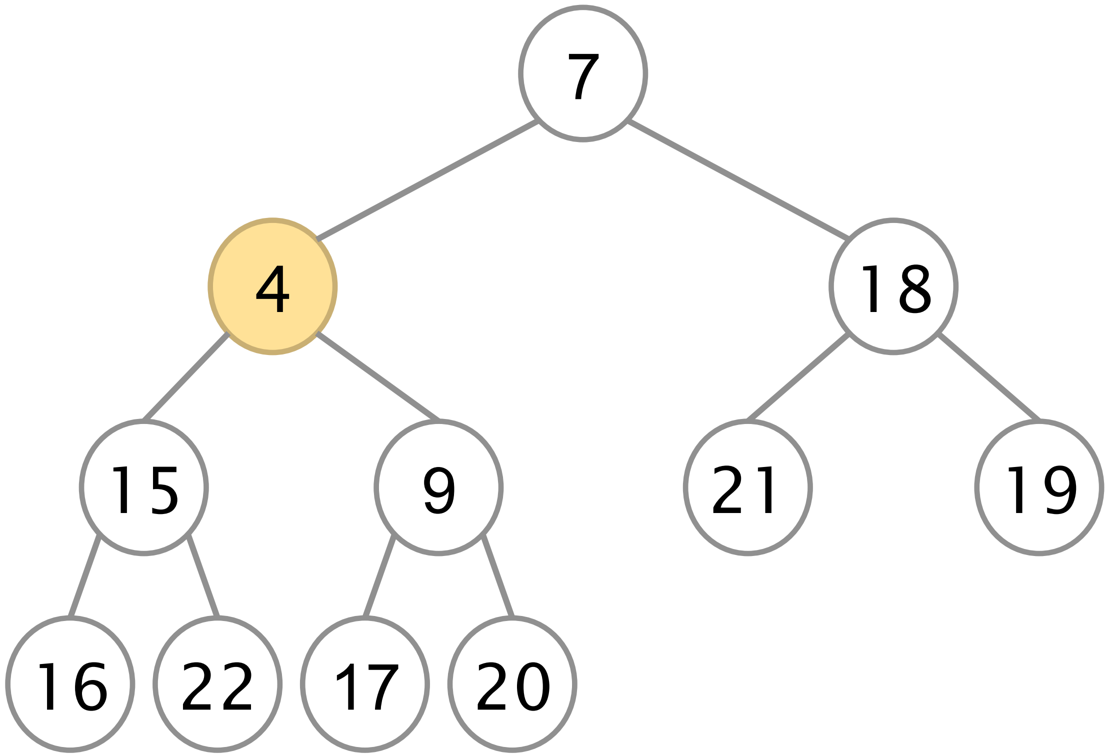
Migração de valores num Min Heap (2)



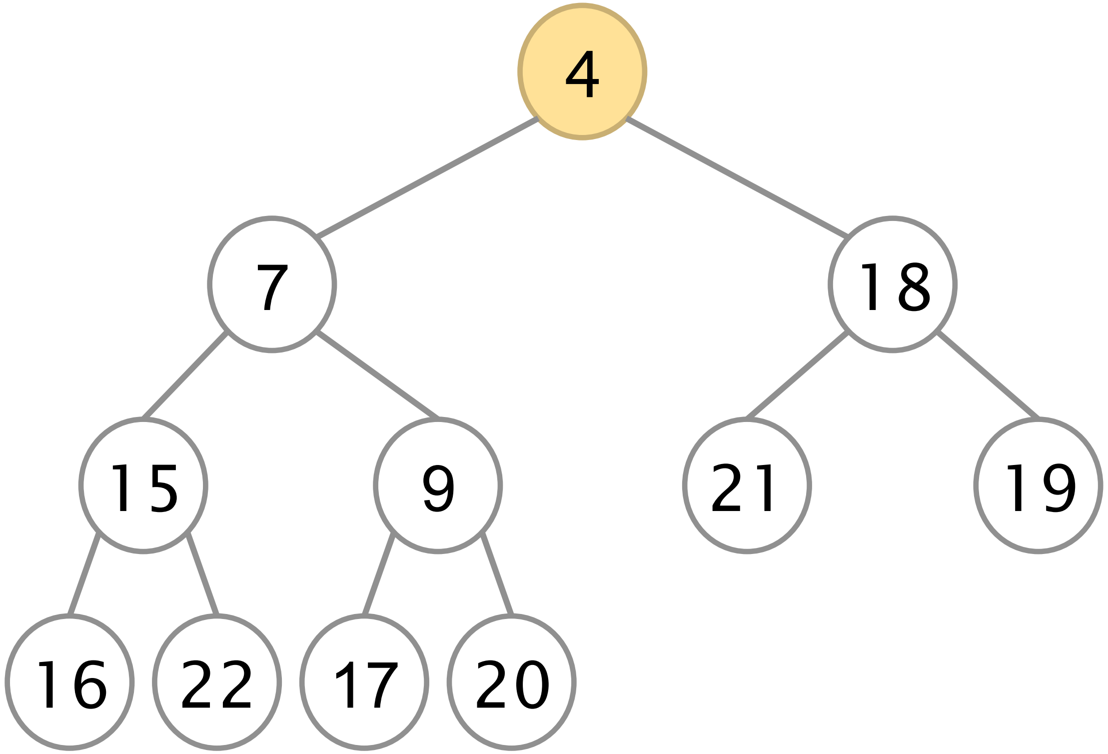
Migração de valores num Min Heap (2)



Migração de valores num Min Heap (2)



Migração de valores num Min Heap (2)



Construção de Heaps

Algoritmo ingênuo:

- Insira um-a-um todos os n elementos.
  - Cada elemento é inserido na base e sobe até seu lugar.

Complexidade:

$$O\left(n \log(n)\right)$$

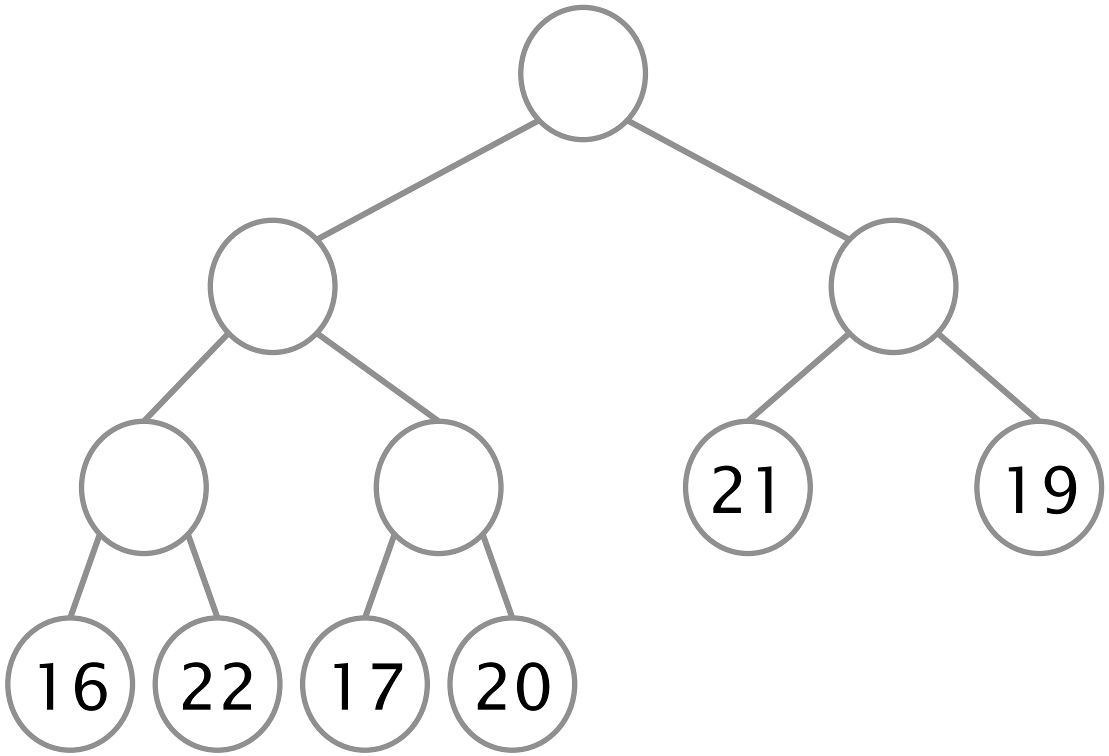
Construção de Heap

Observe que:

- As folhas da árvore (elementos $n/2 + 1$ .. n) não têm descendentes e portanto já estão ordenadas em relação a eles
- Se acertarmos todos os nós internos (elementos 1 .. $n/2$) em relação a seus descendentes, o heap estará pronto
- É preciso trabalhar de trás para frente, desde $n/2$ até 1, pois as propriedades da heap estão corretas apenas nos níveis mais baixos.

Construção de Min Heap (exemplo)

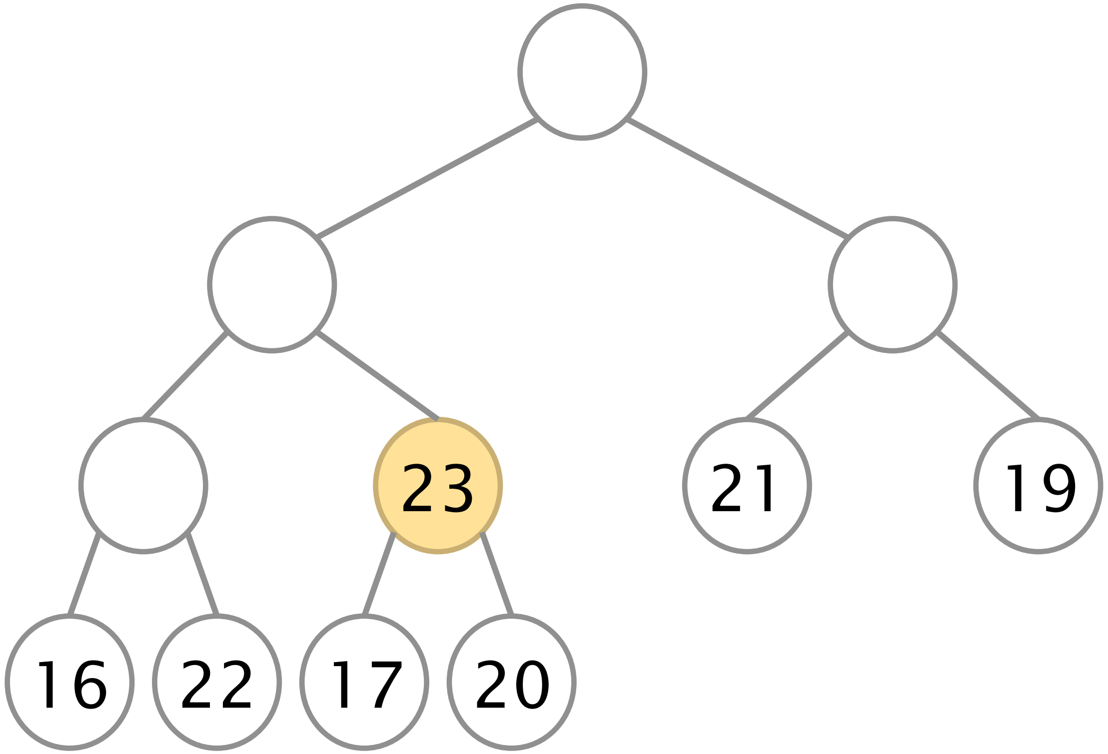
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

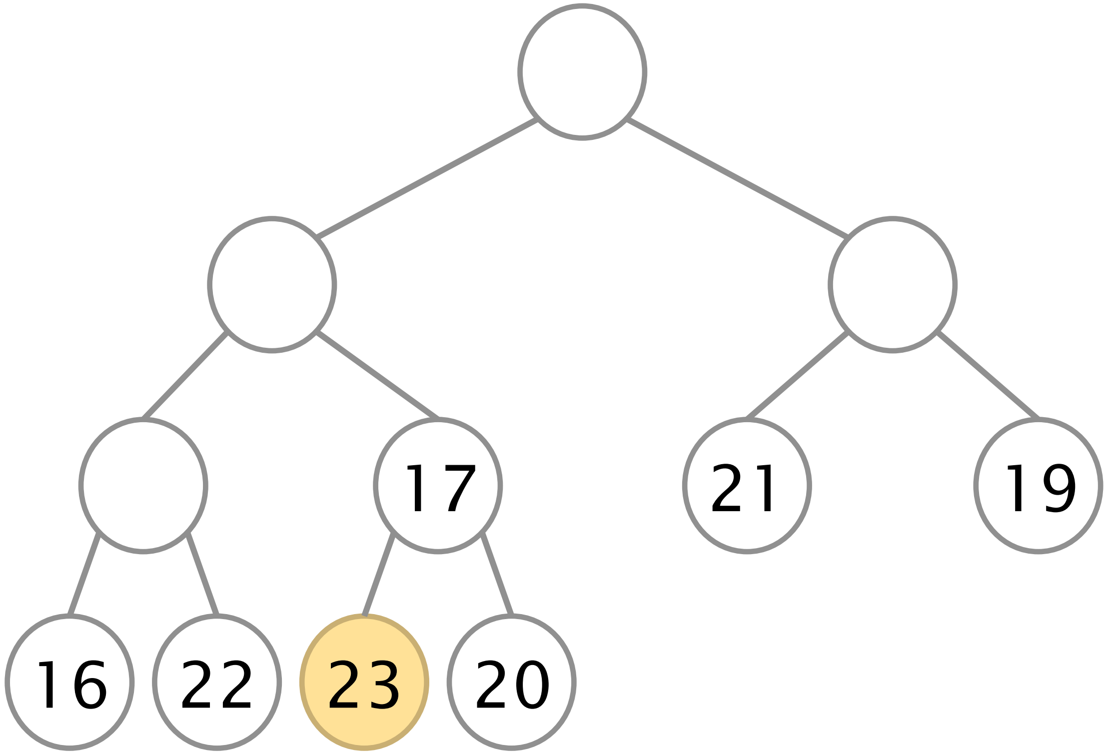
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

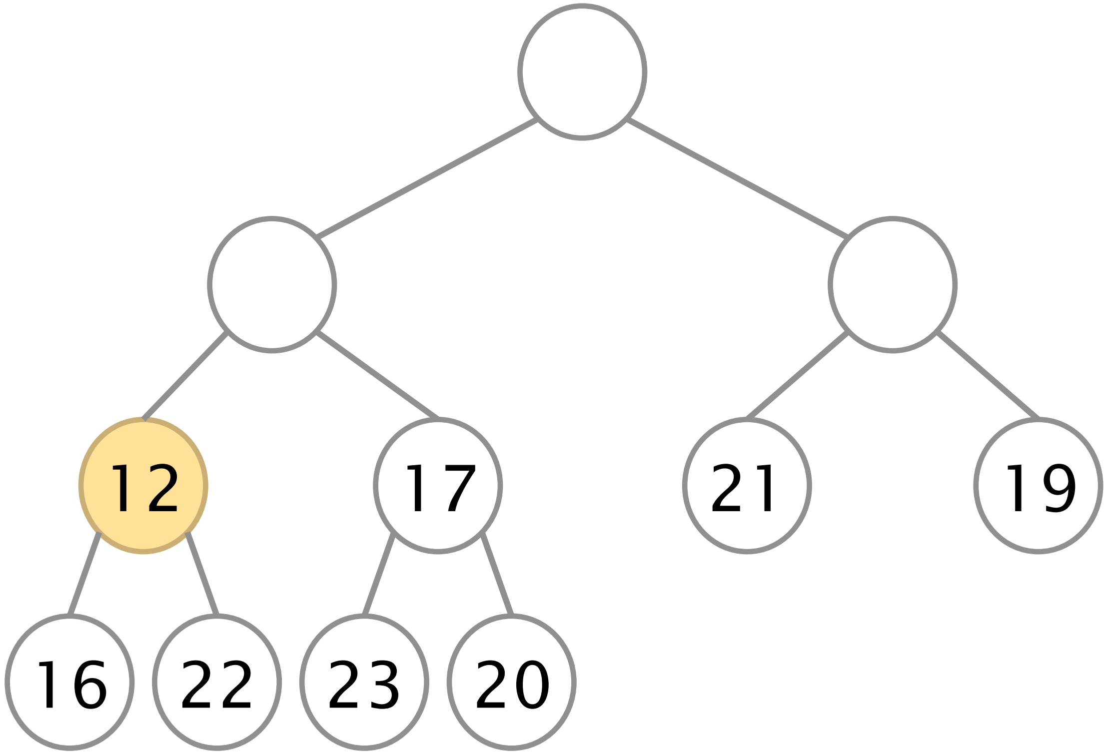
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

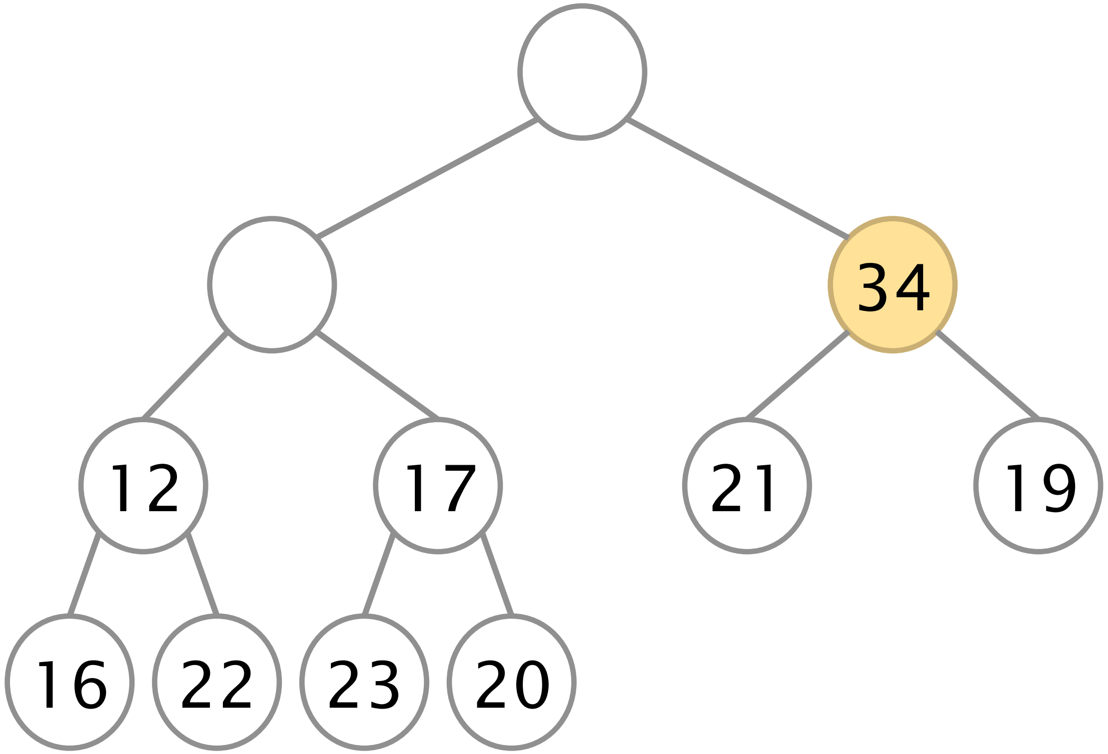
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

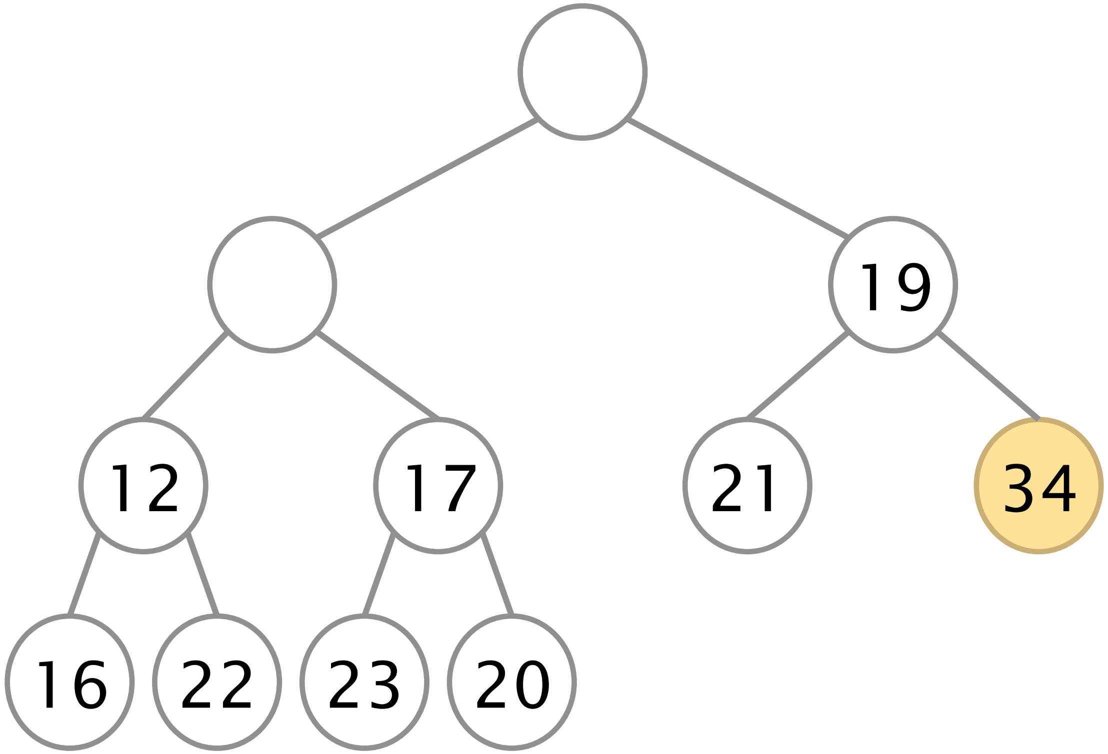
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

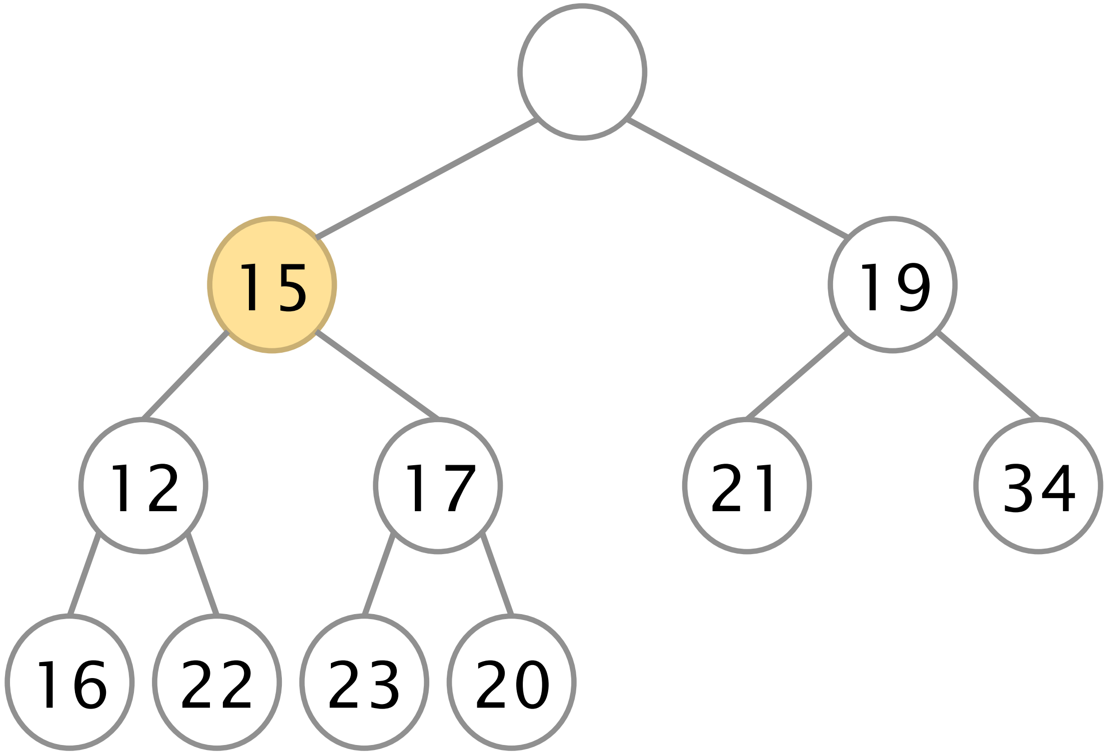
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

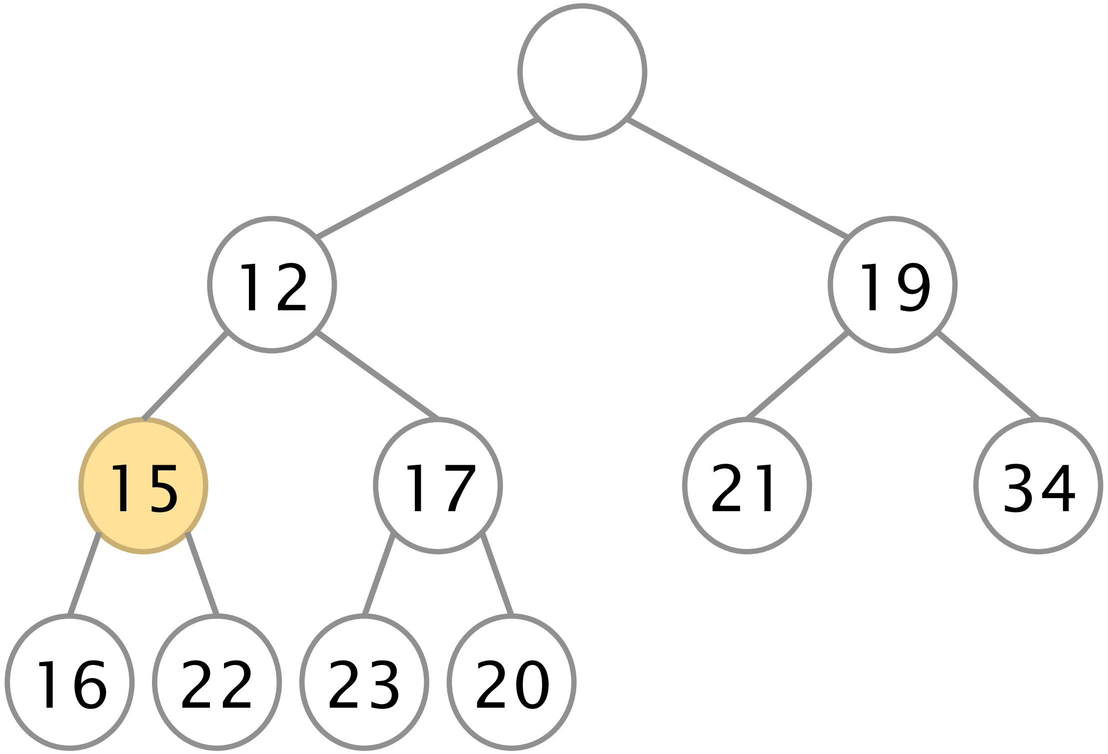
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

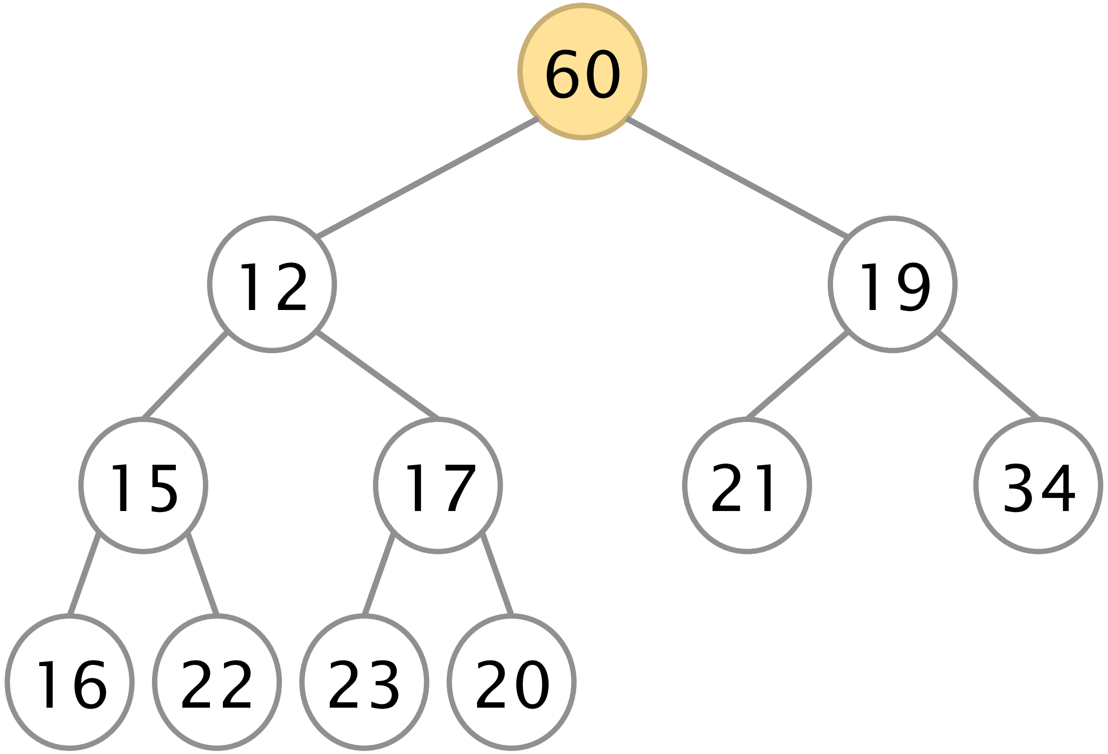
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

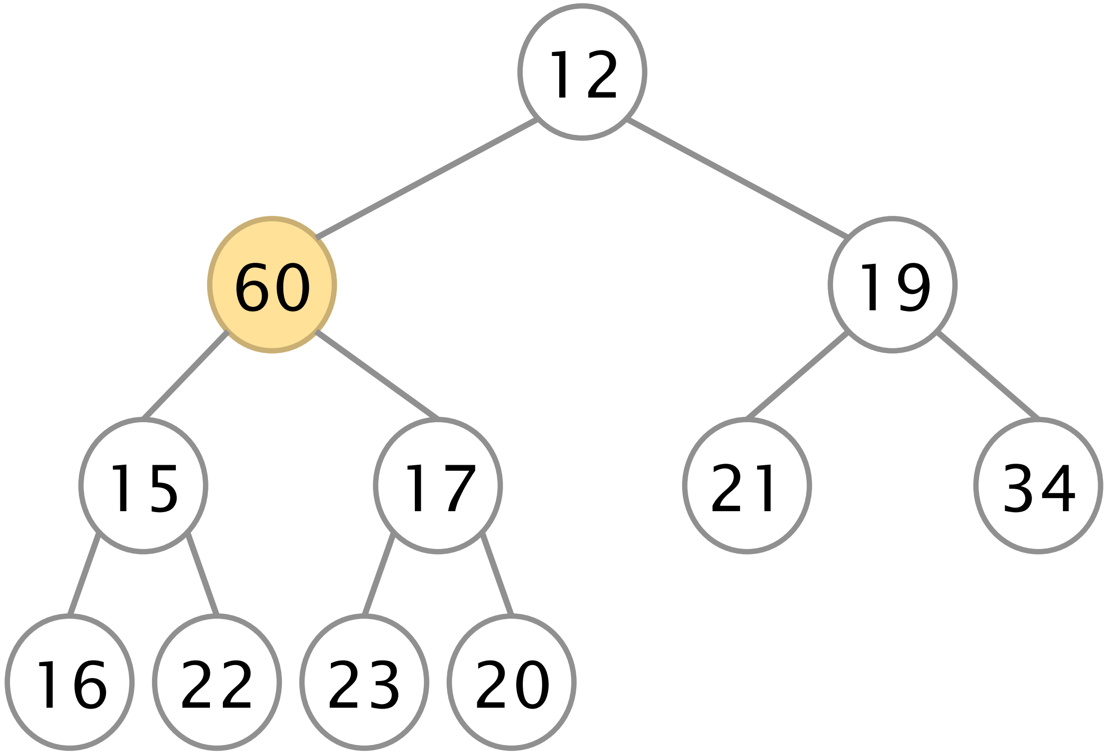
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

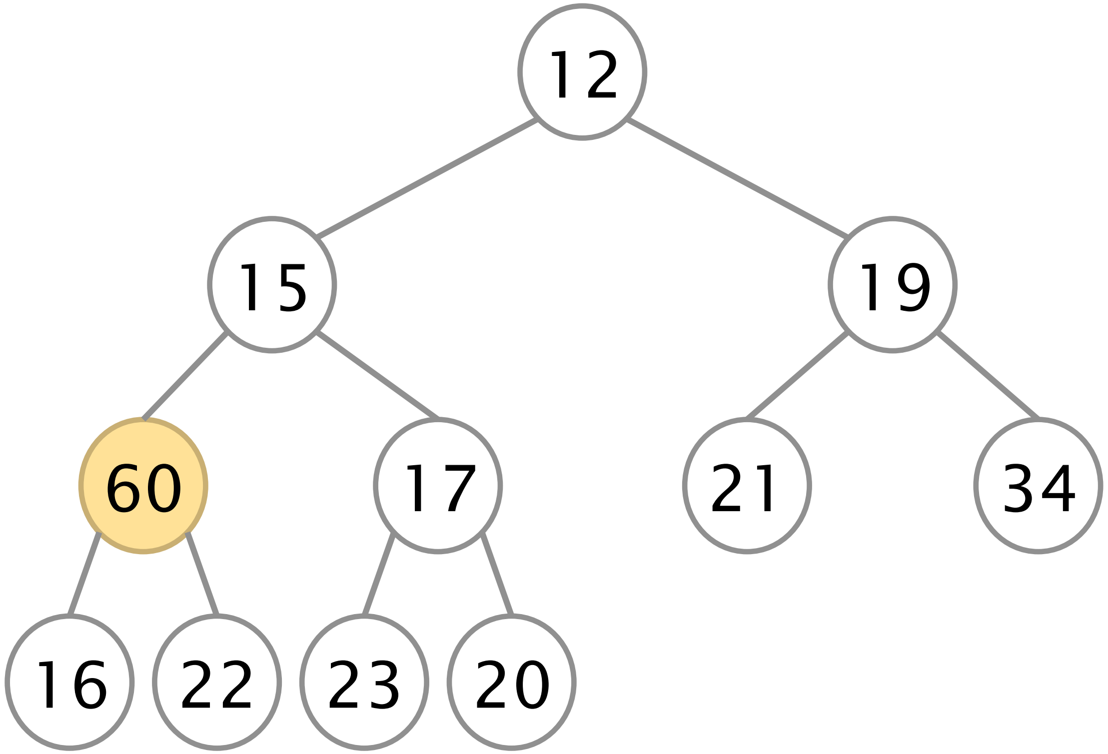
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

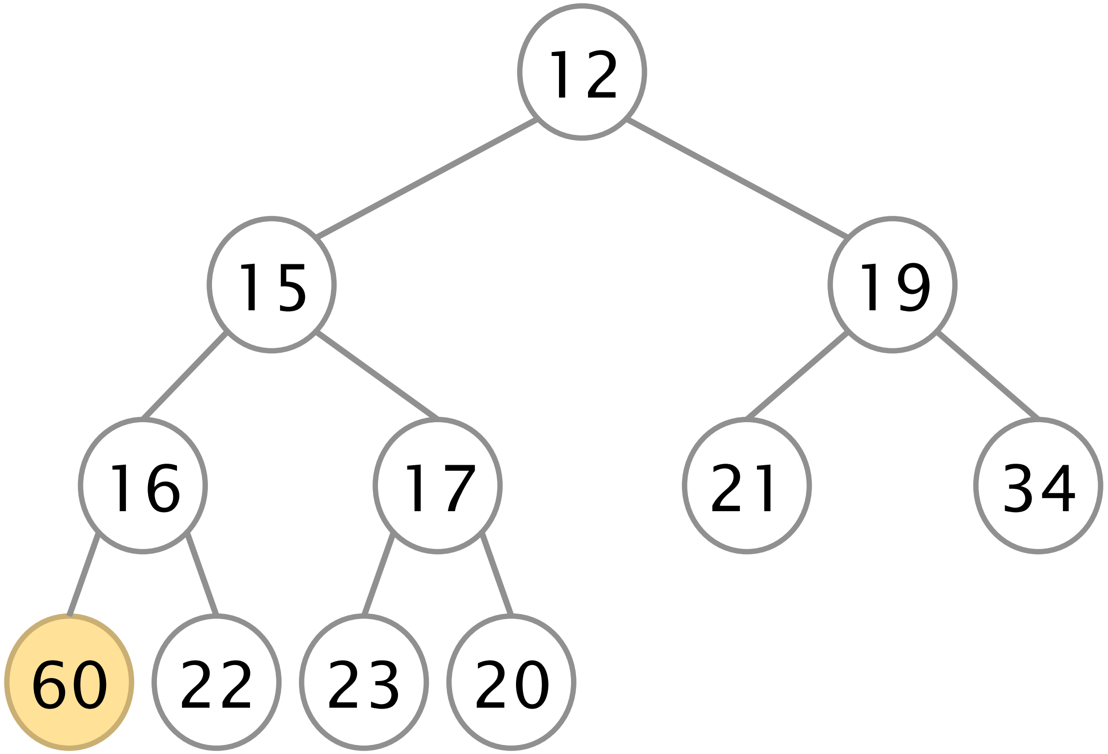
para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

para i desde n/2-1, decrementando até 0:



(elementos a serem inseridos: 23, 12, 34, 15, 60)

Complexidade do algoritmo de construção de Heap

Suponhamos que a árvore seja cheia. Então,

- $n = 2^{h+1} - 1$, onde h é a altura
- desses, apenas $2^{h} - 1$ são nós internos
- A raiz da árvore pode descer no máximo h níveis
- Os dois nós de nível 1 podem descer $h-1$ níveis
- ...
- Os 2^{h-1} nós de nível $h-1$ podem descer 1 nível
- Logo, no total temos

$$S = 1(h) + 2(h-1) + 2^2(h-2) + \ldots + 2^{h-1}(1)$$

Complexidade do algoritmo de construção de Heap

$$S = 1(h) + 2(h-1) + 2^2(h-2) + \ldots + 2^{h-1}(1)$$

$$2S = 2(h) + 2^2(h-1) + 2^3(h-2) + \ldots + 2^h(1)$$

$$2S - S = -1(h) + 2 + 2^2 + \ldots + 2^{h-1} + 2^h$$

$$S = -h - 1 + \sum_{i=0}^{h} 2^i = -h - 1 + 2^{h+1} - 1 = -h - 2 + n = O(n)$$

HeapSort

Com os algoritmos de heap é possível ordenar um vetor:

1. Construir o heap [O(n)]
2. Para todos os elementos do heap: [O(nlog(n))]
    1. Remover o elemento topo (acertando o heap).
    2. Salvar este elemento no vetor de heap, logo após o último elemento.

À medida que os elementos vão sendo colocados no final, o heap vai diminuindo de tamanho. Ao final, o vetor está em ordem decrescente.

Para obter ordem crescente, ou inverte-se a ordem do vetor [O(n)] ou utiliza-se um heap onde a raiz é o maior de todos os elementos.