INF 1010
Estruturas de Dados Avançadas

Listas de Prioridades e *Heaps*

uma outra aplicação de árvores binárias...

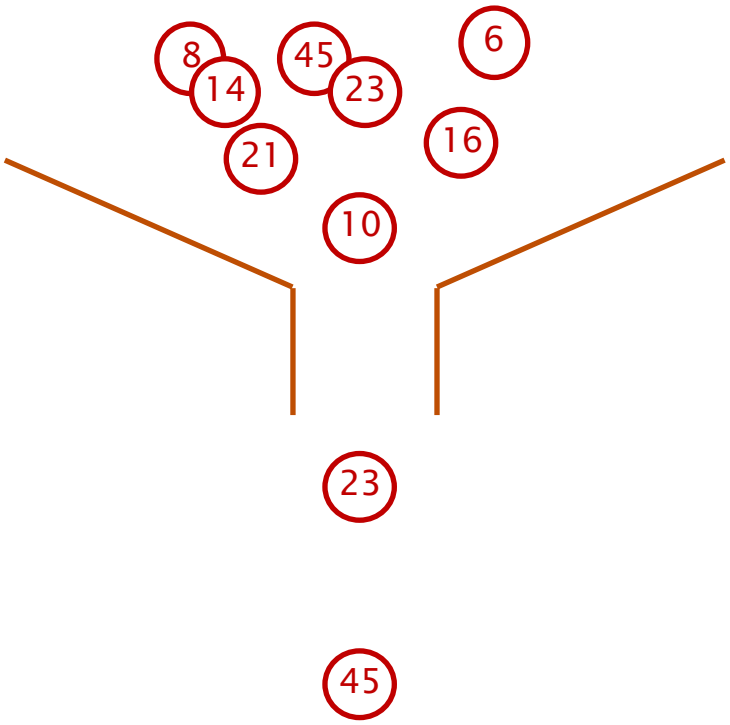
lista de prioridades:

- lista de elementos onde estamos interessados apenas no ”maior de todos” ou ”menor de todos”

  - seleção/remoção do elemento com maior (ou menor) prioridade
  - inserção de um novo elemento

outra interface!

Listas de prioridades



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.

As operações que devem ser eficientes são:

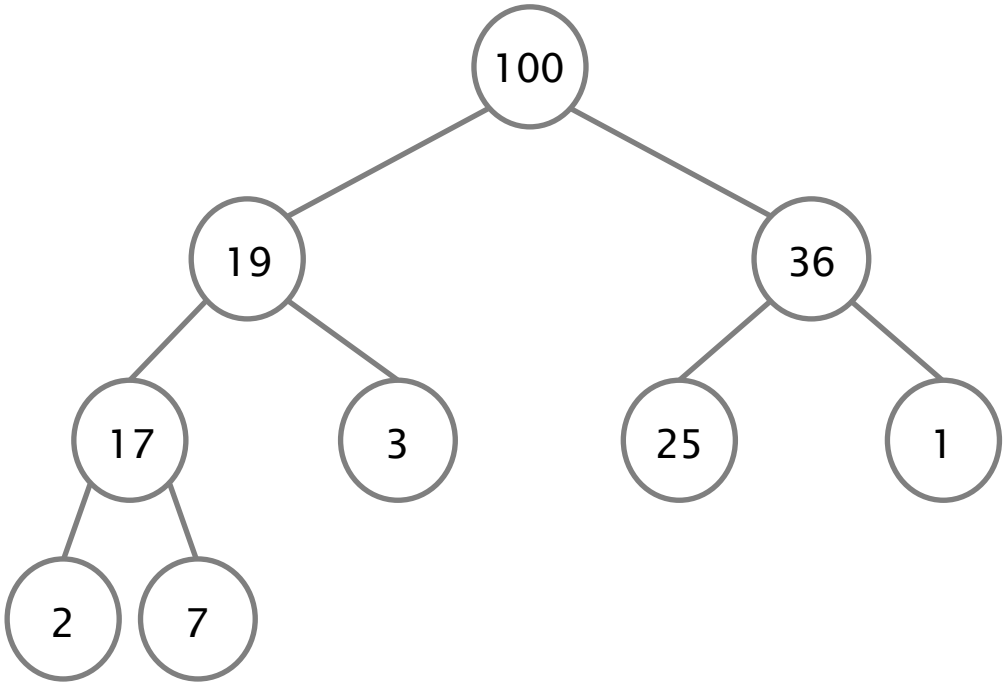
- seleção do elemento com maior (ou menor) prioridade
- remoção do elemento de maior (ou menor) prioridade
- inserção de um novo elemento

implementação usando um heap (binário)

árvore binária completa

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

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



O que é um heap (binário)

- Árvore binária completa

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

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

- e o que é uma árvore binária completa?

Árvore binária - conceitos

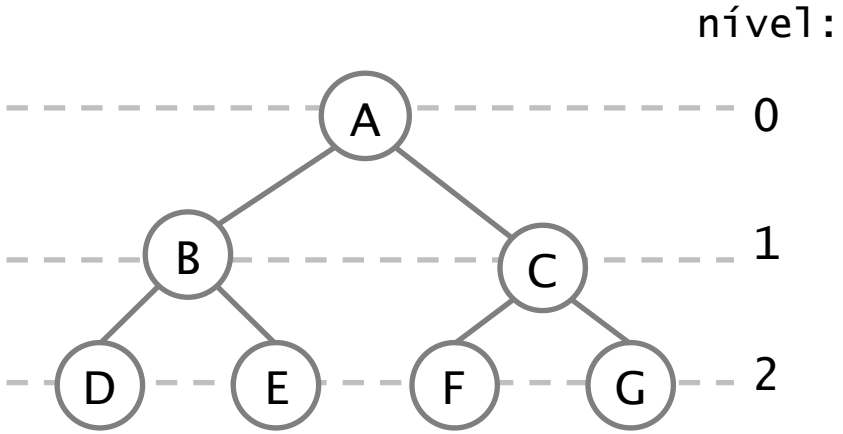
- número máximo de nós no nível *i*: $n_i = 2^i$
- número máximo de nós na árvore de altura *k*:
  - $n_{max} = 2^k + \ldots + 2^2 + 2 + 1 = 2^{k+1} - 1$

Árvore binária - conceitos

Árvore binária cheia de altura k:

árvore com $2^{k+1} - 1$ nós

Exemplo: para k = 2, árvore binária cheia possui $2^3-1 = 7$ nós

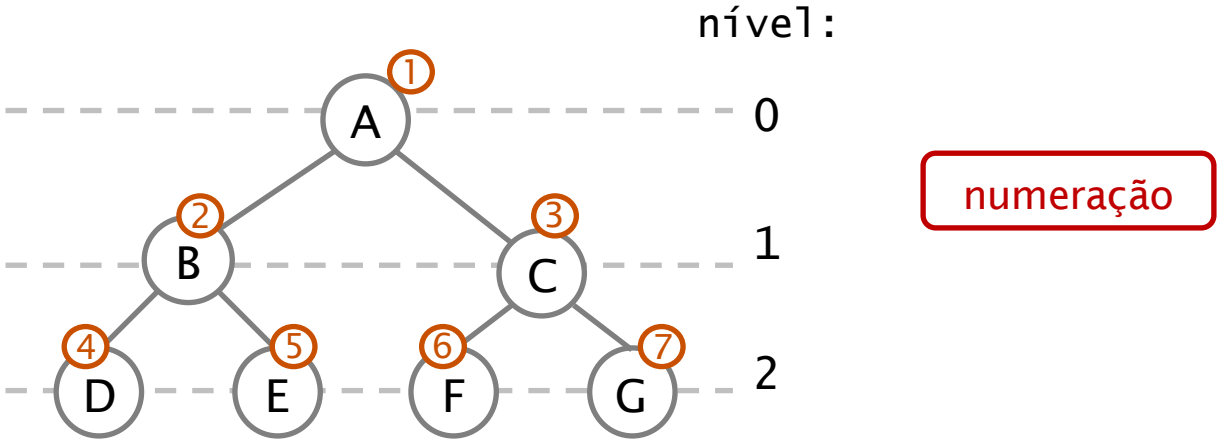


Árvore binária - conceitos

Árvore binária cheia de altura k:

árvore com $2^{k+1} - 1$ nós

Exemplo: para k = 2, árvore binária cheia possui $2^3-1 = 7$ nós



Árvore binária - conceitos

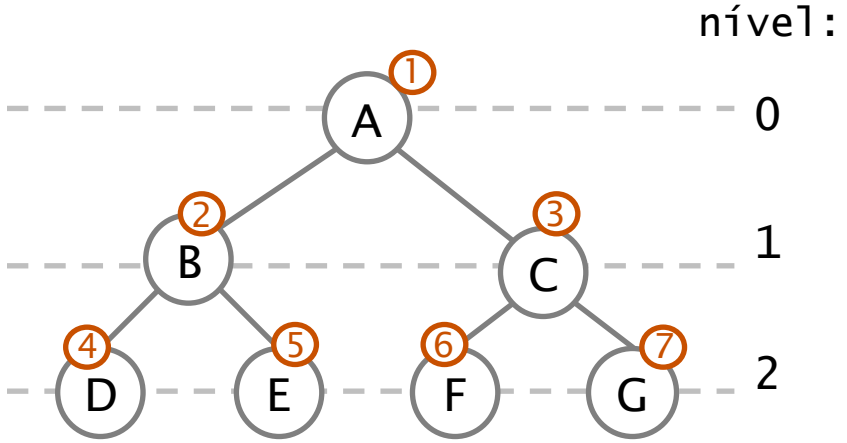
Árvore binária **cheia** de altura k:

árvore com $2^{k+1} - 1$ nós

Exemplo: para k = 2,

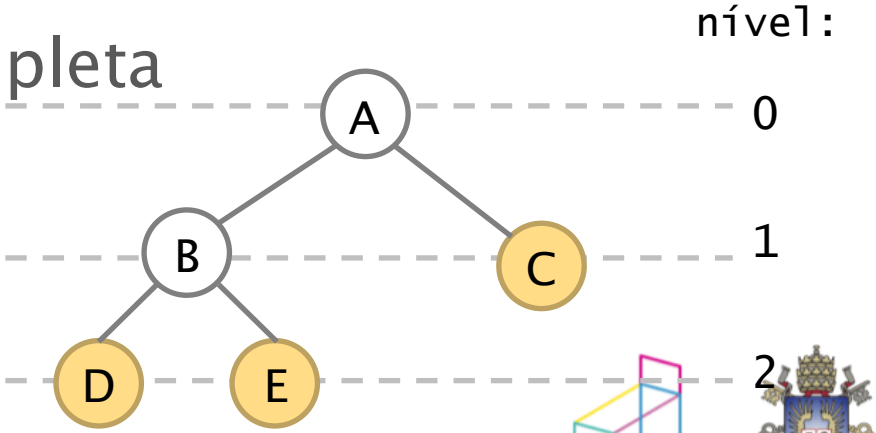
árvore binária cheia

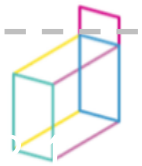
possui $2^3-1 = 7$ nós



Árvore binária **completa**

- nós de 1 a n em árvore completa
- toda folha está no último ou penúltimo nível



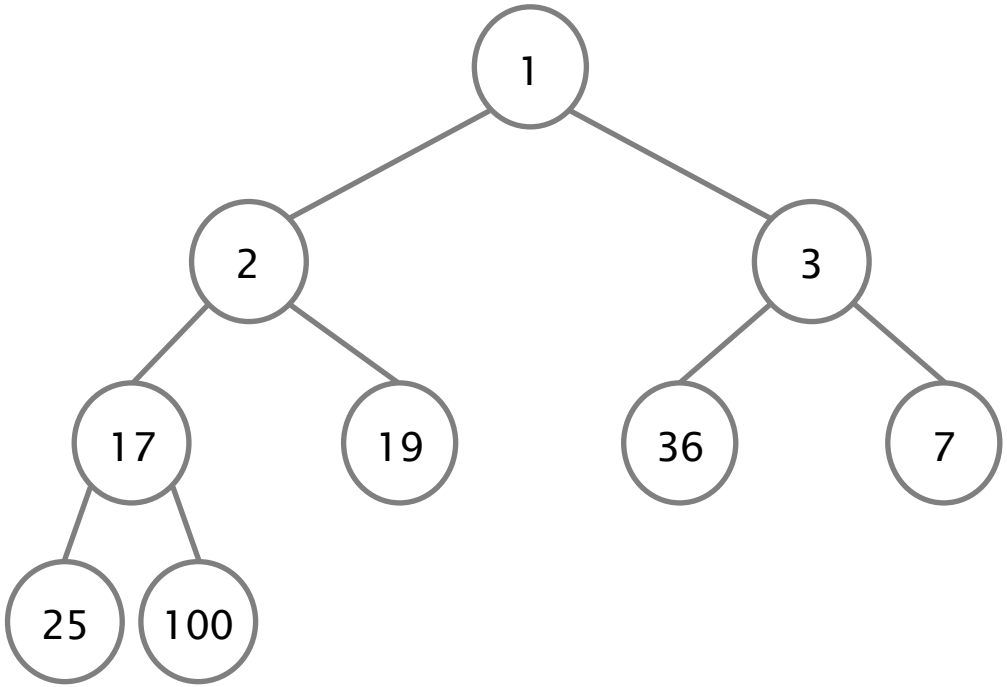


O que é um heap (binário)

Árvore binária completa

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

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

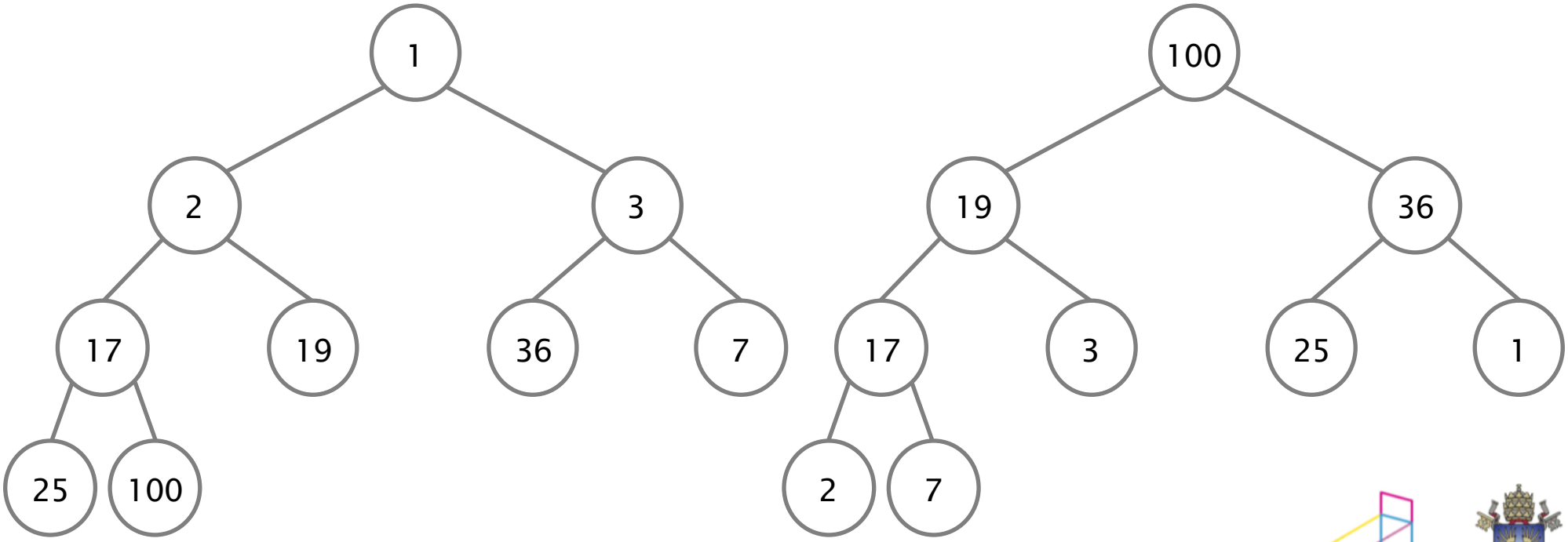


O que é um heap (binário)

Árvore binária completa

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

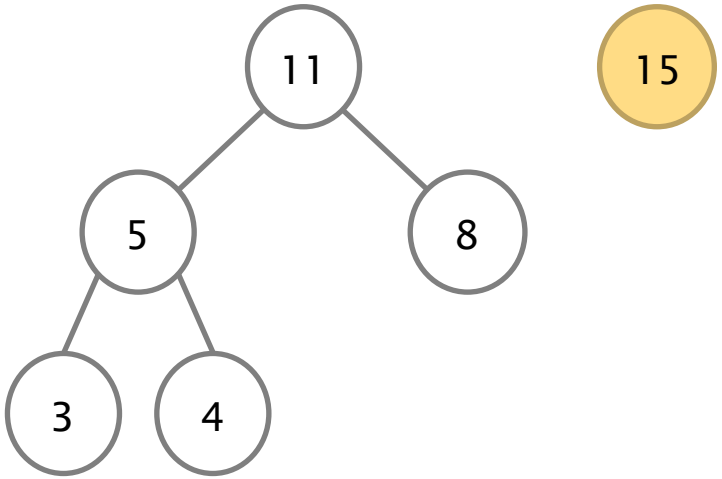
Max heap: Cada nó é **maior** que seus filhos



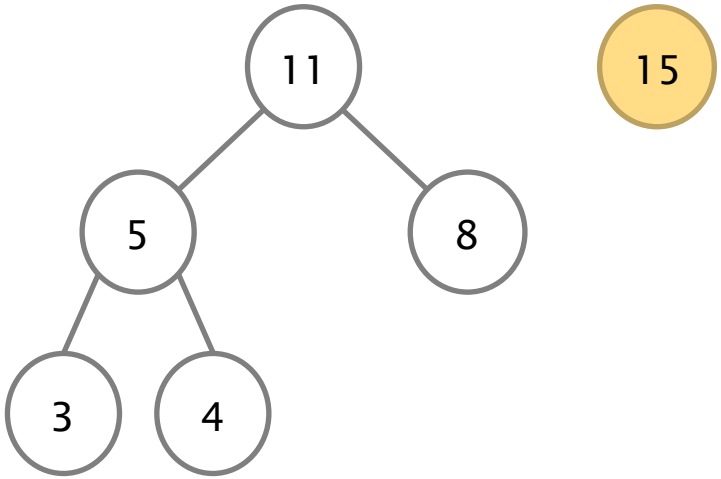
Inserir um elemento

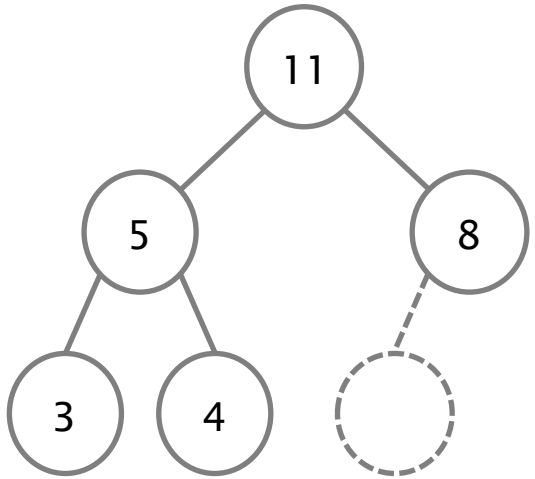
Insira o elemento no final do heap e faça-o ”subir” até a posição correta

Exemplo de inserção

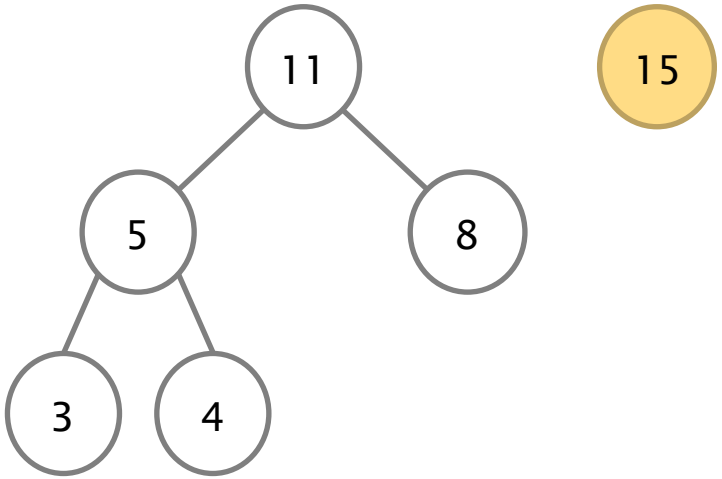


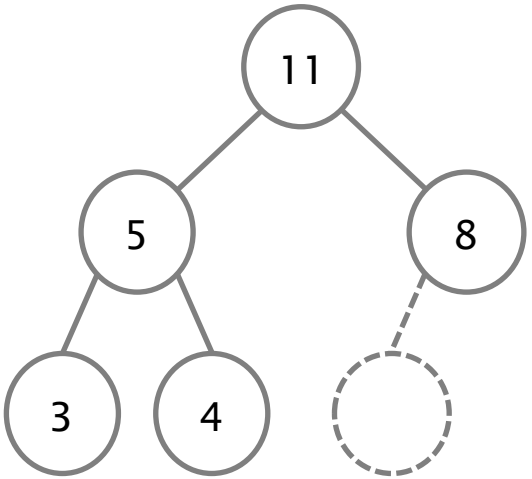
Exemplo de inserção

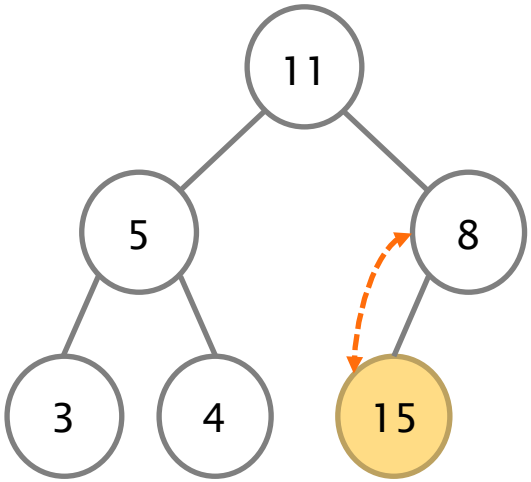




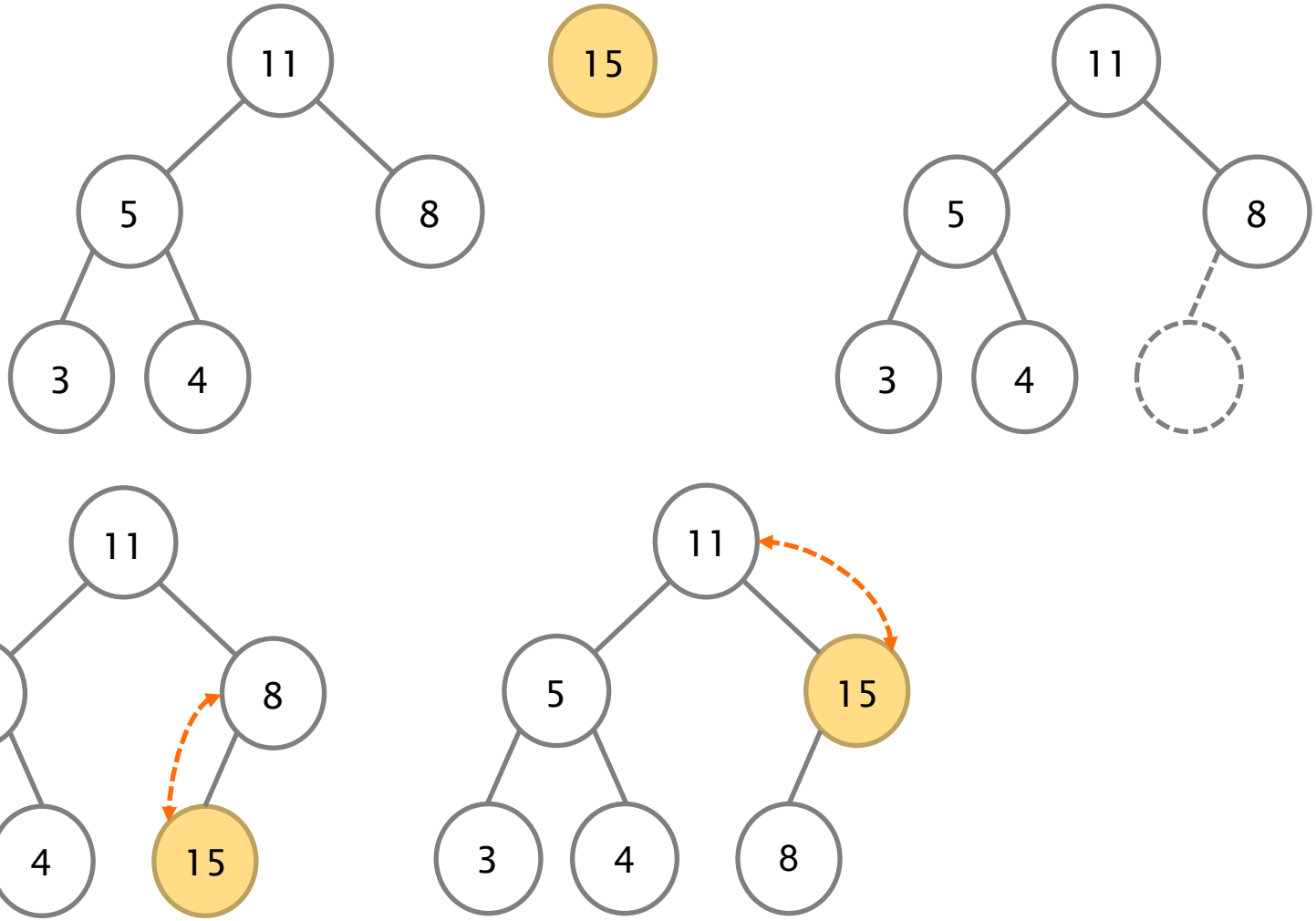
Exemplo de inserção



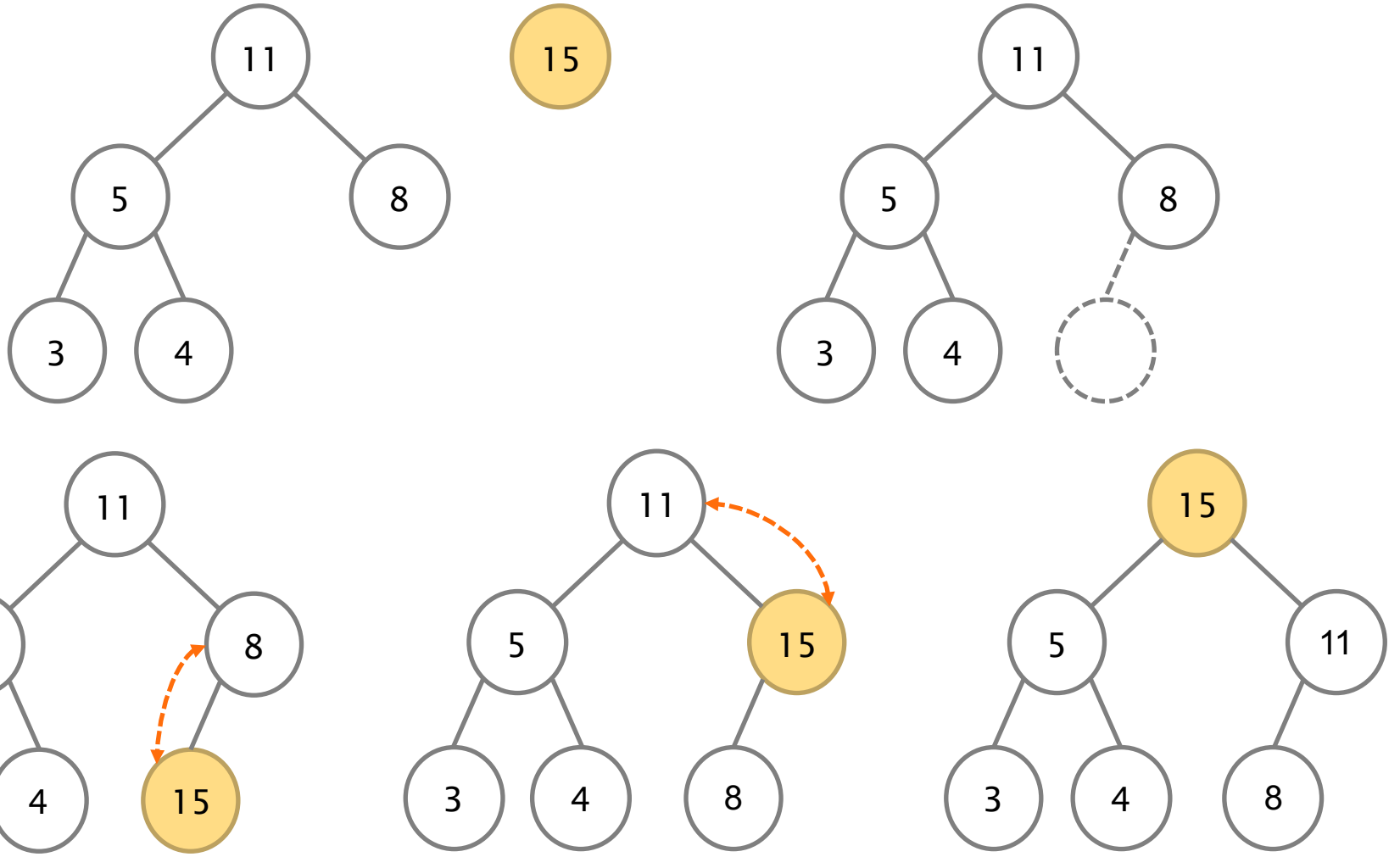




Exemplo de inserção



Exemplo de inserção



Inserir um elemento

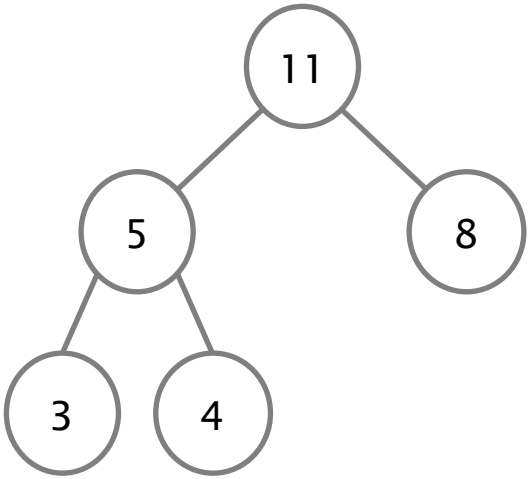
1. Insira o elemento no final do heap
2. Compare ele com seu pai:
   a. Se estiver em ordem, a inserção terminou.
   b. Se não estiver, troque com o pai e repita o passo 2 até terminar ou chegar à raiz.

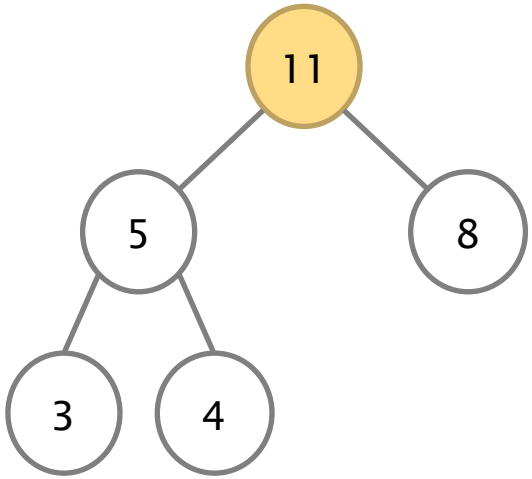
Remoção

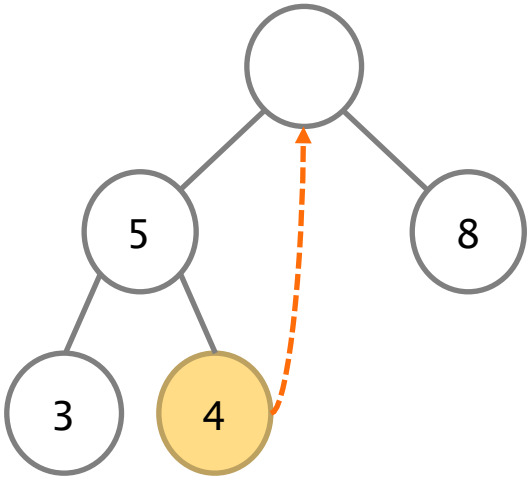
Retira-se sempre a raiz

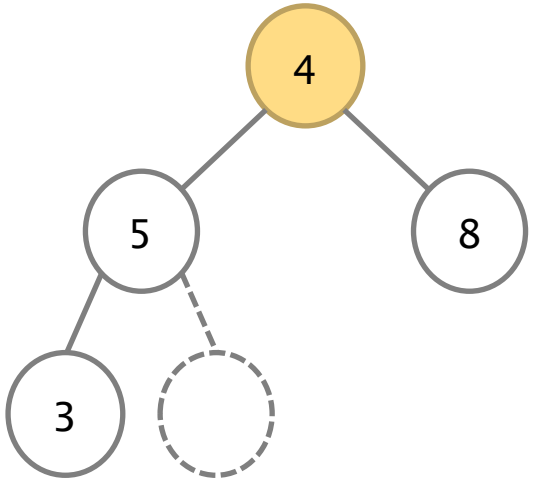
Coloque na raiz o último elemento e faça-o ”descer” até a posição correta

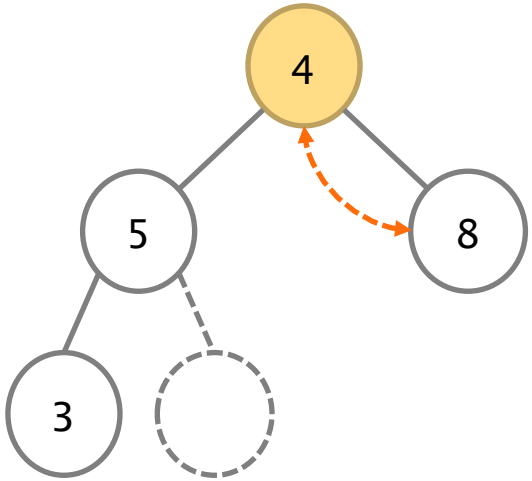
Exemplo de remoção

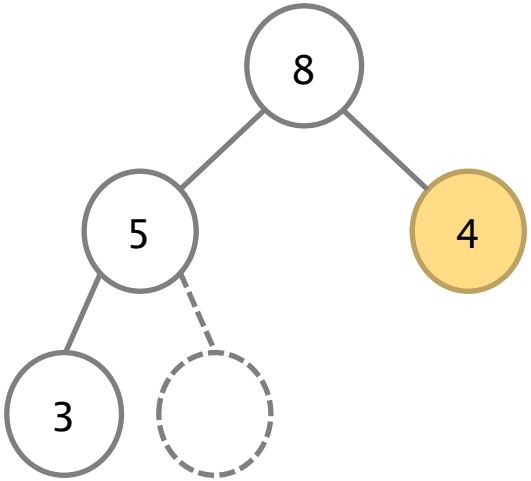












Remoção (do topo)

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

Complexidade

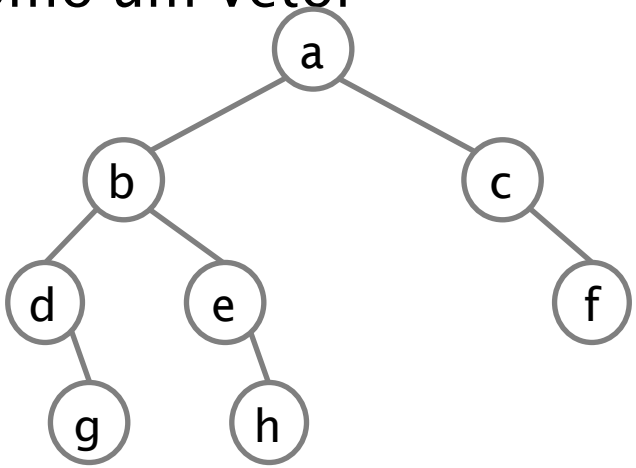
Operação	Lista	Lista Ordenada	Árvore (Balanceada)	Heap
Seleção	$O(n)$	$O(1)$	$O(\log n)$	$O(1)$
Inserção	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$
Remoção	$O(n)$	$O(1)$	$O(\log n)$	$O(\log n)$
Construção	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

Implementando heaps

Implementando árvores binárias com vetores

Podemos representar uma árvore binária como um vetor

- nó filho esquerdo de i : **2*i +1**
- nó filho direito de i : **2*i + 2**
- 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**	...
nível	0	1		2				3				

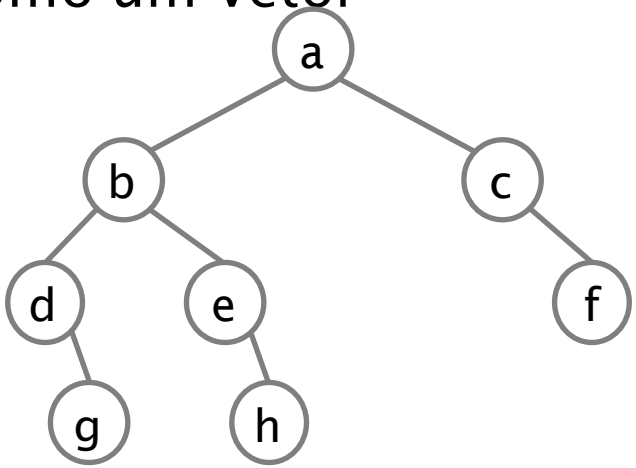
Implementando árvores com vetores

Podemos representar uma árvore binária como um vetor

nó filho esquerdo de i : **2*i +1**

nó filho direito de i : **2*i + 2**

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**	...
nível	0	1		2				3				

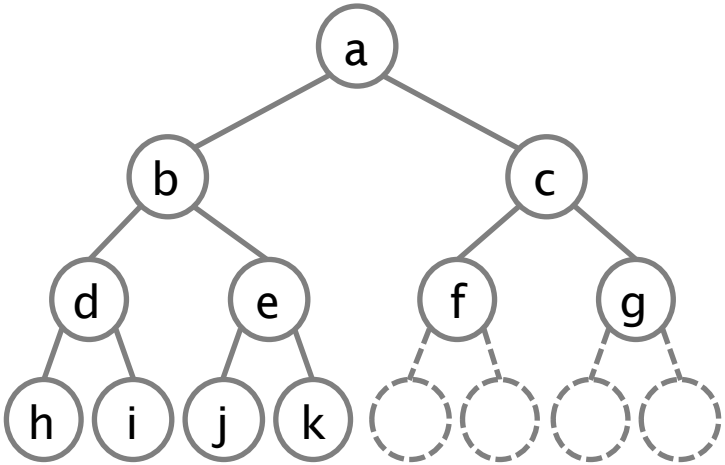
...mas podemos ter bastante memória desperdiçada no caso geral!

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*).

PUC RIO

Implementando heap com vetor

- no caso de heaps, as árvores são completas!



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				

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

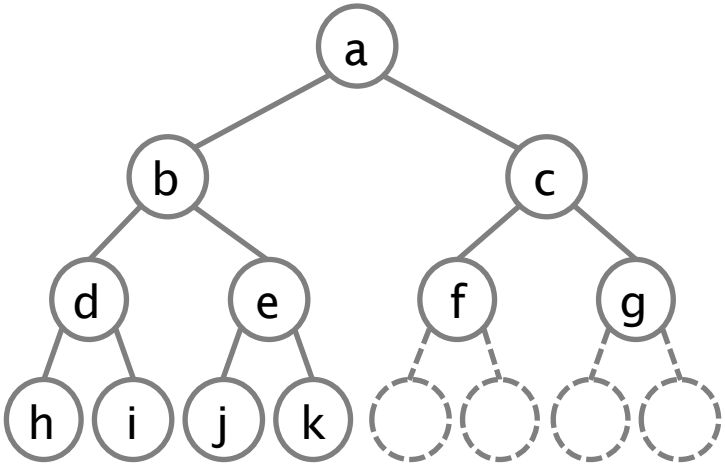
Implementando heap com vetor

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**

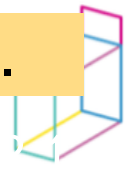
do nó pai de i : **(i-1)/2**



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.



PUC RIO

Implementação de um módulo Heap

```
typedef struct _heap Heap;

Heap* heap_cria(int max);
void  heap_insere(Heap* heap, int prioridade, dados* d);
dados * heap_remove(Heap* heap);
```

Implementação de um módulo Heap

```
struct _heap {
  int max;          /* tamanho maximo do heap */
  int pos;          /* proxima posicao disponivel no vetor */
  int* prioridade; /* vetor das prioridades */
}; /* estou ignorando os dados! */

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

Insere

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

Insere

Insere

```
static void troca(int a, int b, int* v) {
    int 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

```
int heap_remove(Heap* heap)
{
  if (heap->pos>0) {
    int 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;
  }
}
```

Construção de Heap

Algoritmo ingênuo:

Insira um-a-um todos os n elementos.

Cada elemento é inserido na base e sobe até seu lugar.

Complexidade: $O(n \log n)$

Construção de Heap

Observe que:

- As folhas da árvore (elementos $n/2 + 1 \ldots n$) não têm descendentes e portanto já estão ordenadas em relação a eles
- Se acertarmos todos os nós internos (elementos $1 \ldots n/2$) 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 um min heap

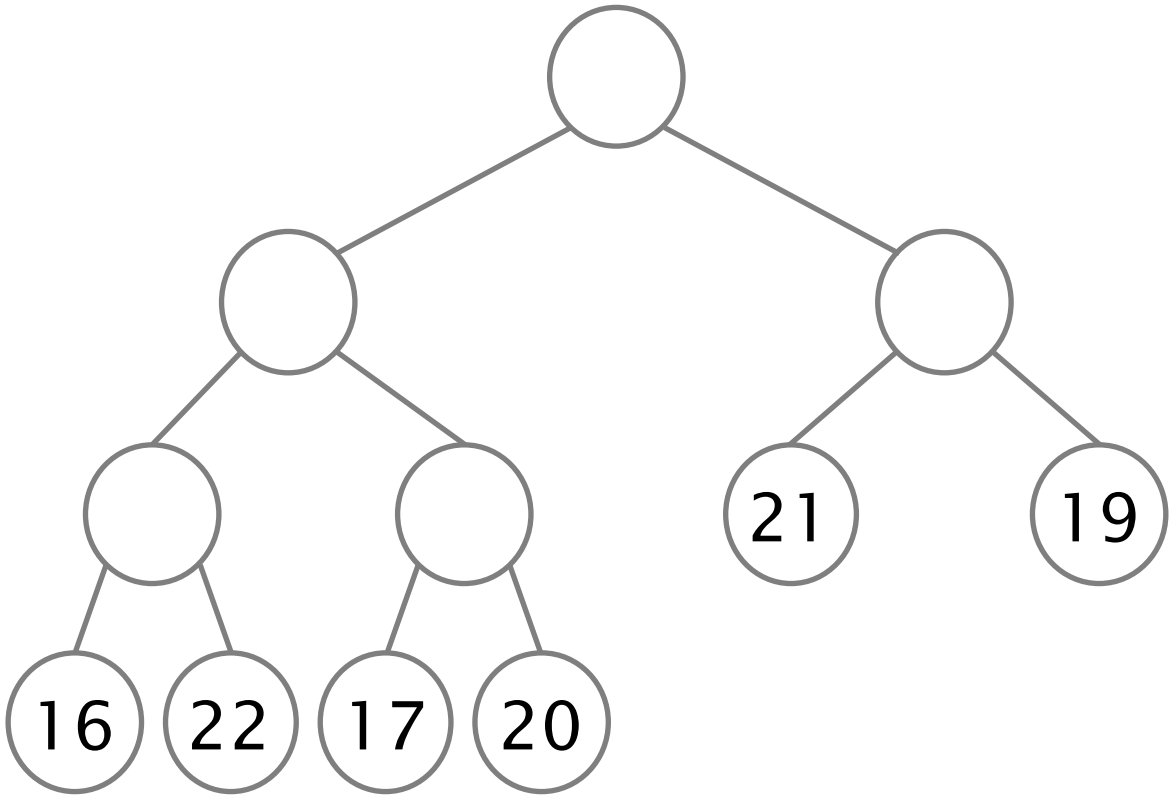
lista de 11 prioridades: 21, 19, 16, 22, 17, 20, 23, 12, 34, 15, 60

construção de um min heap

lista de 11 prioridades: 21, 19, 16, 22, 17, 20, 23, 12, 34, 15, 60

insere-se n/2+1 elementos nas últimas posições

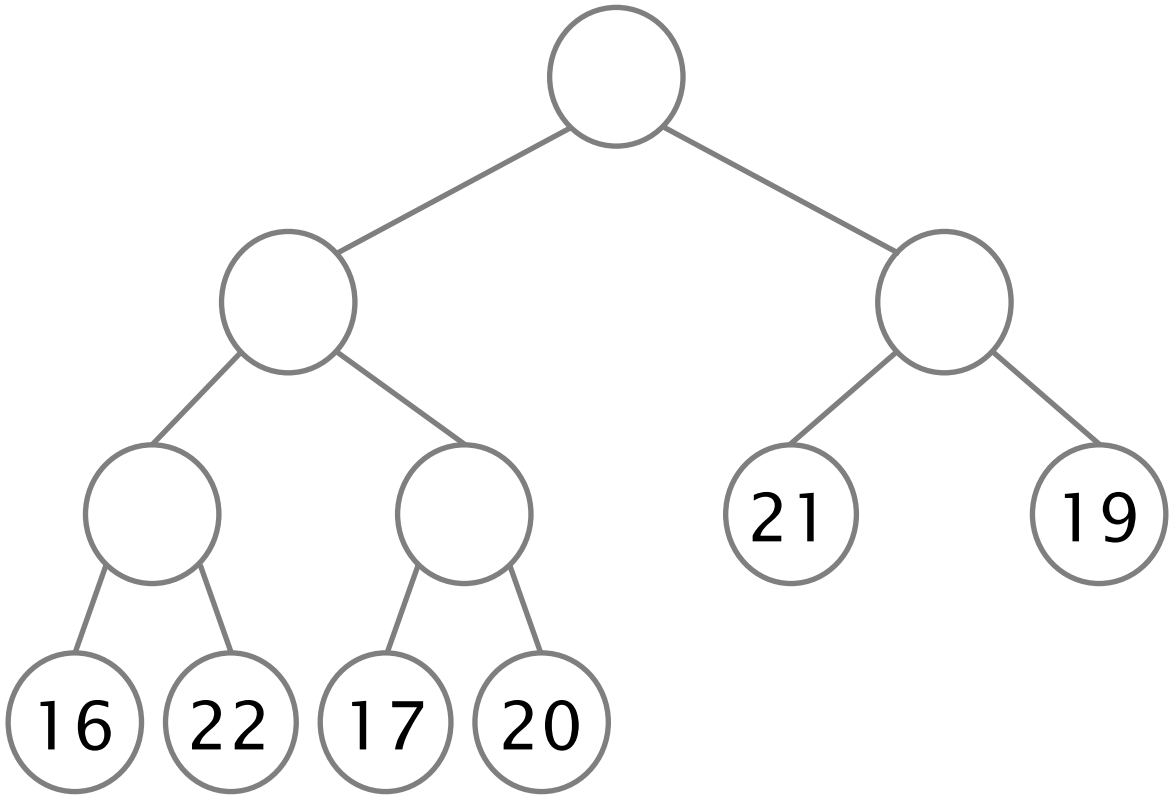
Construção de Min Heap (exemplo)



falta inserir: 23, 12, 34, 15, 60)

Construção de Min Heap (exemplo)

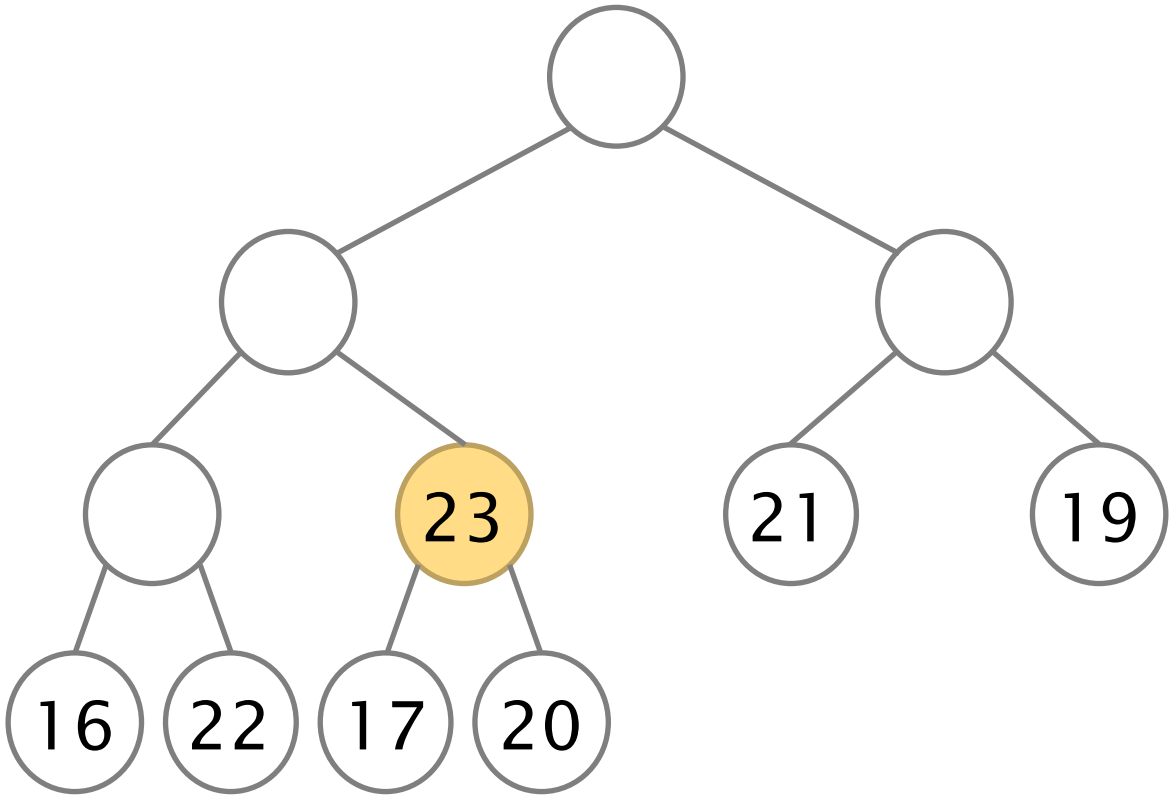
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

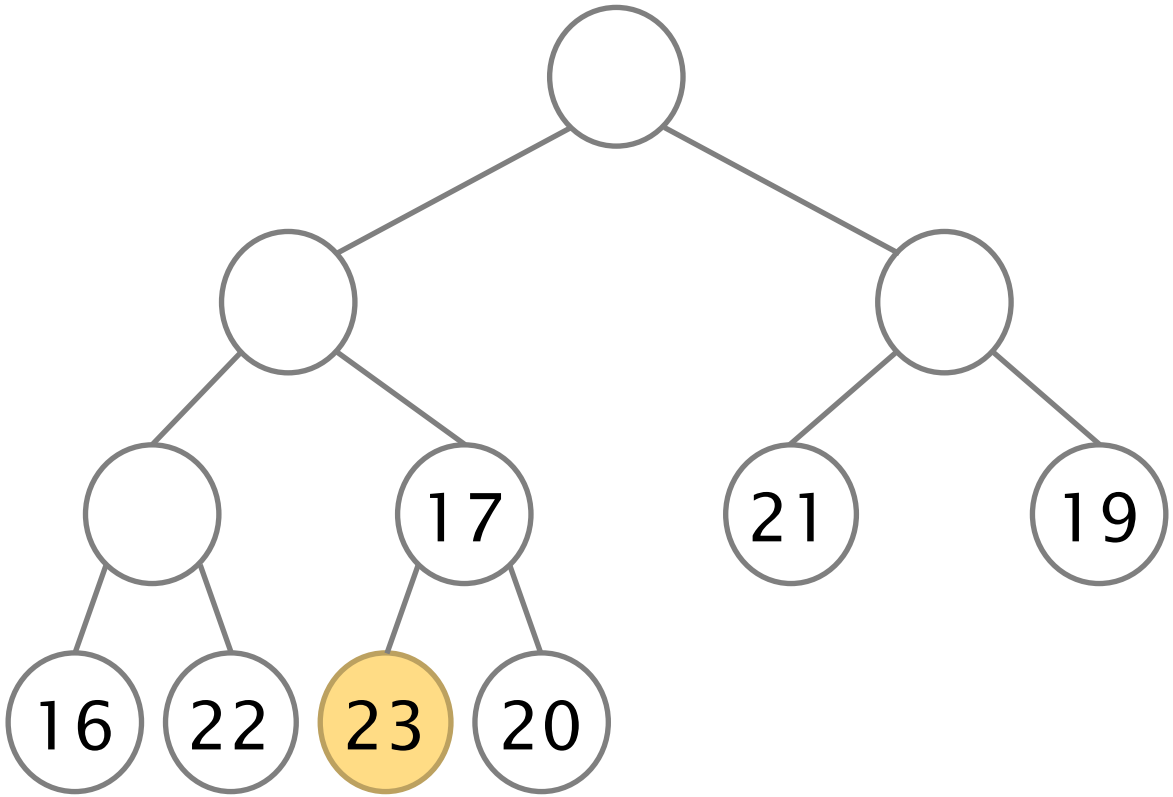
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

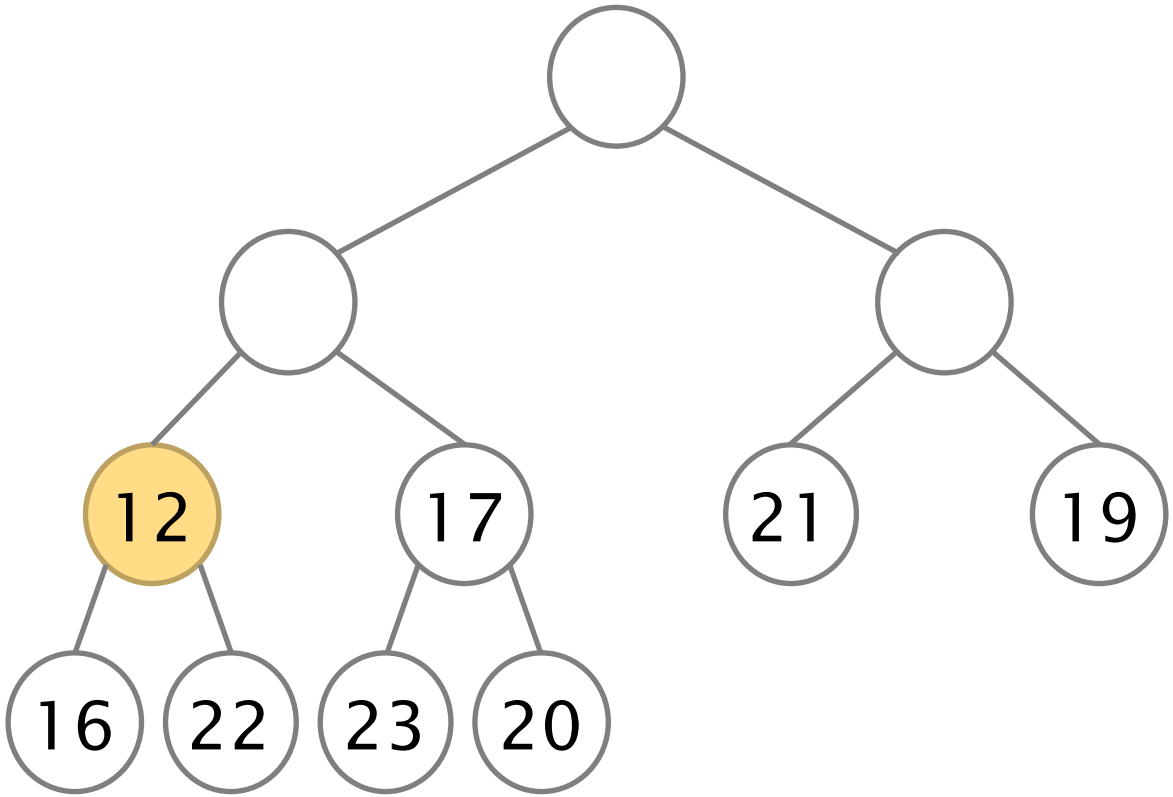
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

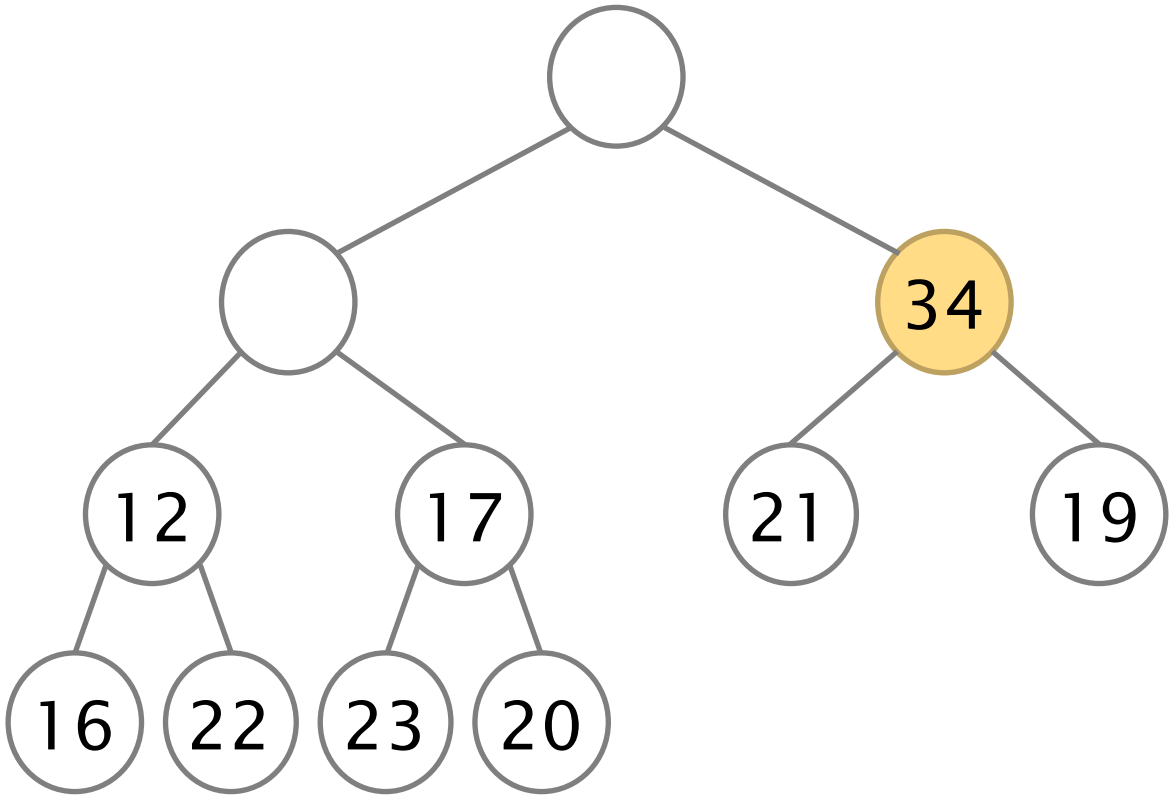
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

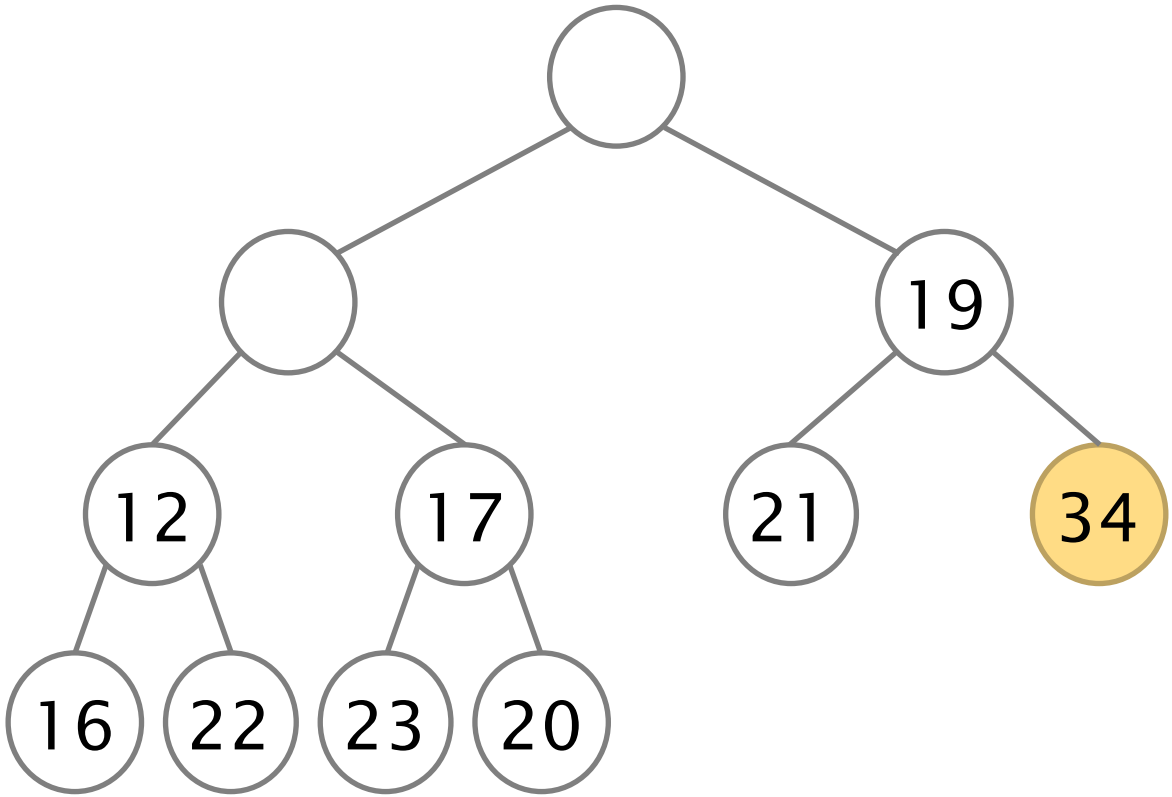
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

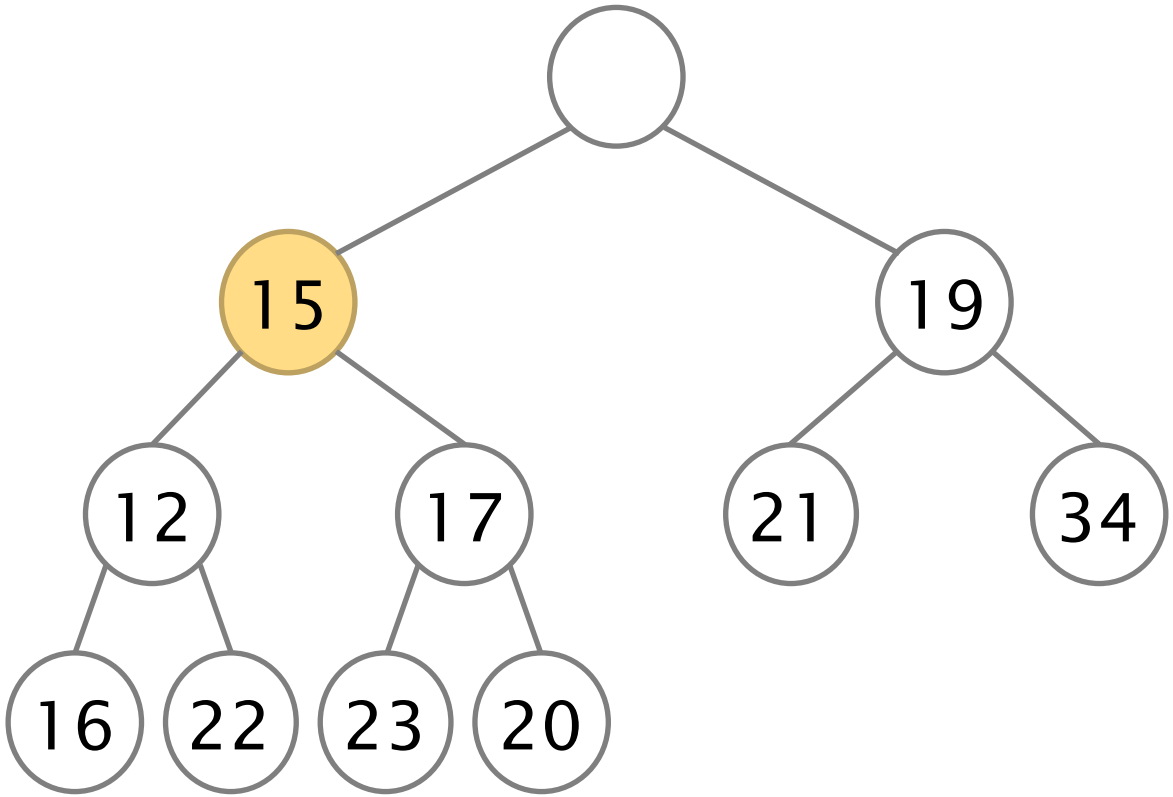
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

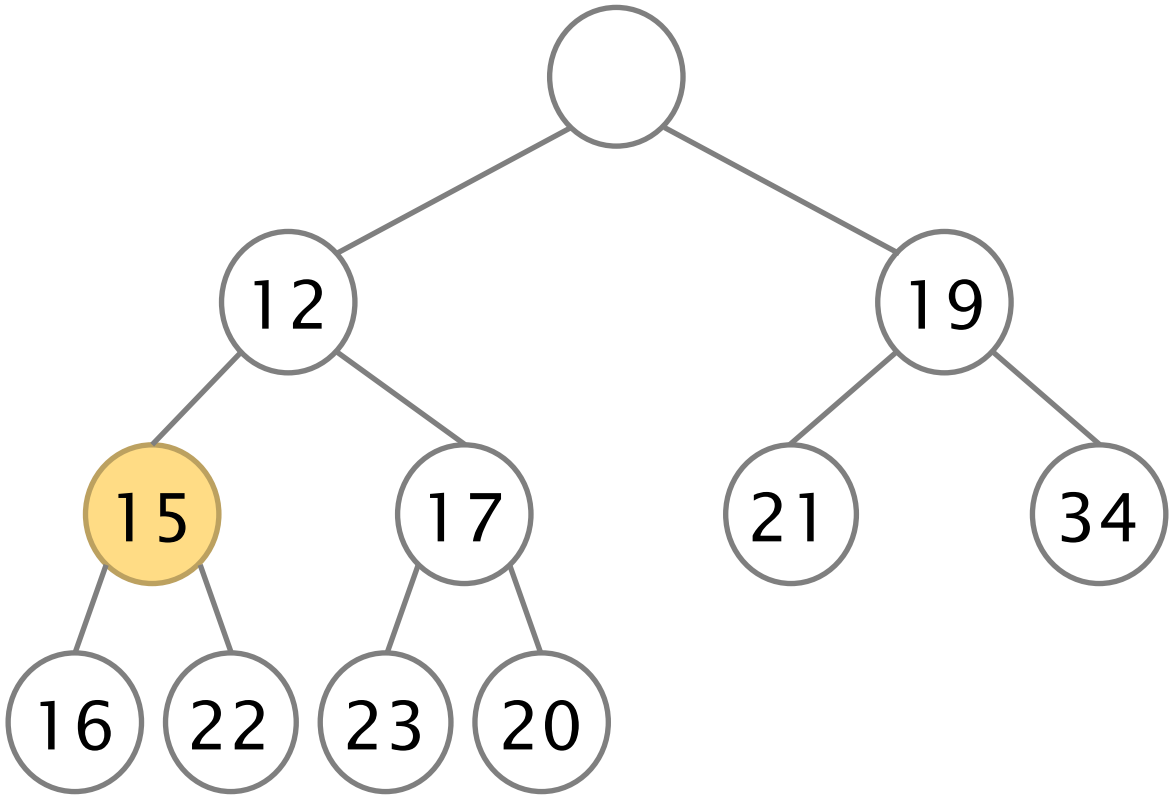
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

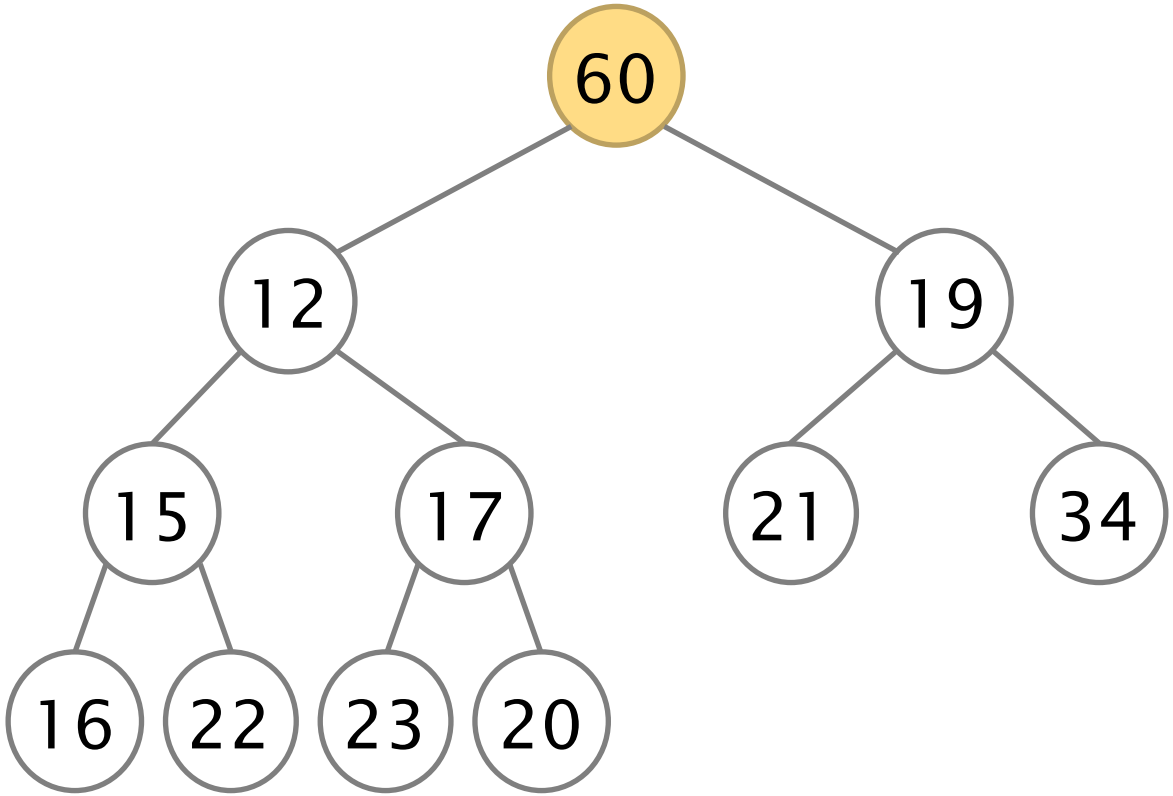
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

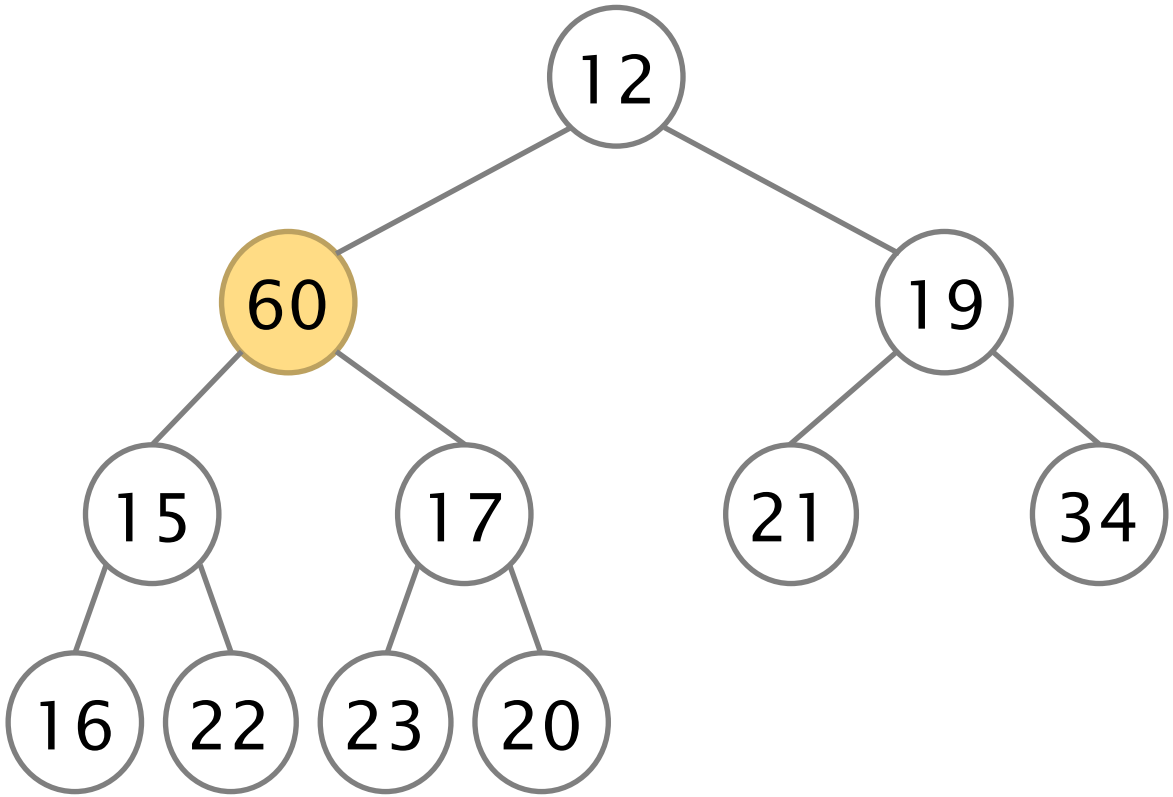
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

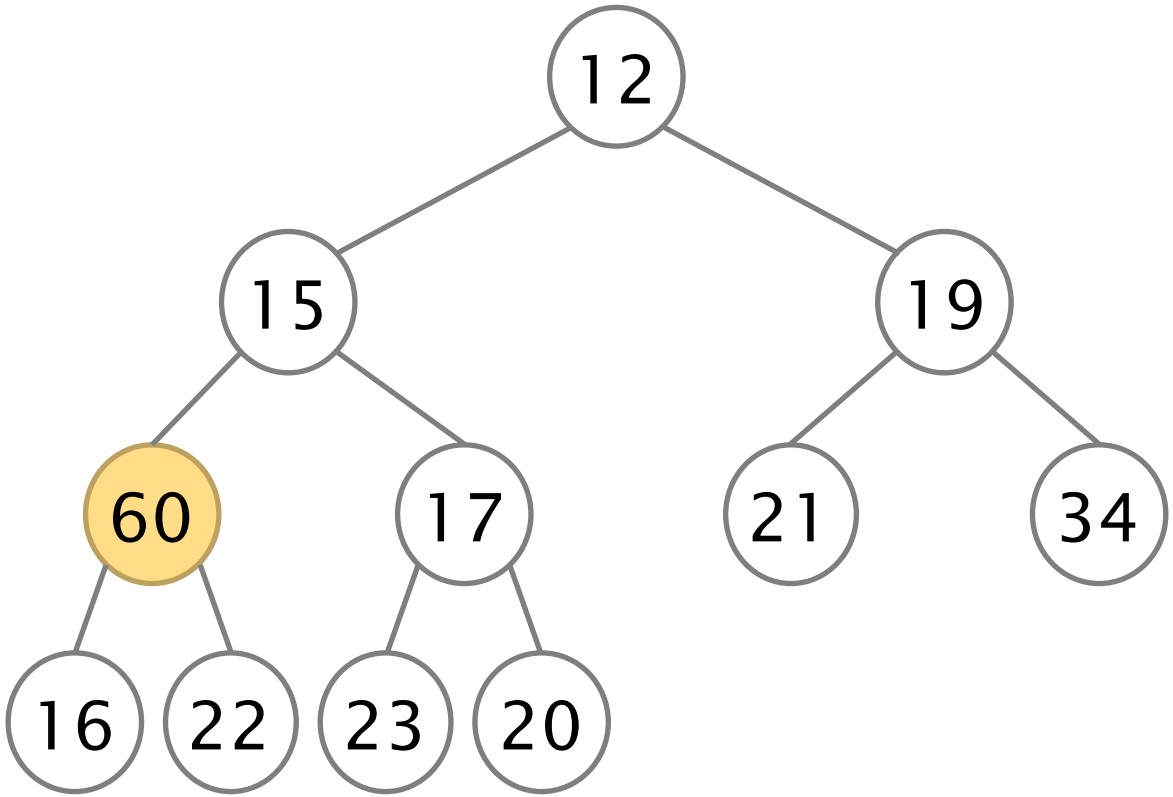
Para i, desde n/2-1, decrementando até 0:



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

Construção de Min Heap (exemplo)

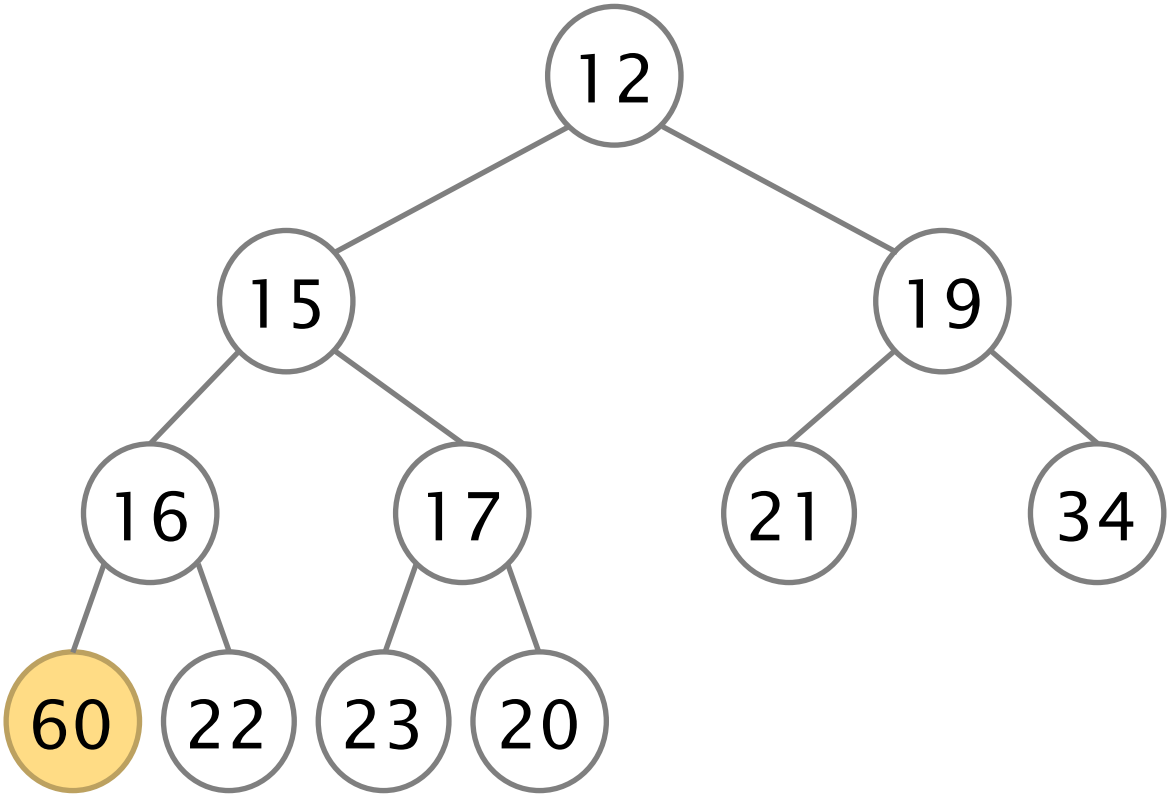
Para i, desde n/2-1, decrementando até 0:



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

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.

Destes, 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

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

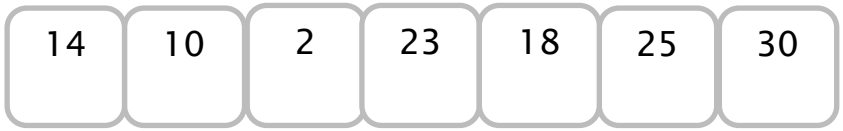
$$2S = 2(h) + 2^2(h-1) + 2^3(h-2) + \ldots + 2^h(1)$$

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

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

Logo, o algoritmo de construção é *O(n).*

heapsort



colocar em ordem crescente....

HeapSort

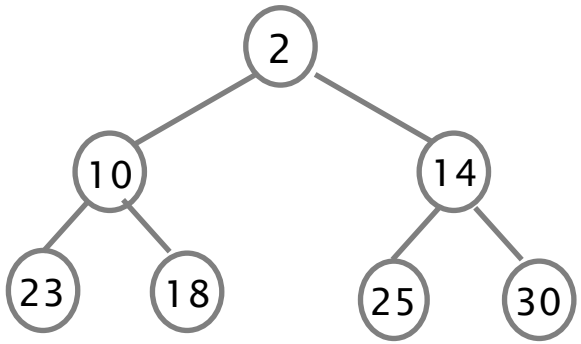
Ordenação de vetor utilizando um heap:

1. Construa o heap (complexidade O(n))
2. Para todos os elementos do heap (complexidade O(nlog(n)))
   a. Remova o elemento topo (acertando o heap)
   b. Salve este elemento no vetor de heap, logo após o último elemento

heapsort

14	10	2	23	18	25	30

ordenação do vetor



2 10 14 23 18 25 30

10 18 14 23 30 25 | 2

14 18 25 23 30 | 2 10

18 23 25 30 | 2 10 14

23 30 25 | 2 10 14 18

25 30 | 2 10 14 18 23

30 | 2 10 14 18 23 25

| 2 10 14 18 23 25 30

HeapSort

Construção intuitiva:

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

HeapSort

Ordenação de vetor utilizando um heap:

1. Construa o heap (complexidade O(n))
2. Para todos os elementos do heap (complexidade O(nlog(n)))
   a. Remova o elemento topo (acertando o heap)
   b. Salve este elemento no vetor de heap, logo após o último elemento

Construção intuitiva:

- 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:

- inverta a ordem do vetor (complexidade O(n)), ou
- utilize um heap onde a raiz é o maior de todos os elementos