Estruturas de Dados Avançadas (INF1010)

Listas de Prioridade e *Heaps*

Listas de Prioridades

Em algumas aplicações, dados de coleções são acessados por ordem de prioridade

- fila de impressão, escalonamento de tarefas, simulações
- valor ordenável: tempo, custo, ...

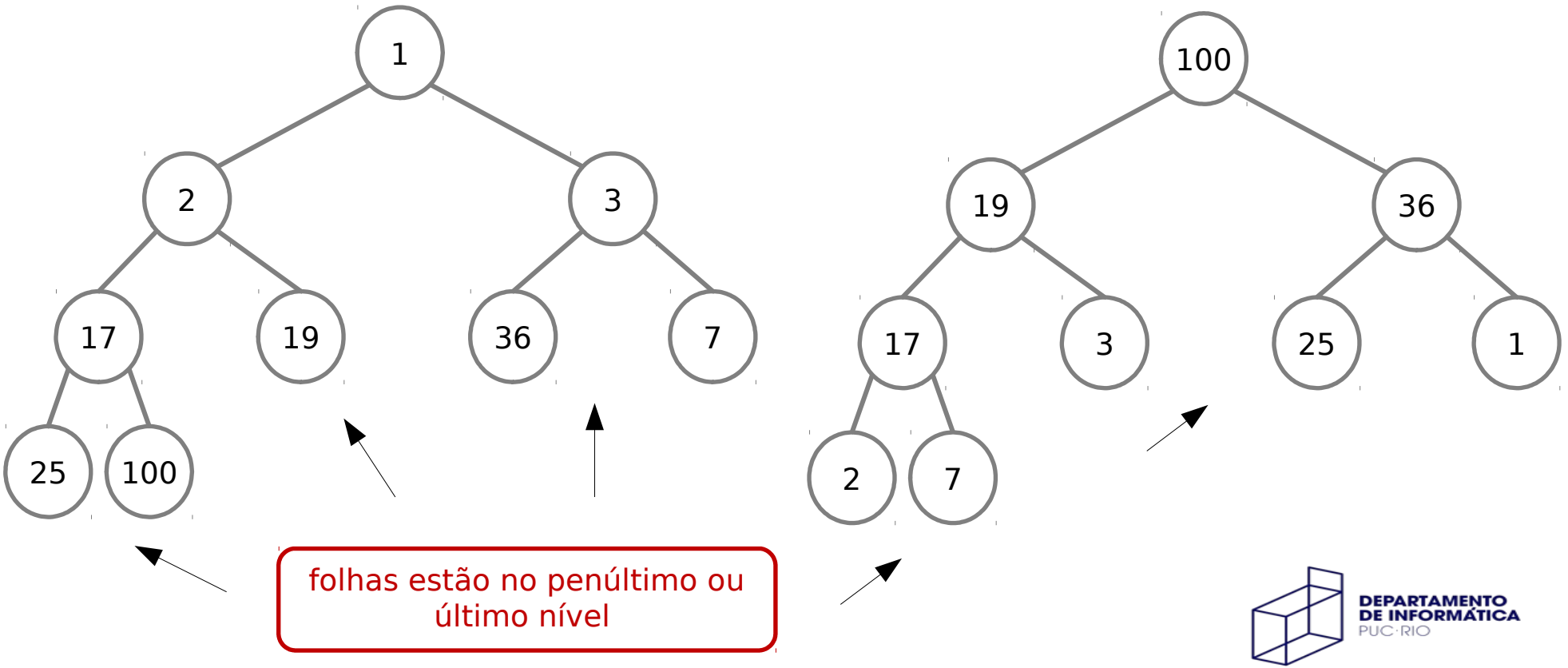
Operações que devem ser eficientes:

- seleção/remoção de elemento com maior/menor prioridade
- inserção de um novo elemento

Heap (binário)

Árvore Binária completa

- **min** heap: cada nó é **menor** que seus filhos
- **max** heap: cada nó é **maior** que seus filhos



Inserção

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

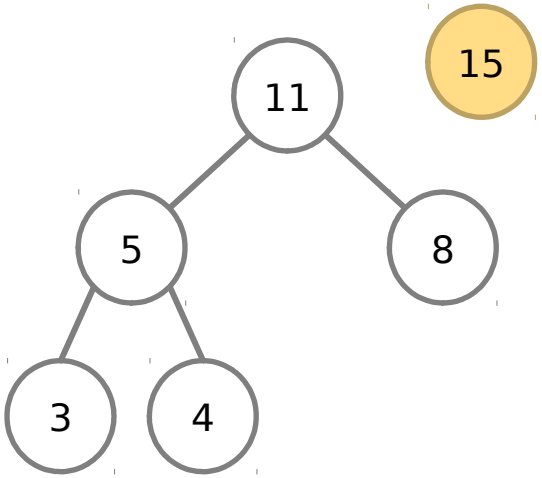
Inserção

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

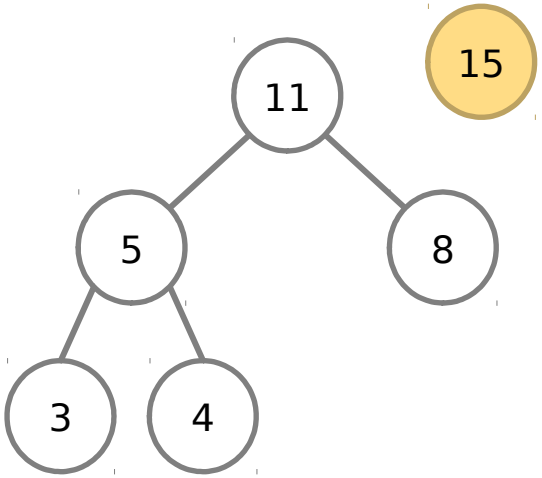
Compare o elemento com seu pai:

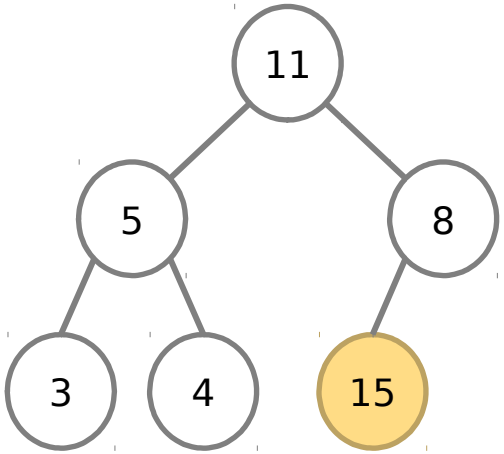
- se estiver em ordem, a inserção terminou
- se não estiver, troque com o pai e repita o processo até terminar ou chegar à raiz

Exemplo de Inserção

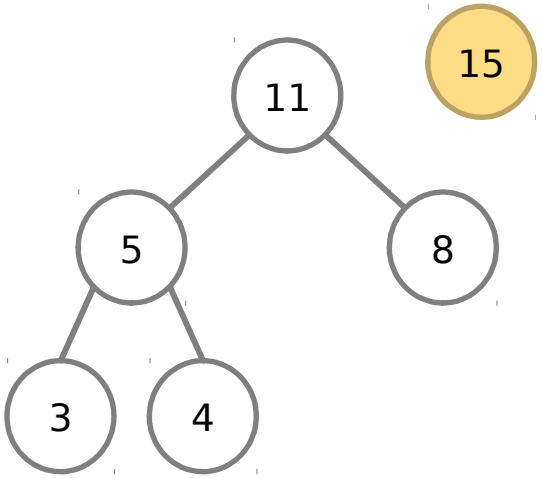


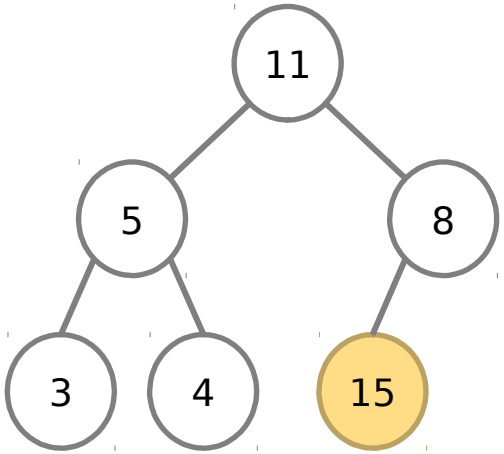
Exemplo de Inserção

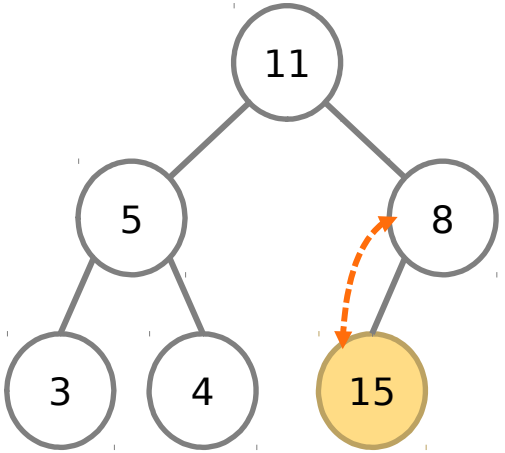




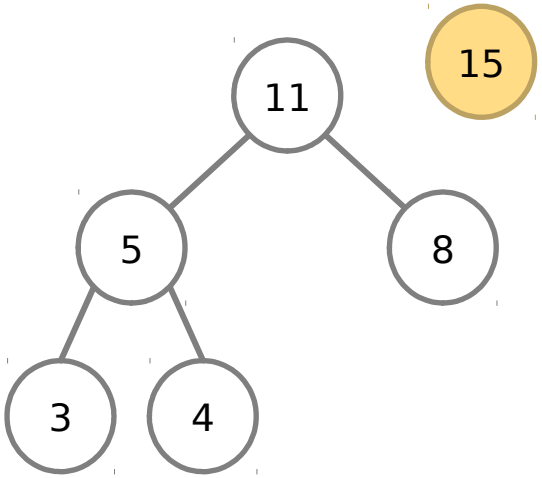
Exemplo de Inserção

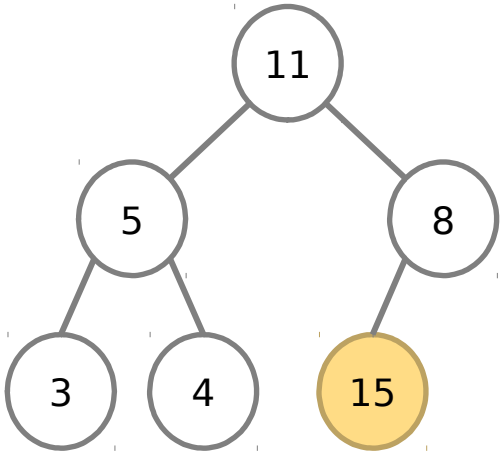


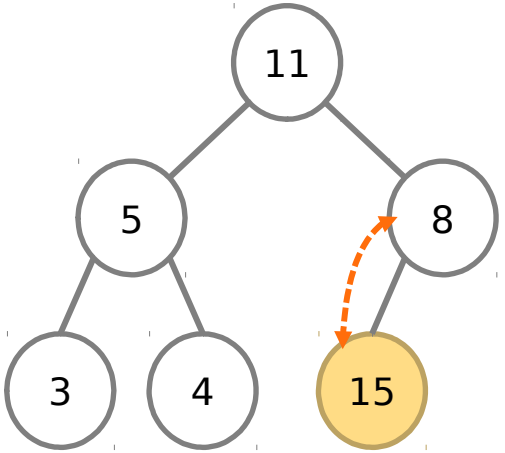


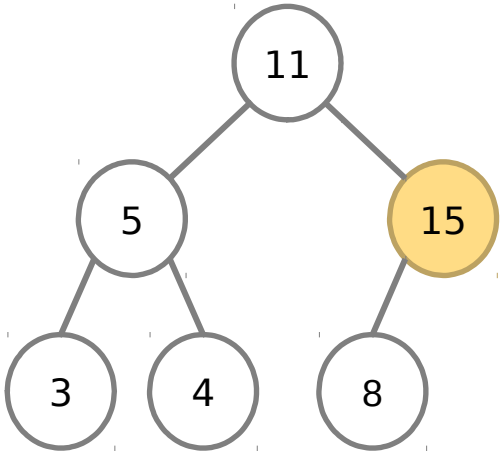


Exemplo de Inserção

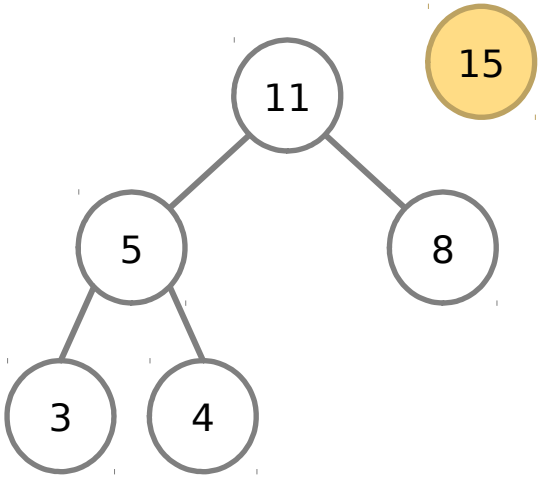


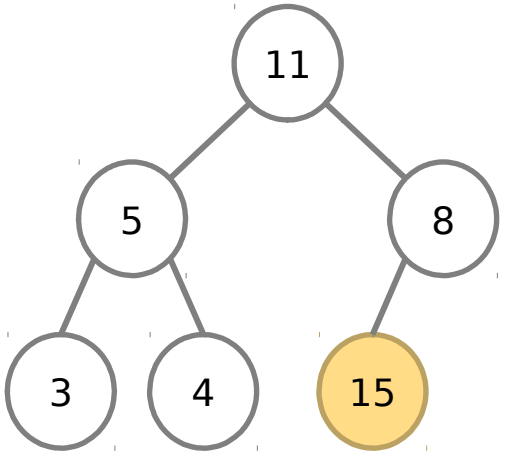


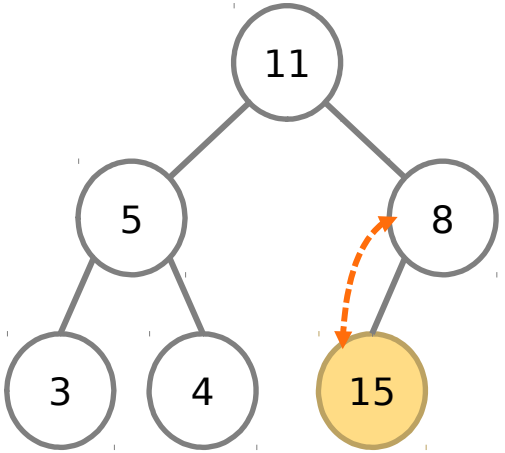


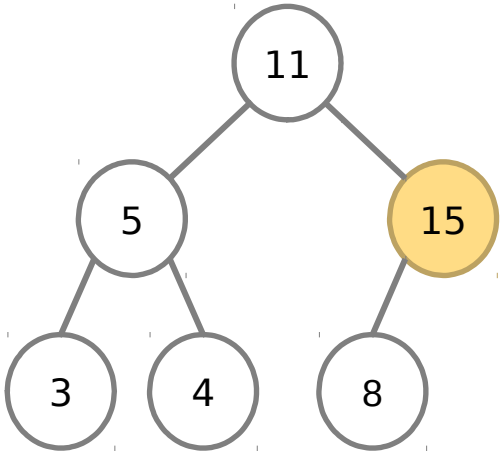


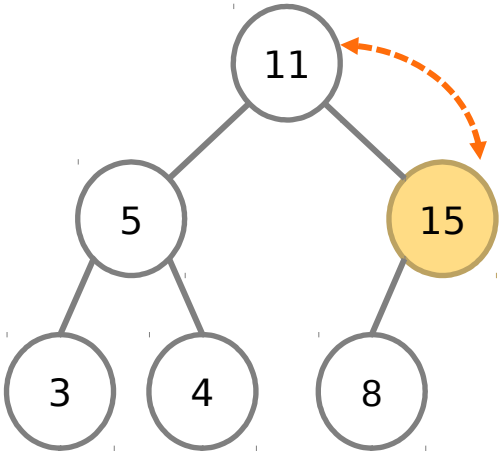
Exemplo de Inserção





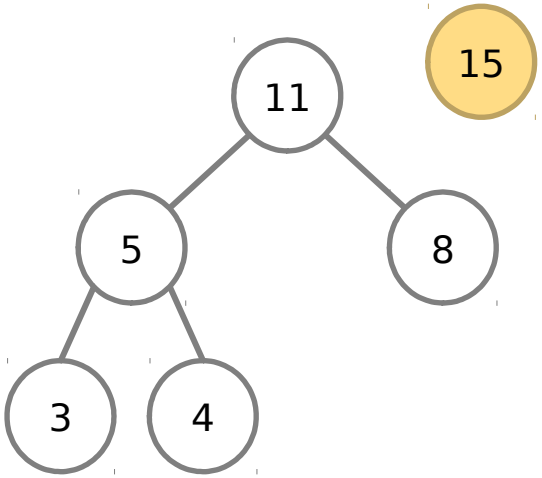


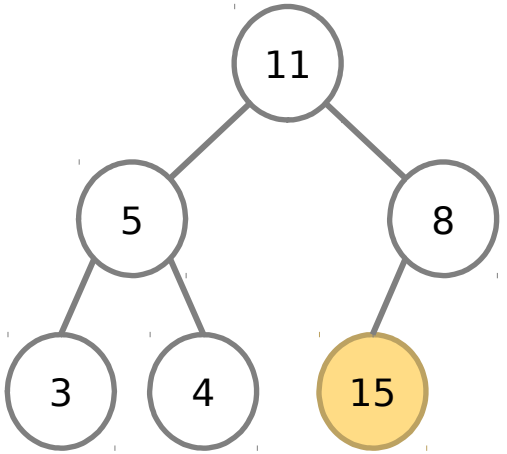


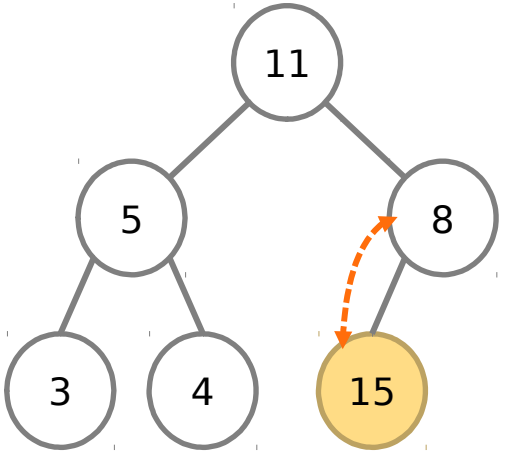


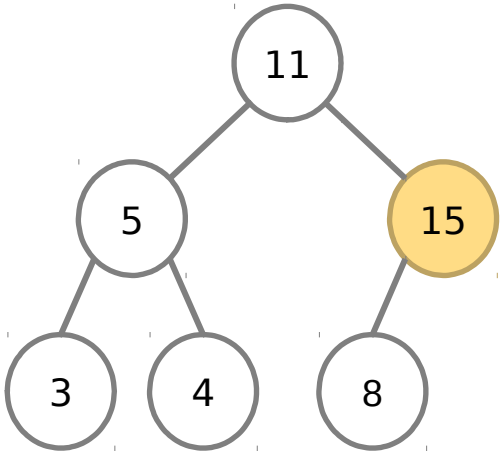
DEPARTAMENTO DE INFORMÁTICA
PUC·RIO

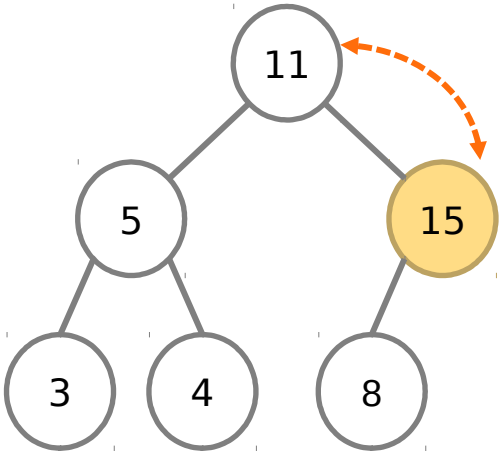
Exemplo de Inserção

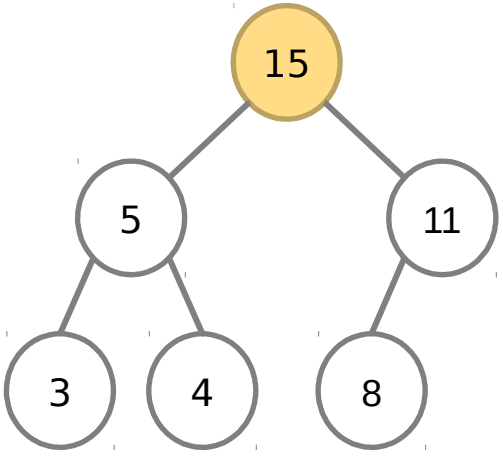












DEPARTAMENTO DE INFORMÁTICA
PUC·RIO

Remoção

Retira-se sempre a raiz

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

Remoção

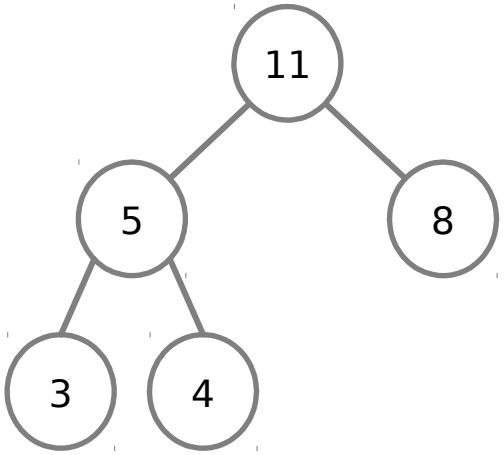
Retira-se sempre a raiz

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

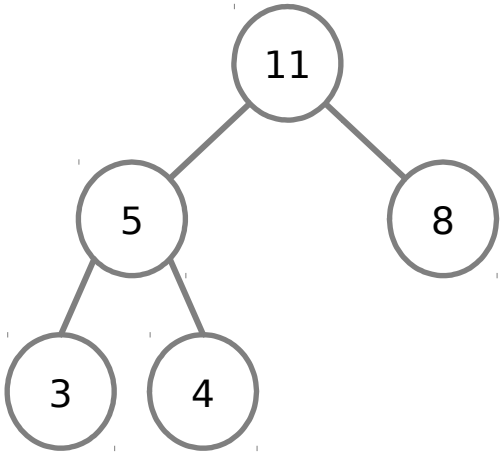
Compare o elemento com seus filhos:

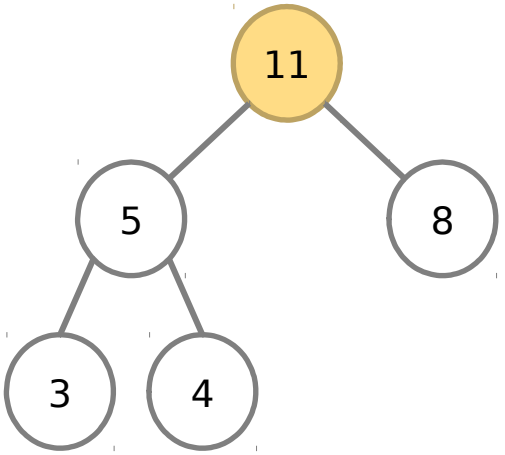
- se estiver em ordem, a remoção terminou
- se não estiver, troque com o maior filho e repita o processo até terminar ou chegar a uma folha

Exemplo de Remoção

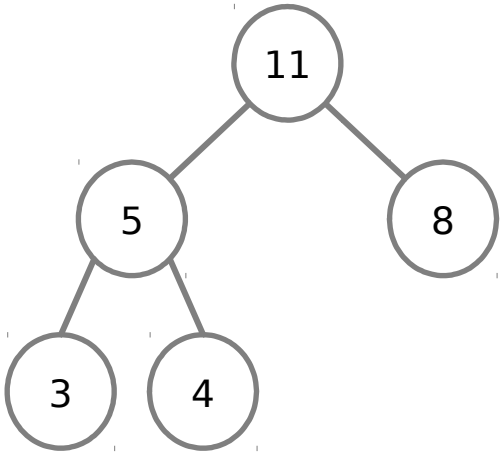


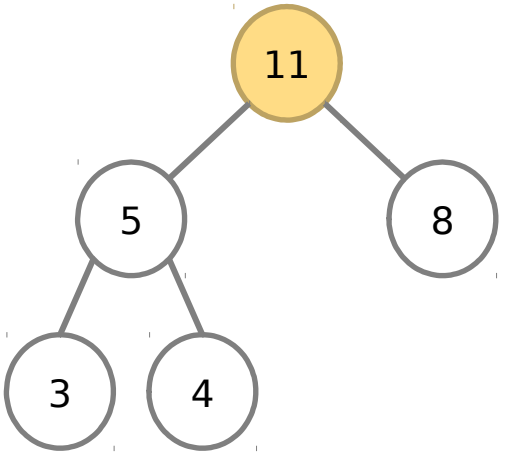
Exemplo de Remoção

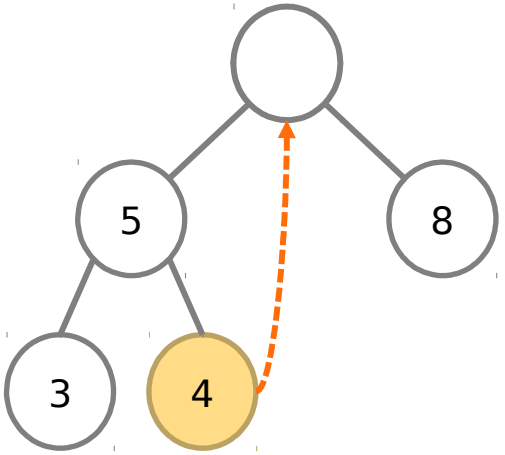




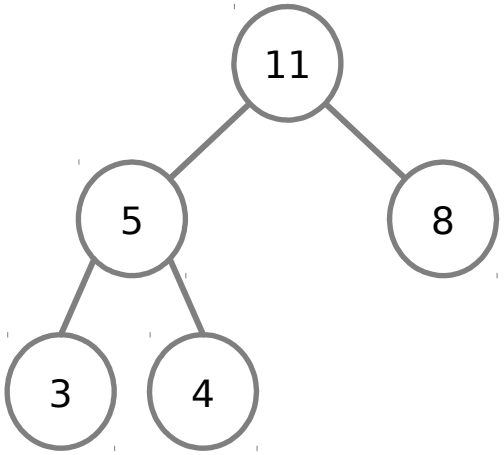
Exemplo de Remoção

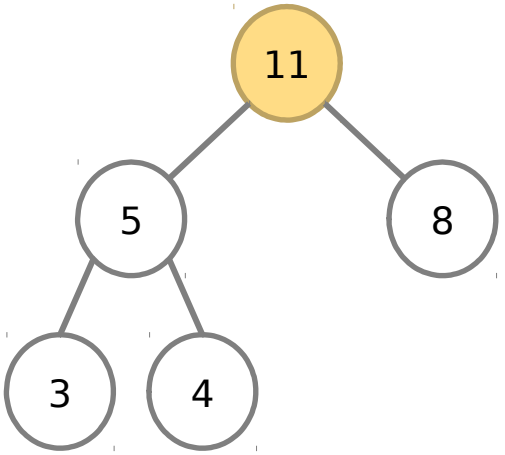


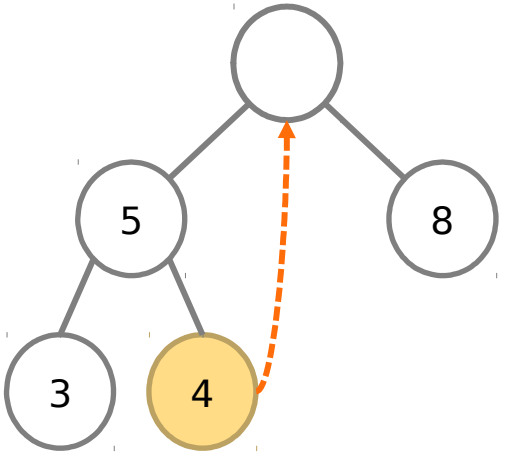


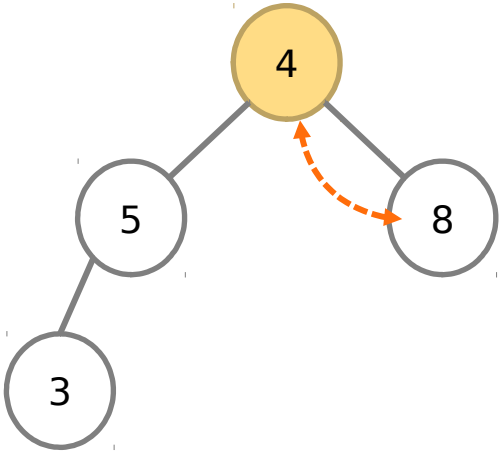


Exemplo de Remoção

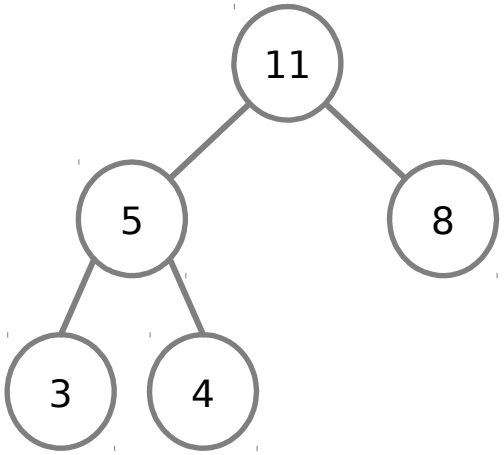


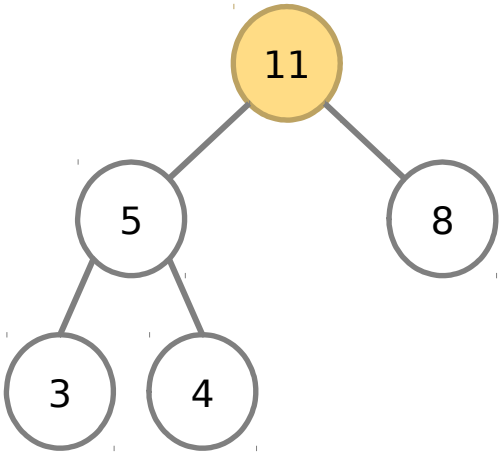


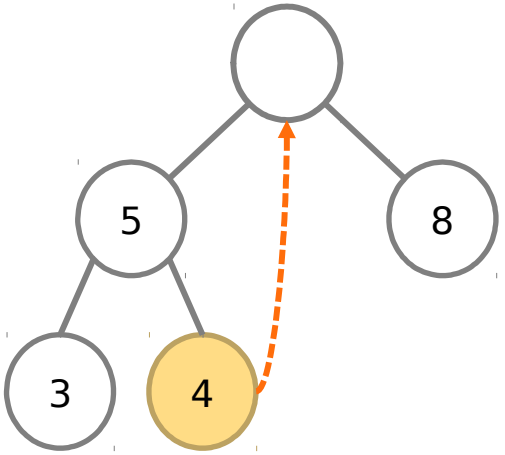


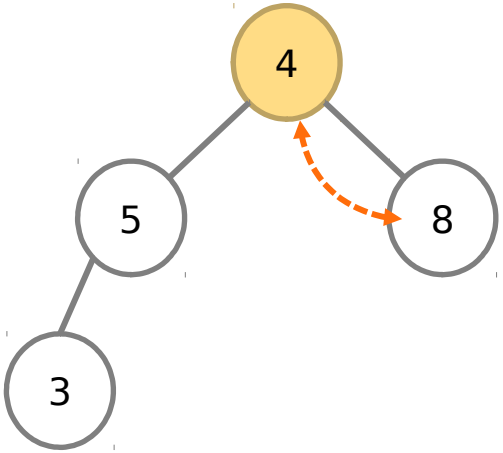


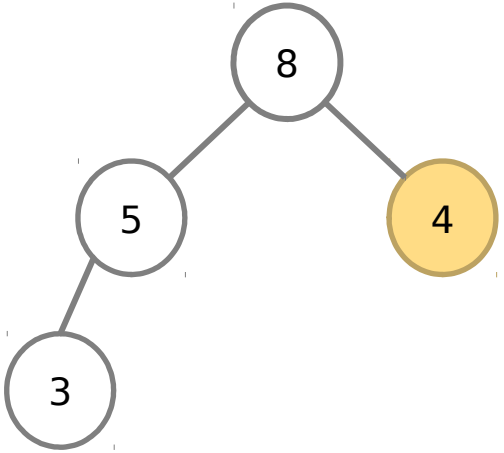
Exemplo de Remoção











Eficiência

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	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

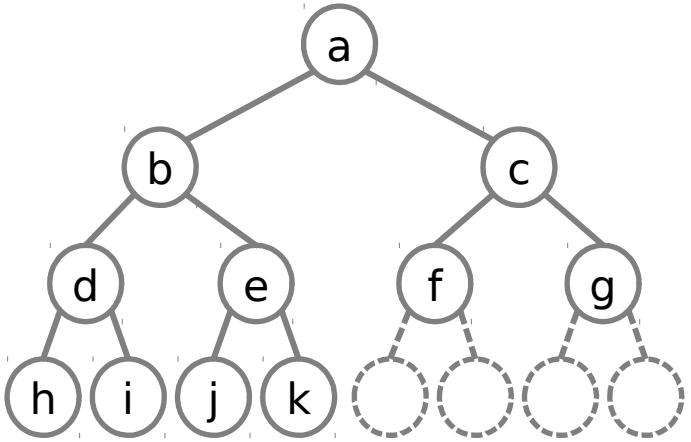
Implementando Heap com Vetor

Podemos representar uma árvore binária (completa) com um vetor:

→ filho esquerdo de i: **2*i +1**

→ filho direito de i: **2*i + 2**

→ pai de i: **(i-1)/2**

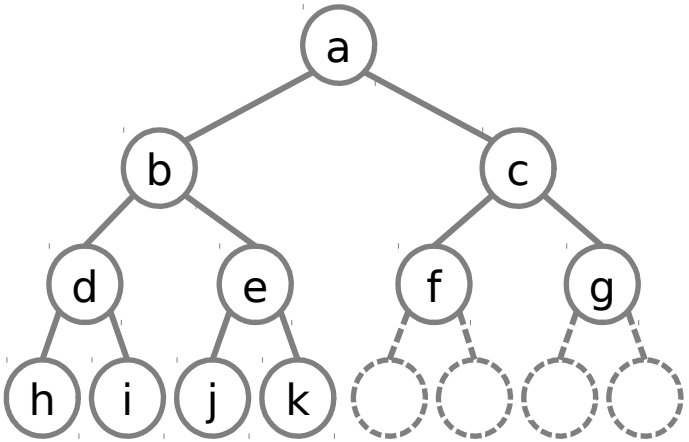


índice	0	1	2	3	4	5	6	7	8	9	10	...
nó	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**...**
nível	0	1		2				3				

Implementando Heap com Vetor

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

→ número de nós de uma árvore cheia de altura *h*



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				

Interface Heap

```
typedef struct heap Heap;

Heap* heap_cria(int max);

void  heap_insere(Heap* heap, int prioridade, dados* d);

dados *heap_remove(Heap* heap);

void heap_libera(Heap *heap);
```

Implementando Módulo Heap

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

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

Implementando Inserção

```
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");
}
```

Subindo o elemento

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

Implementando Remoção

```
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->prioridade, 0, heap->pos);
    return topo;
  }
  else {
     printf("Heap VAZIO!");
     return -1;
  }
}
```

Descendo o elemento

```
static void corrige_abaixo(int *prios, int atual, int tam){
  int pai = atual;
  int filho_esq, filho_dir, filho;
  while (2*pai+1 < tam){
    filho_esq=2*pai+1;
    filho_dir=2*pai+2;
    if (filho_dir >= tam) filho_dir=filho_esq;
    if (prios[filho_esq] > prios[filho_dir])
        filho = filho_esq;
    else
        filho = filho_dir;
    if (prios[pai] < prios[filho])
            troca(pai,filho,prios);
    else
            break;
    pai = filho;
  }
}
```

Construção de um Heap

Algoritmo ingênuo: inserção um-a-um

- complexidade: O(n log n)

Observe que:

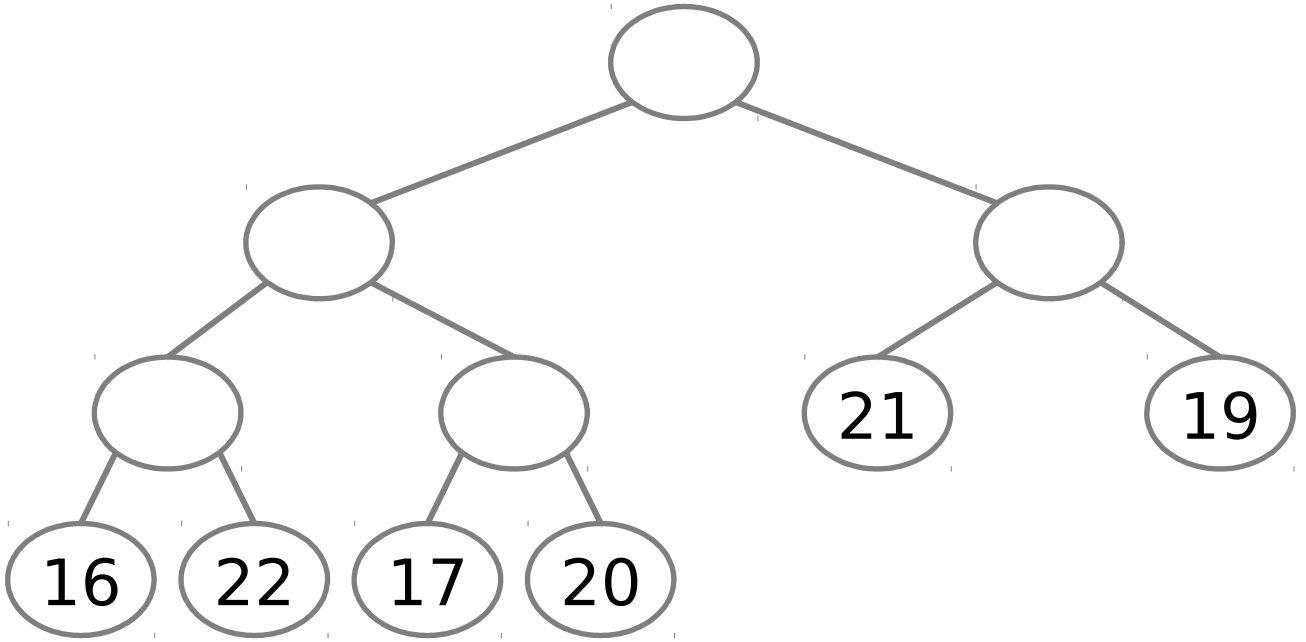
- as folhas (elementos n/2 ... n-1) já estão ordenadas, pois não têm descendentes
- se acertarmos os nós internos (elementos 0 ... n/2 - 1) em relação a seus descendentes, o *heap* estará pronto
- devemos trabalhar de trás para frente, a partir de n/2 - 1, pois as propriedades do *heap* estão corretas nos níveis mais baixos

Exemplo

Lista de 11 prioridades

→ 21, 19, 16, 22, 17, 20, 23, 12, 34, 15, 60

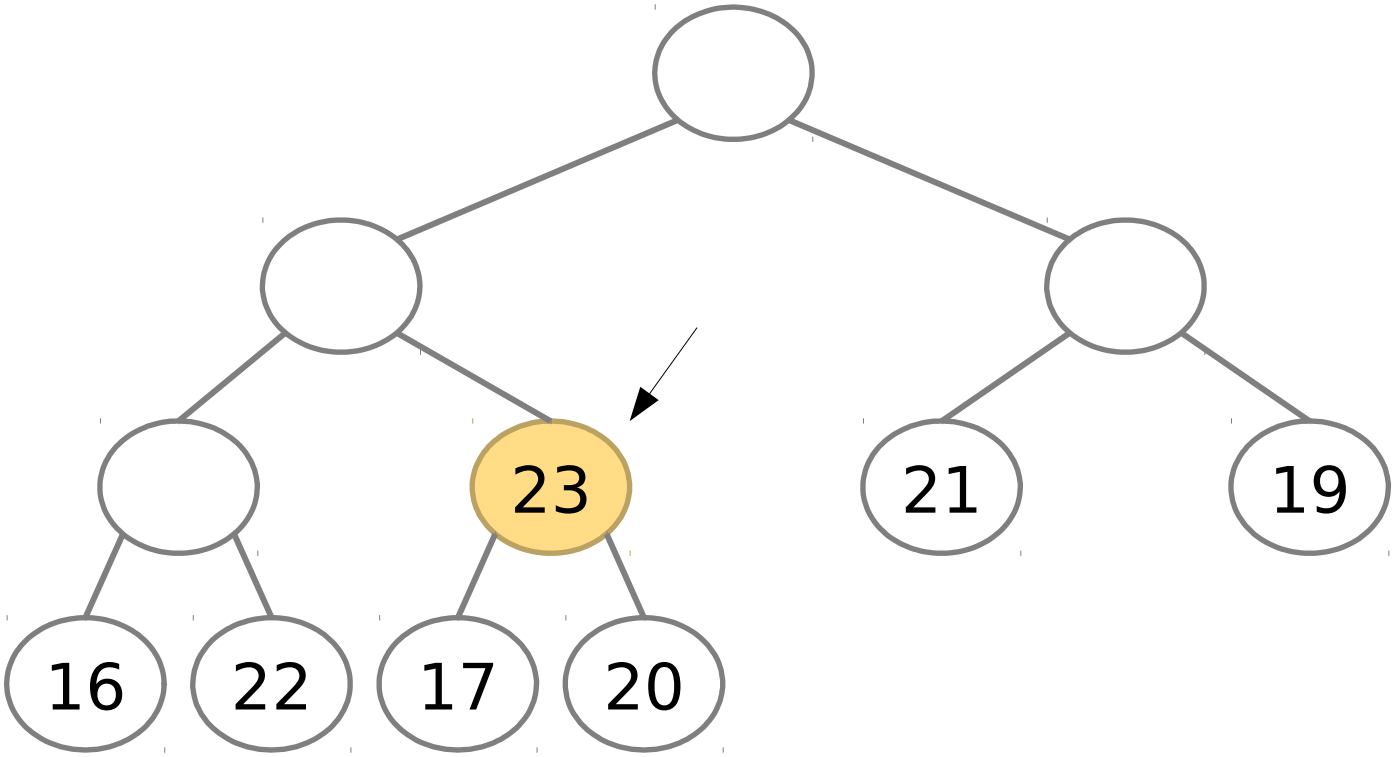
Inserimos elementos nas posições de n/2 até n-1 (folhas)



Exemplo

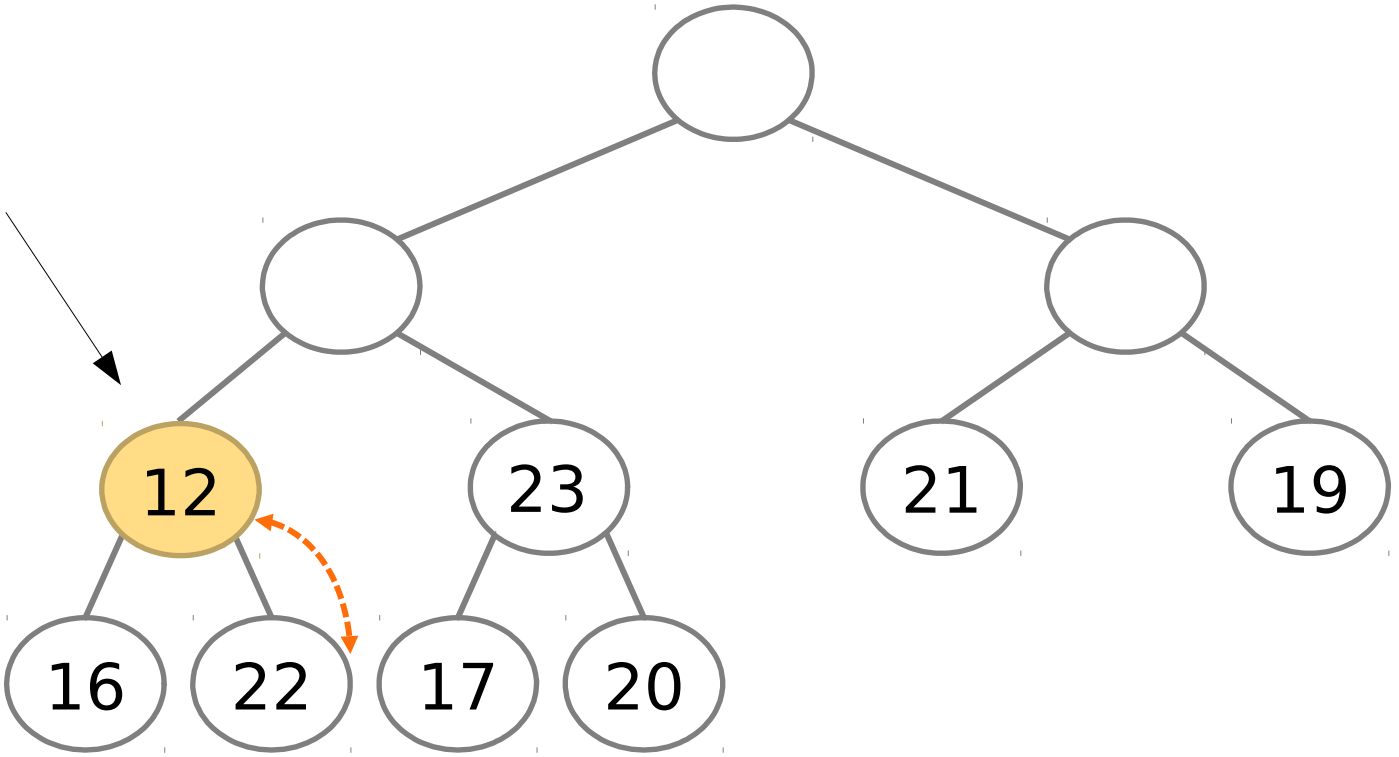
Falta inserir: **23**, 12, 34, 15, 60

Começamos de n/2 – 1



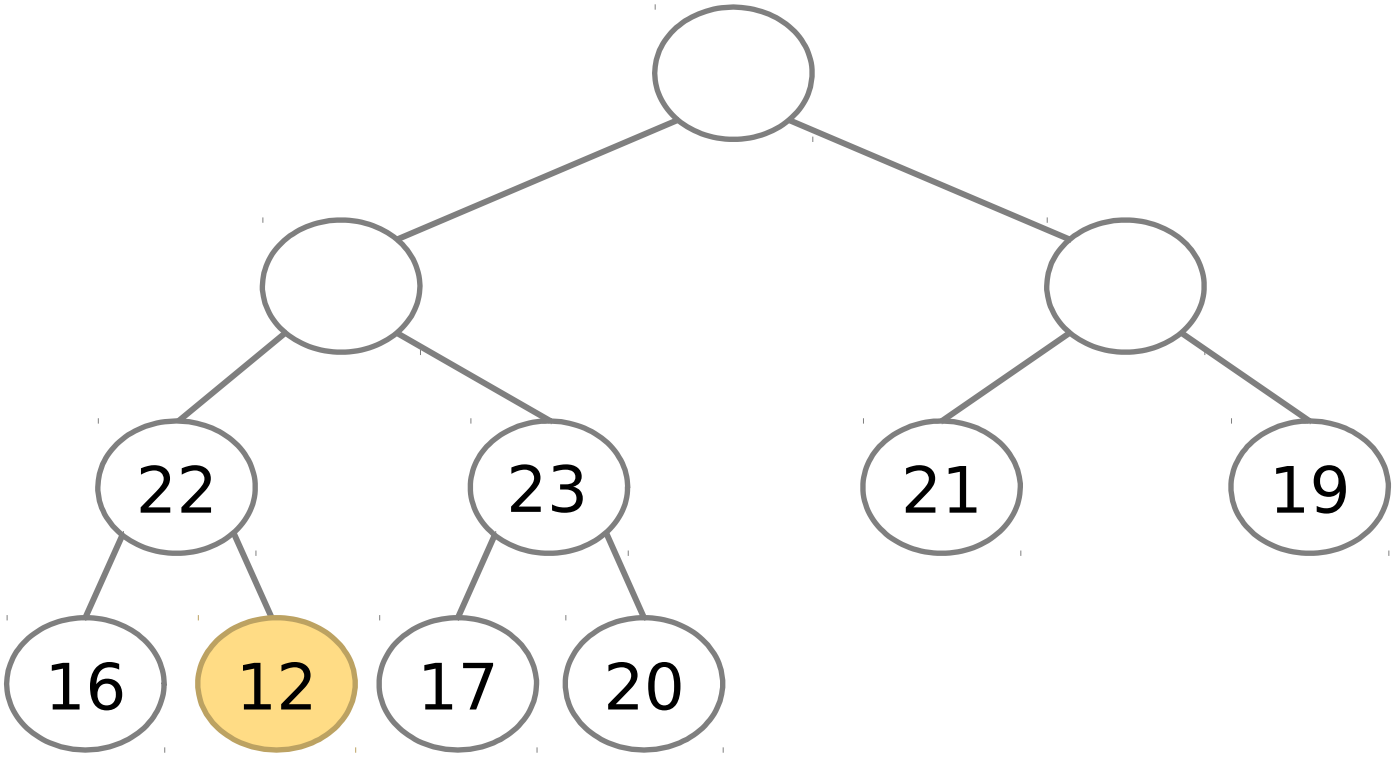
Exemplo

Falta inserir: **12**, 34, 15, 60



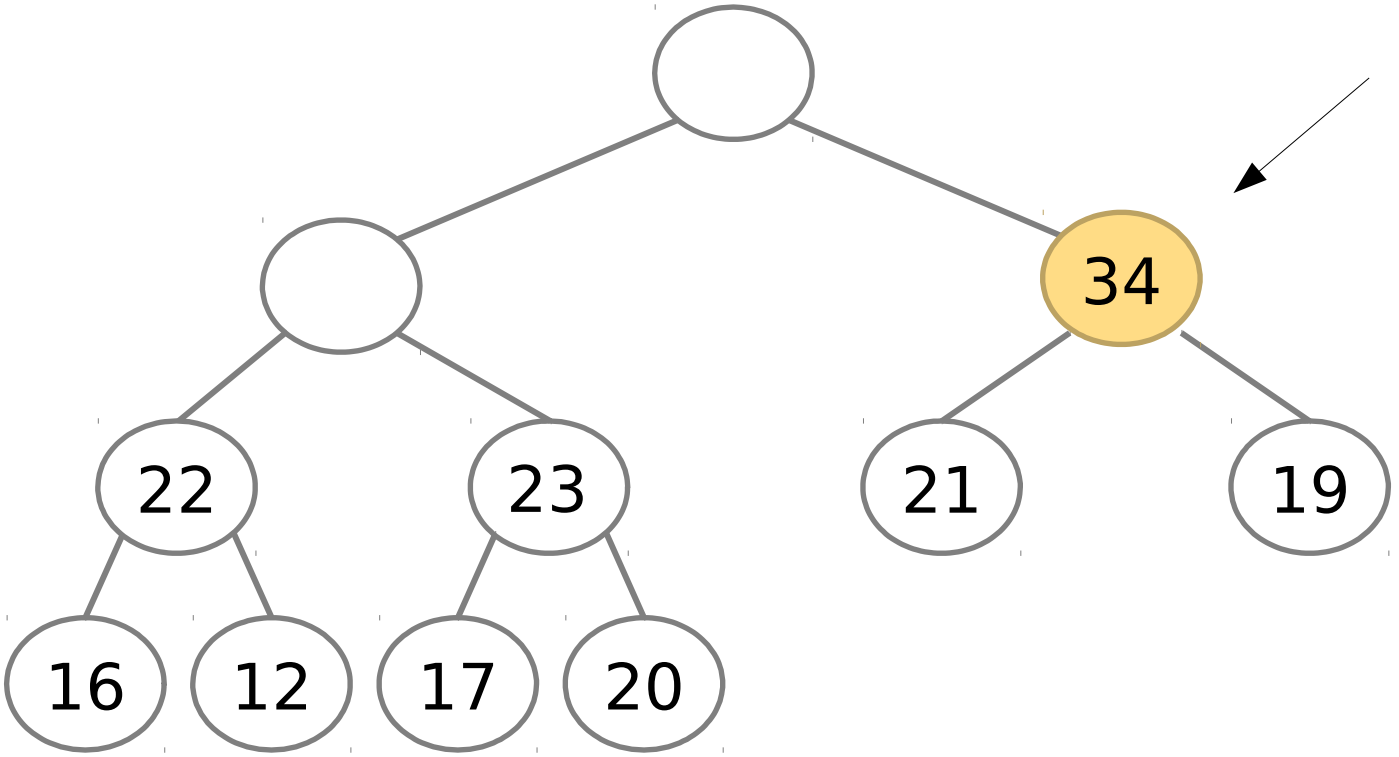
Exemplo

Falta inserir: 34, 15, 60



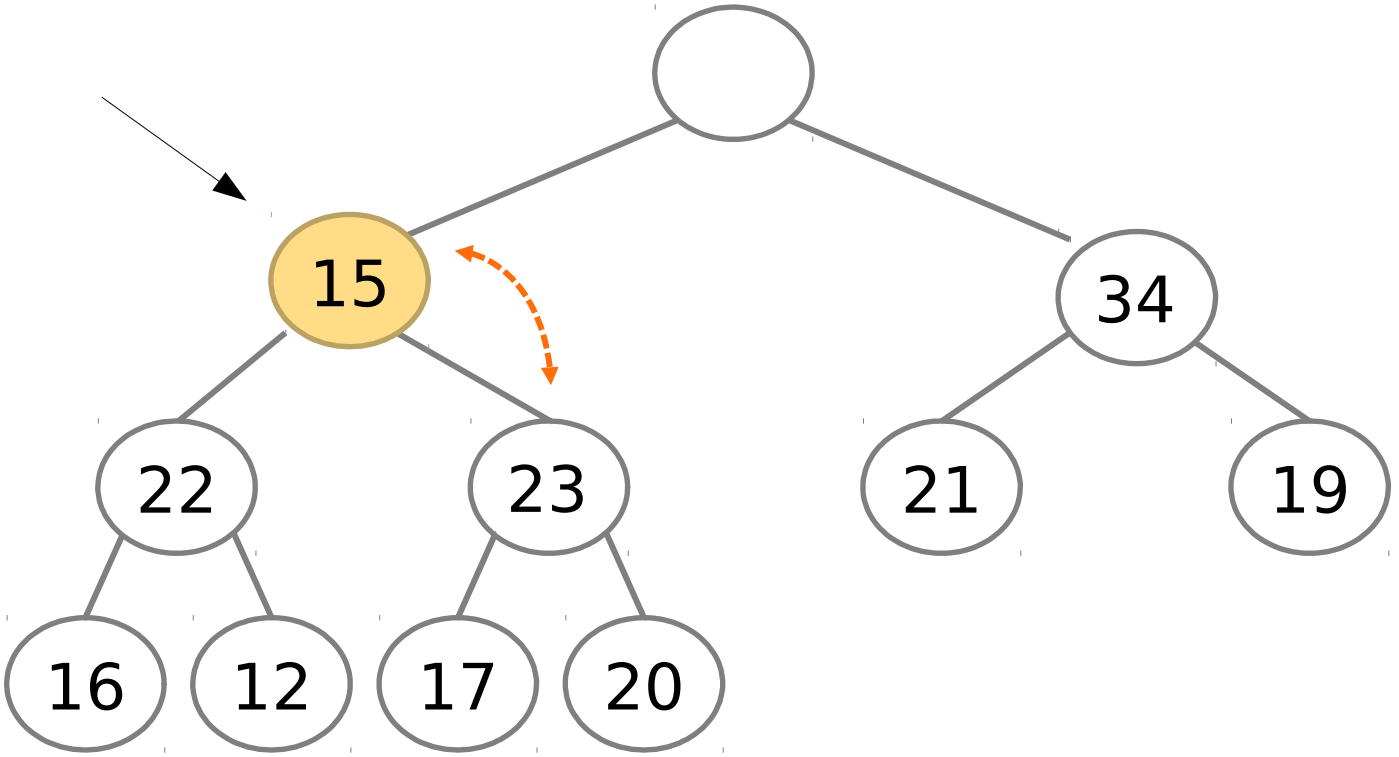
Exemplo

Falta inserir: **34**, 15, 60



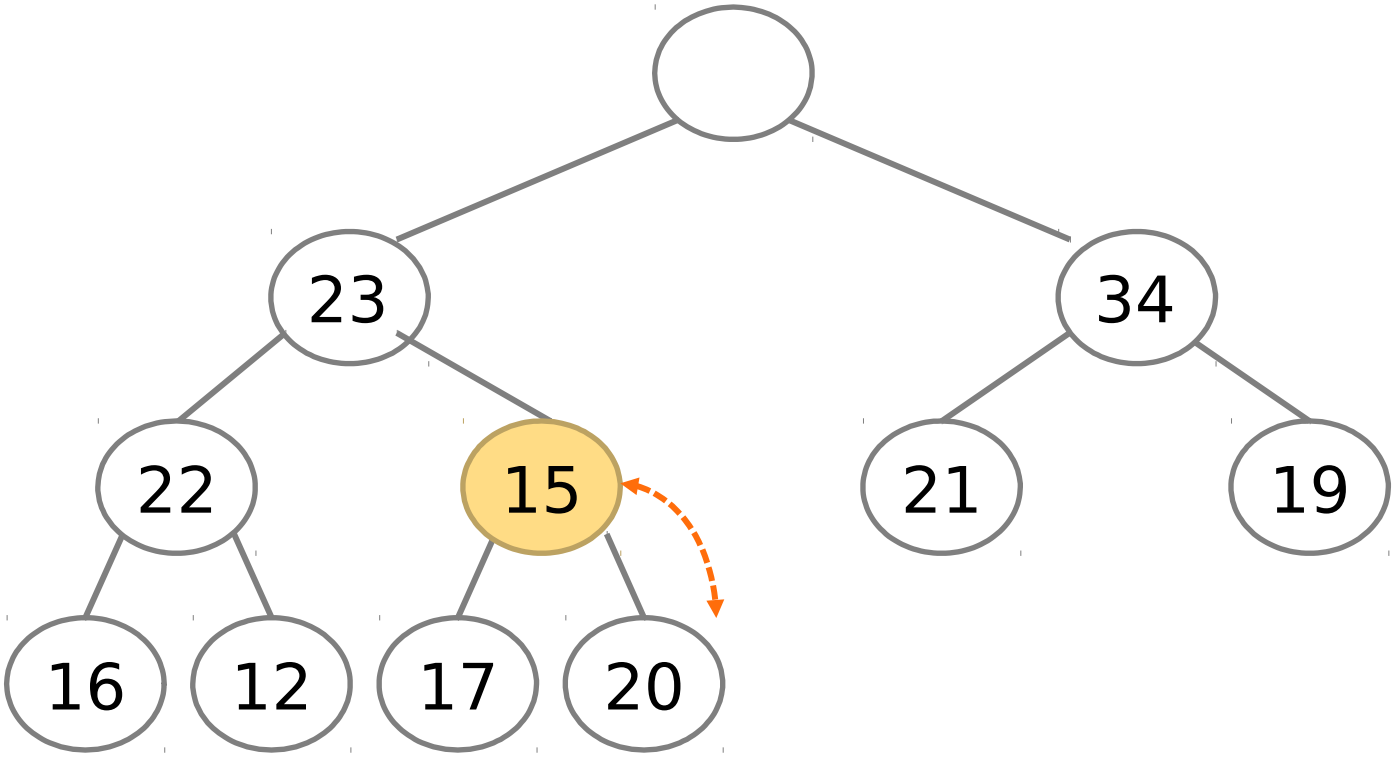
Exemplo

Falta inserir: **15**, 60



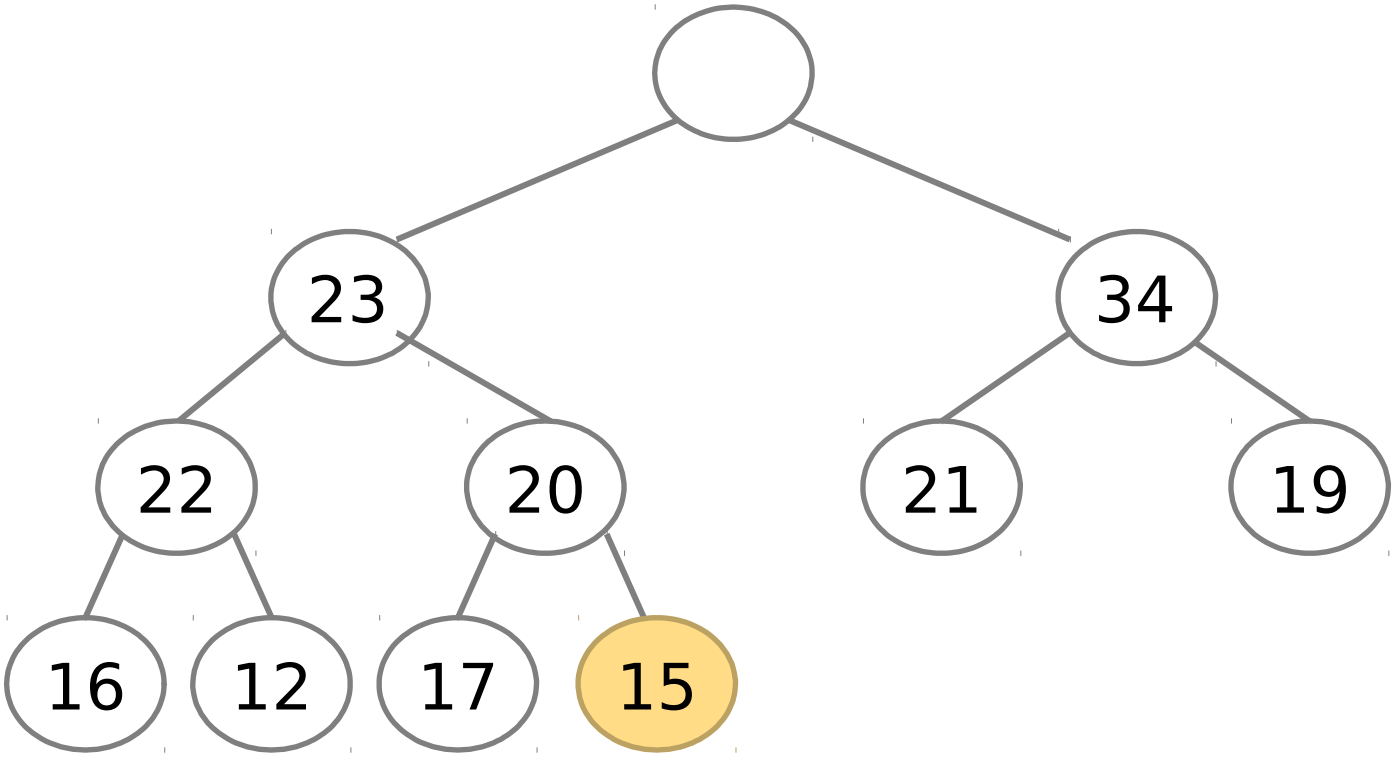
Exemplo

Falta inserir: 60



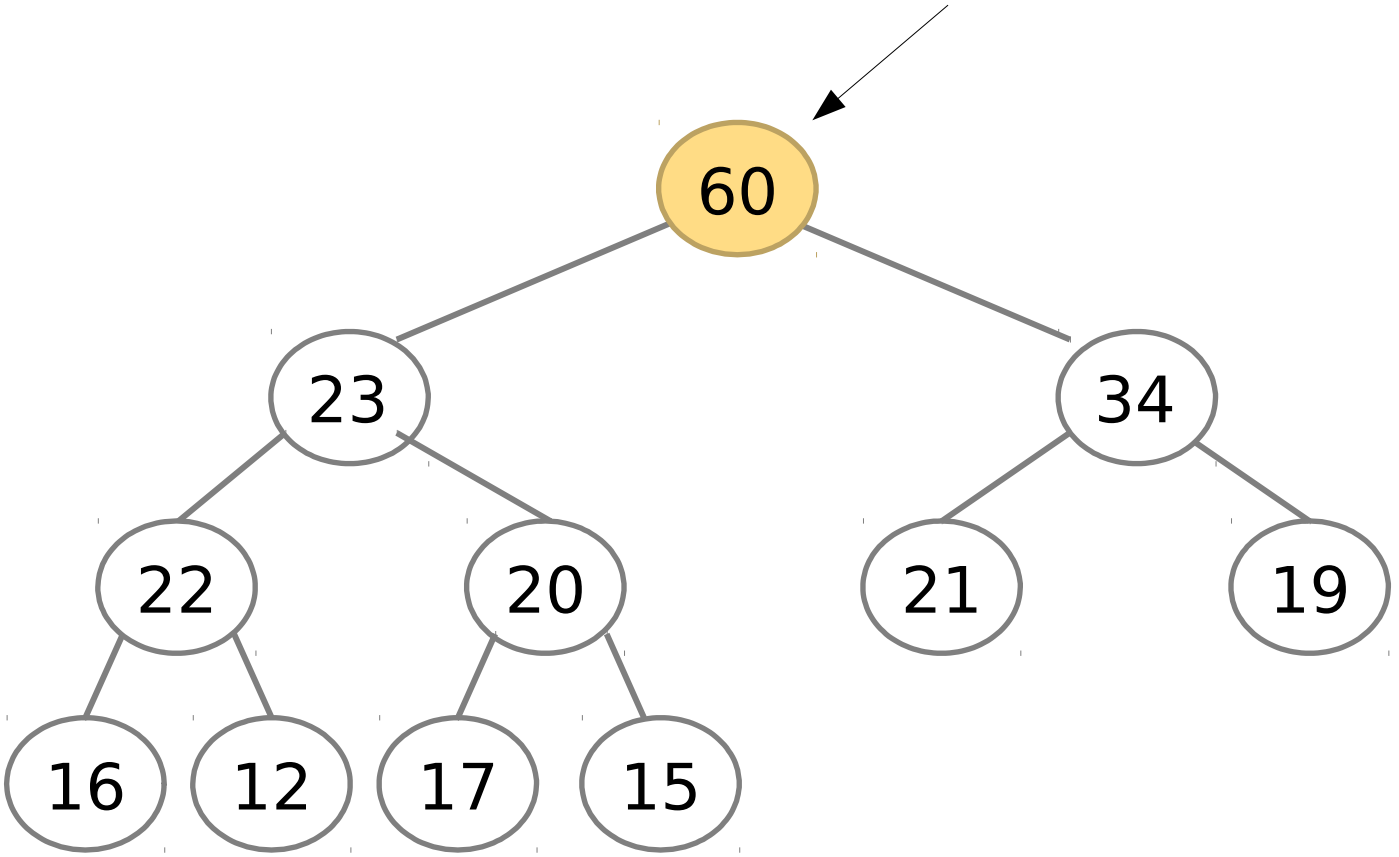
Exemplo

Falta inserir: 60



Exemplo

Falta inserir: **60**



Complexidade da Construção

Se a árvore está cheia:

→ número de nós → $n = 2^{h+1} - 1$, 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 da Construção

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

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

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

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

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

Heapsort

Ordenação com complexidade **O(n log n)**, mesmo no pior caso

Passo 1: construção do *heap* → O(n)

Passo 2: para os n elementos do *heap* → O(n log n)

→ remova o elemento do topo

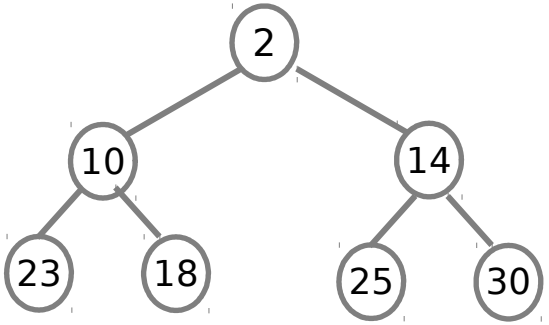
→ salve o elemento no final do vetor de *heap*

Construção intuitiva:

→ à medida que os elementos vão sendo colocados no final, o *heap* diminui de tamanho

→ ao final, o vetor está em ordem decrescente (ou crescente)

Ordenação do Vetor (**min** *heap*)



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	

DEPARTAMENTO DE INFORMÁTICA
PUC·RIO