Estruturas de Dados
Heap e Torneio

Prof. Joaquim Uchôa
Profa. Juliana Greghi
Prof. Renato Ramos

- Conceitos Básicos
- Heaps em Arranjos
- Retirada e Inserção de Elementos
- Implementação
- Torneio

Conceitos Básicos

Comentários Iniciais

Um heap binário é uma estrutura de dados usada para implementação de filas de prioridade. Em geral, é implementado em um arranjo, mas de forma a ser acessado como uma árvore binária.

Uma árvore binária é um tipo especial de árvore, em que cada nó tem zero, um ou dois nós filhos.

Uma árvore é um tipo especial de grafo.

Comentários Iniciais

Um heap bin … usada
para implem … . Em
geral, é im … de forma
a ser acess …

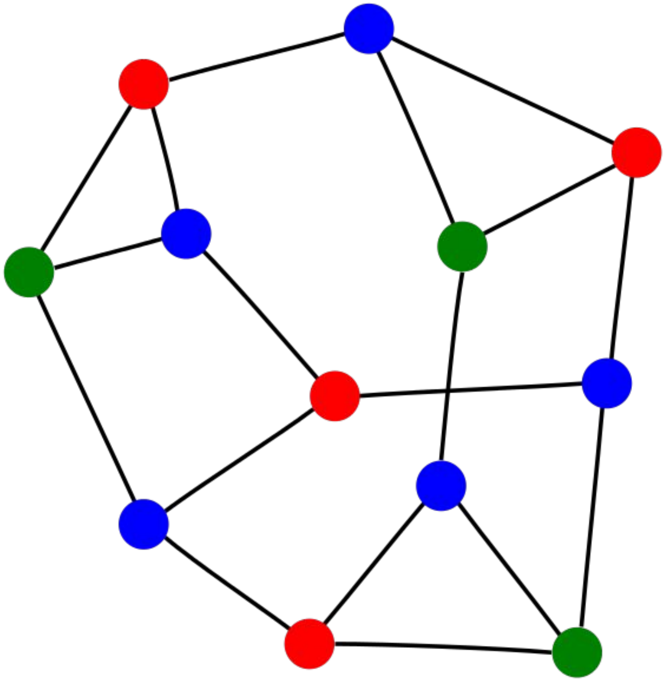
Uma árvore … árvore,
em que cada … filhos.

Uma árvore …

PERAÍ,
PERAÍ!!!

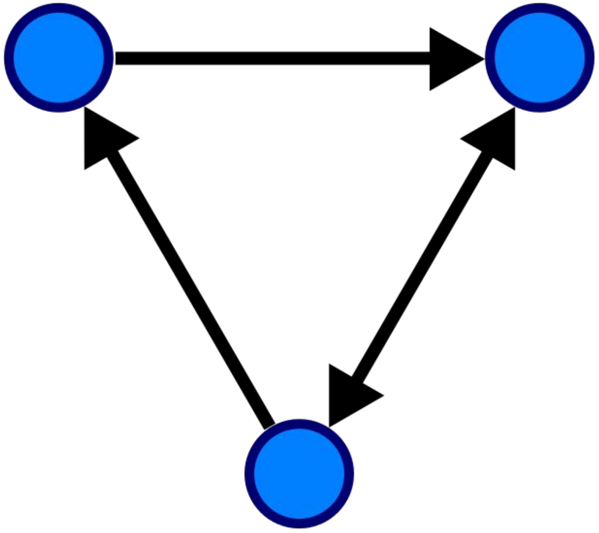
Conceitos Iniciais - Grafos I

Um grafo G(V,E) é formado por um conjunto de vértices (V) e um subconjunto de pares de V, chamados arestas (E). As arestas ligam justamente os vértices do grafo.



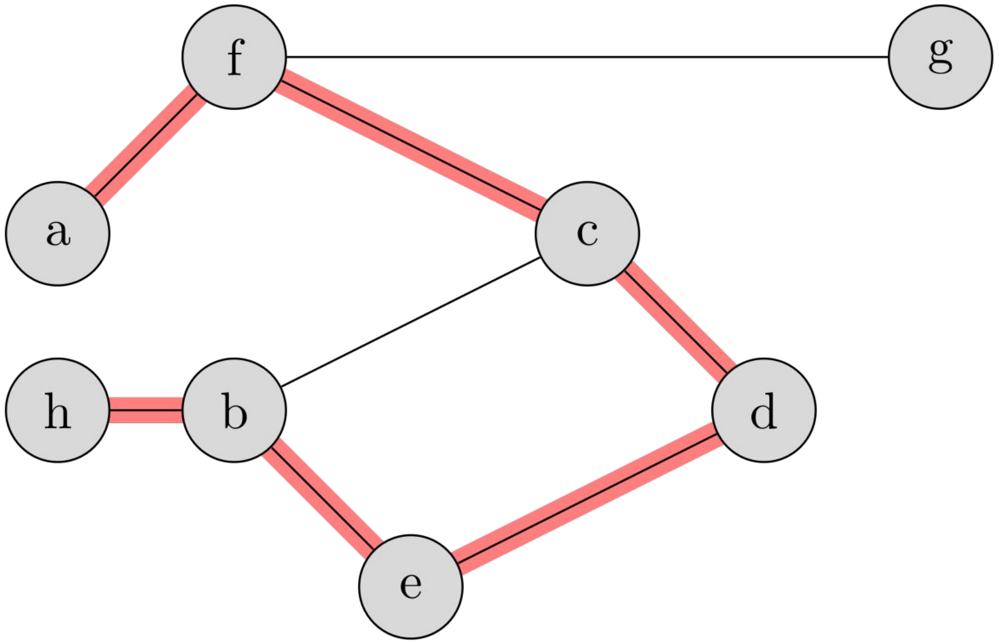
Conceitos Iniciais - Grafos II

Se os pares em E forem ordenados, então as arestas possuem direção o grafo é dito ser ordenado ou orientado (ou dígrafo).



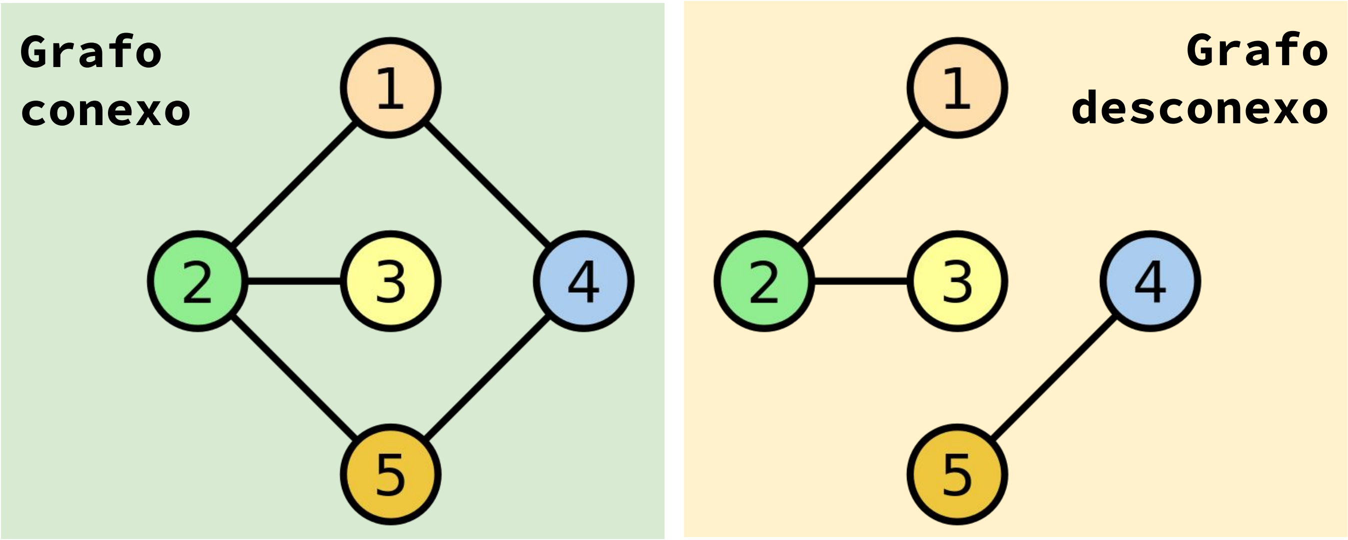
Conceitos Iniciais - Grafos IV

Um caminho no grafo é qualquer conjunto de arestas que leva de um nó a outro. Ao lado temos um caminho que liga os nós a a h.



Conceitos Iniciais - Grafos IV

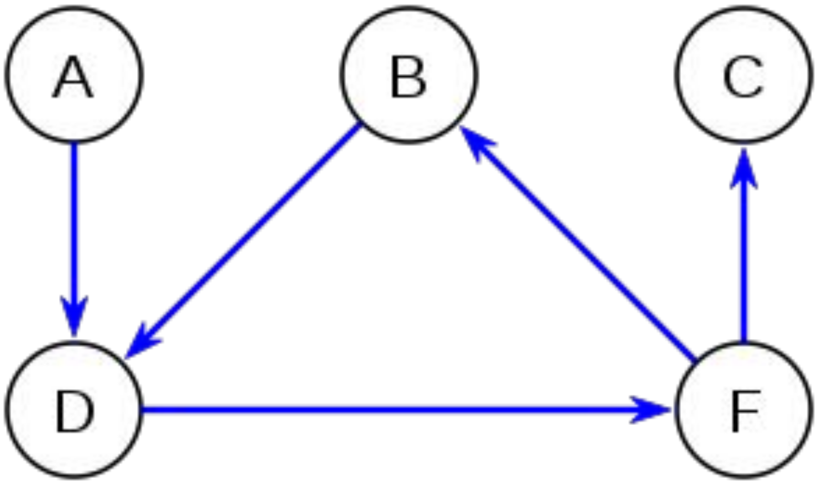
Um grafo é conexo se existe um caminho entre dois nós diferentes do mesmo grafo.



Conceitos Iniciais - Grafos IV

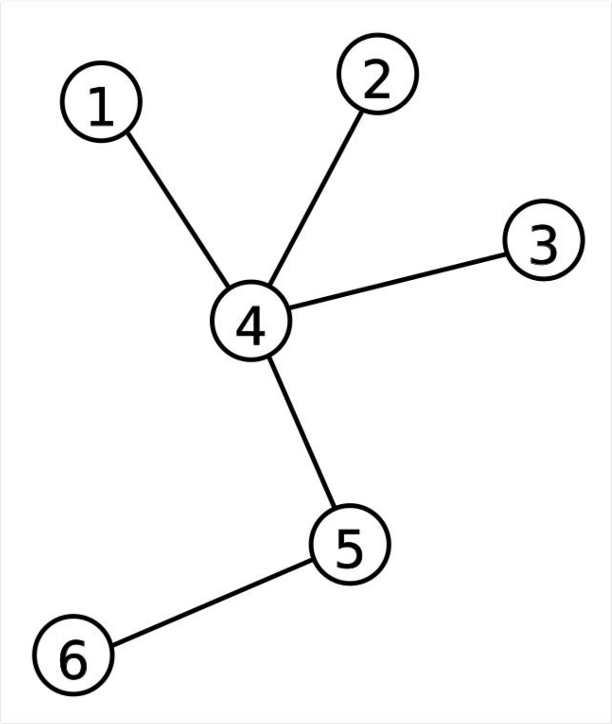
Um grafo é cíclico se existe algum nó que possui um caminho que volta a ele passando por outros nós.

Isso ocorre ao lado com os nós B, D e F.



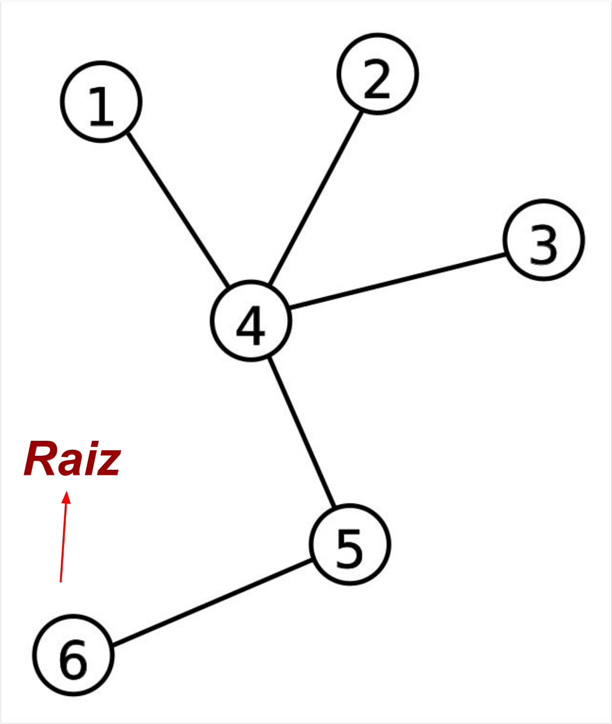
Conceitos Iniciais - Árvore I

Uma árvore é um grafo conexo (existe caminho entre quaisquer dois de seus vértices) e acíclico (não possui ciclos - ou seja, não existe dois caminhos entre vértices)



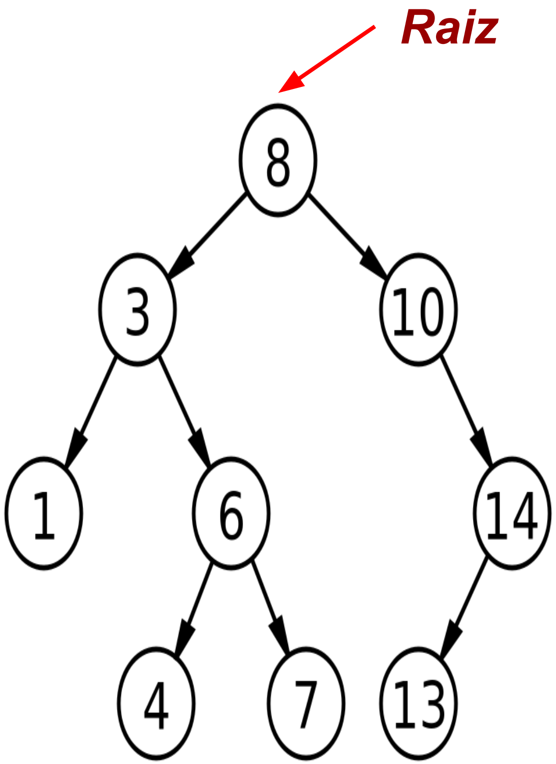
Conceitos Iniciais - Árvore II

Em geral, nas aplicações em Computação, são utilizadas árvores enraizadas. Uma árvore é enraizada se um vértice é escolhido como especial, chamado raiz.



Conceitos Iniciais - Árvore III

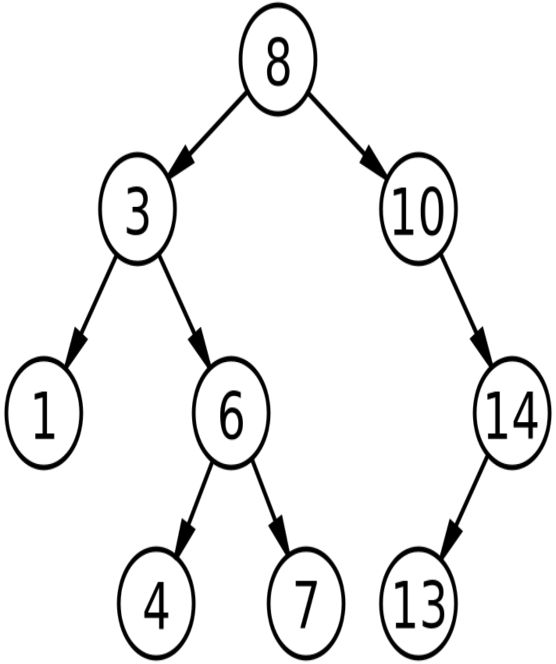
Como em geral a raiz tem grande importância (seja por ser o primeiro elemento ou o mais valioso), é comum representar árvores com a raiz na parte superior.



Conceitos Iniciais - Árvore IV

Dado um nó específico de uma árvore enraizada, seus **filhos** são os nós que estão ligados diretamente a ele em um nível inferior. No exemplo ao lado, os filhos do nó 6 são 4 e 7.

Nós sem filhos são denominados **folhas**, caso do 1, 4, 7 e 13.



Conceitos Iniciais - Árvores V

Em uma árvore enraizada, tem-se para cada nó:

➡ **Grau:** número de nós filhos.

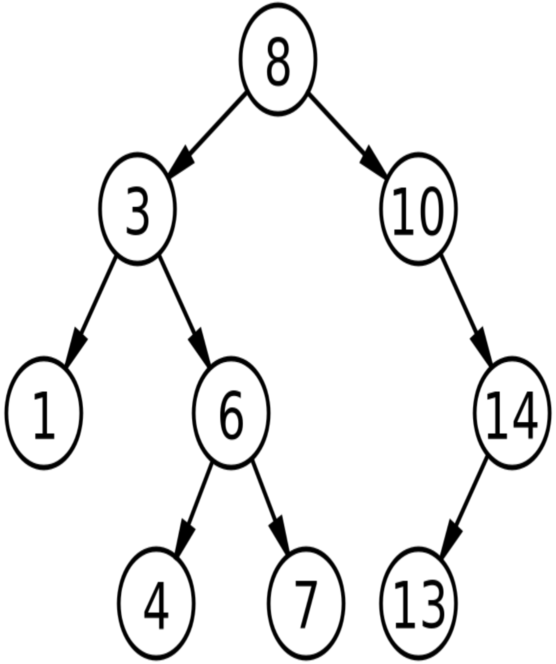
➡ **Profundidade:** distância (número de arestas) até a raiz. Nós com mesma profundidade estão em um mesmo **nível**.

➡ **Altura da árvore:** maior profundidade.

➡ **Grau da árvore:** maior grau de seus nós.

Exemplo de Árvore Enraizada

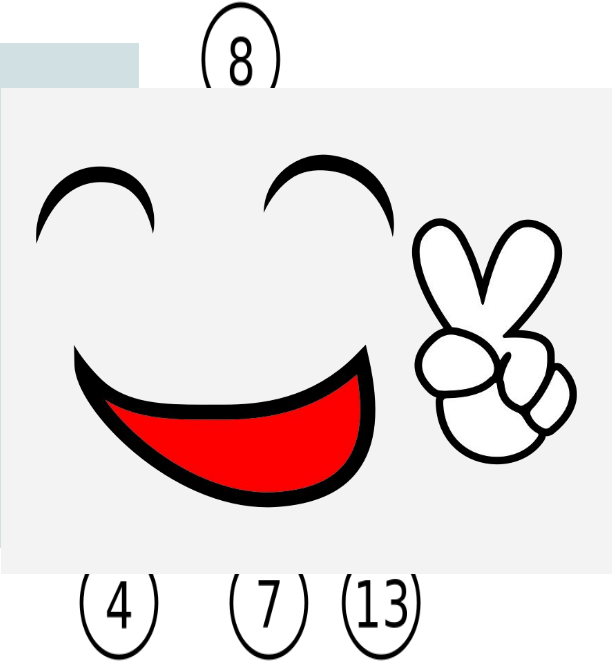
Na árvore ao lado, em que 8 é a raiz, os nós 1, 6 e 14 estão no mesmo nível, profundidade 2. O nó 14 possui grau 1 e o nó 6 grau 2. O nó 1 possui grau 0. Os nós 1, 4, 7 e 13 são folhas. O grau da árvore é 2 (árvore binária) e sua altura é 3.



Exemplo de Árvore Enraizada

Na árvore ao lado, em que 8
é a
está
pro
poss
2. O
nós
O grau da árvore é 2 (árvore
binária) e sua altura é 3.

Um heap é um tipo especial de árvore binária!

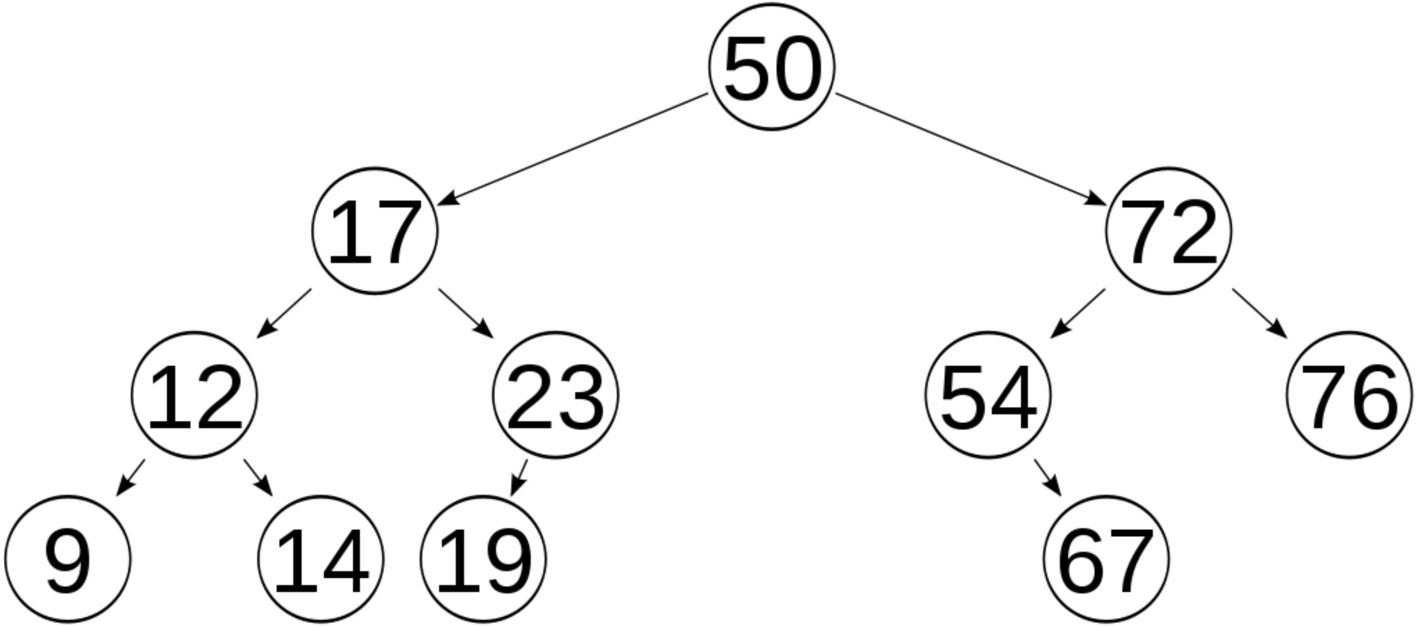


Completando as definições - I

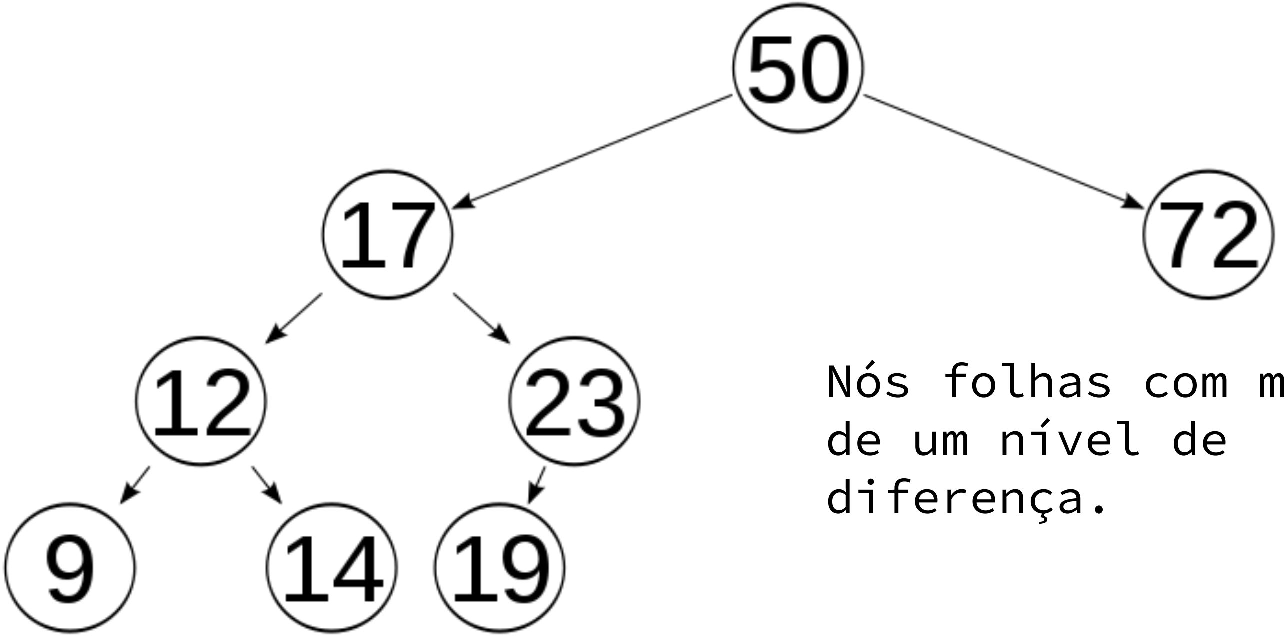
Um heap é uma árvore binária balanceada e completa.

Árvore balanceada: todos os nós folhas estão no mesmo nível ou no máximo com um nível de diferença.

Exemplo de Árvore Balanceada



Exemplo de Árvore Desbalanceada

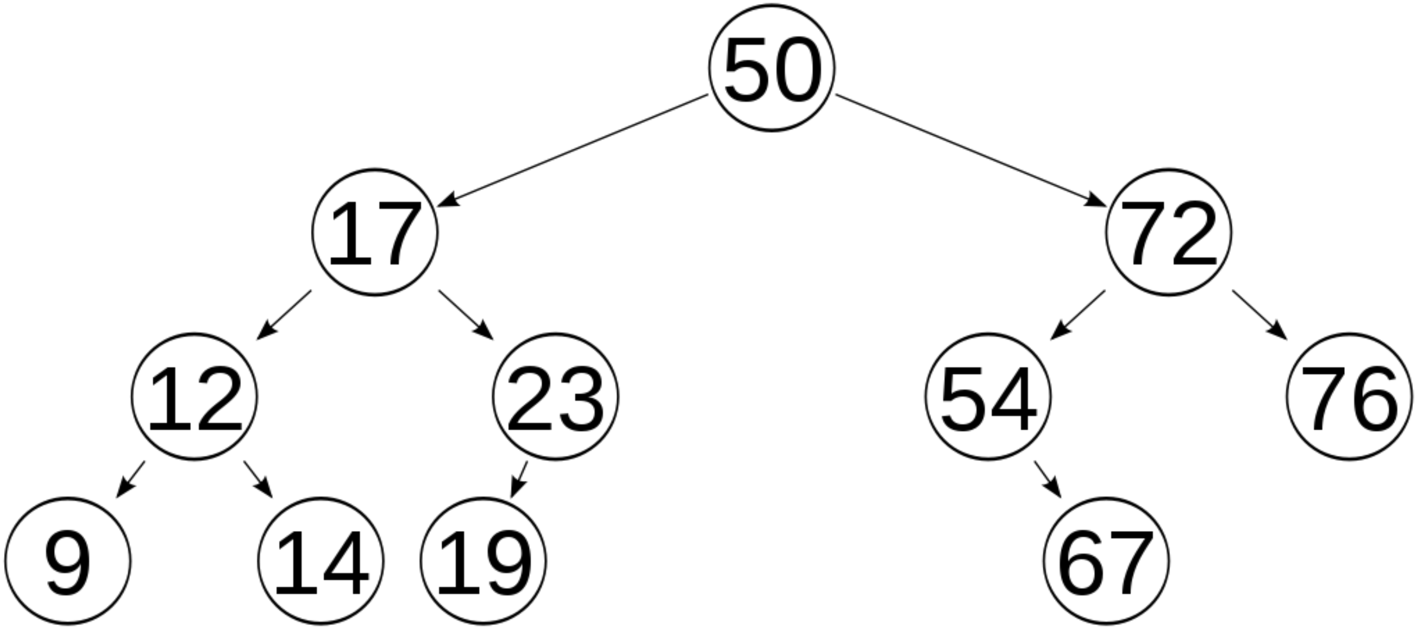


Nós folhas com mais de um nível de diferença.

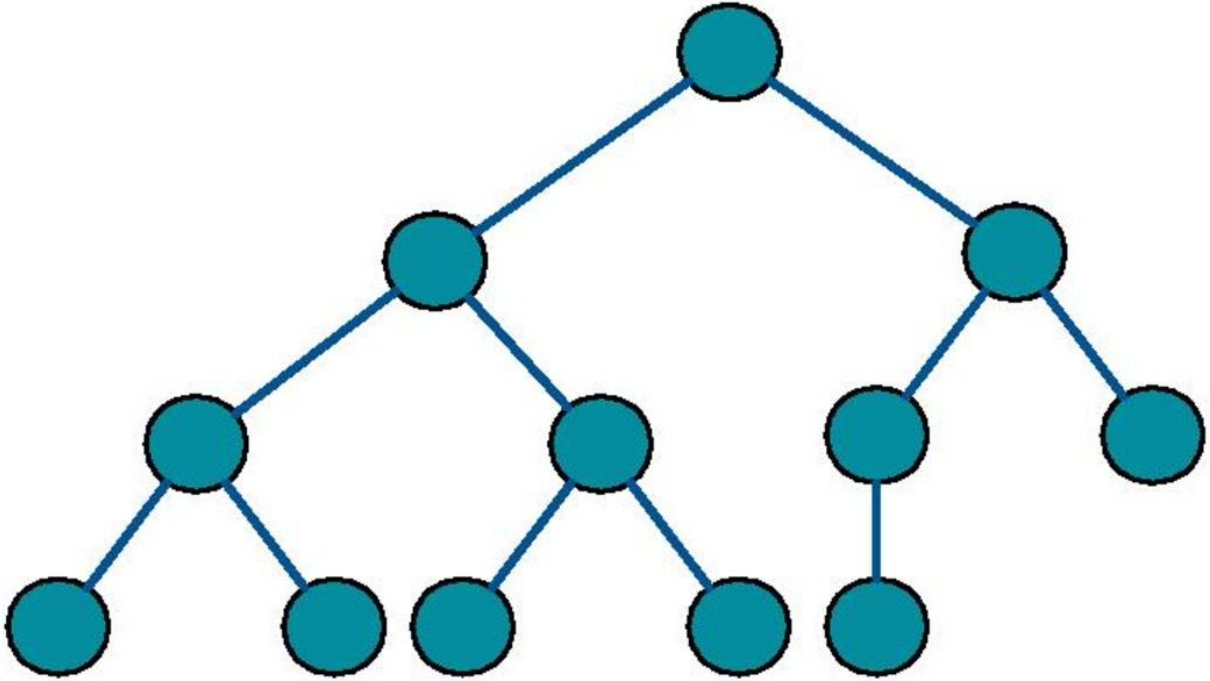
Completando as definições - II

Uma árvore é completa, quando, com exceção das folhas, todos os nós possuem o mesmo grau da árvore. Também é aceitável que o último nó não folha, à direita ou à esquerda, possua grau inferior.

Ex. de árvore não-completa



Ex. de árvore completa



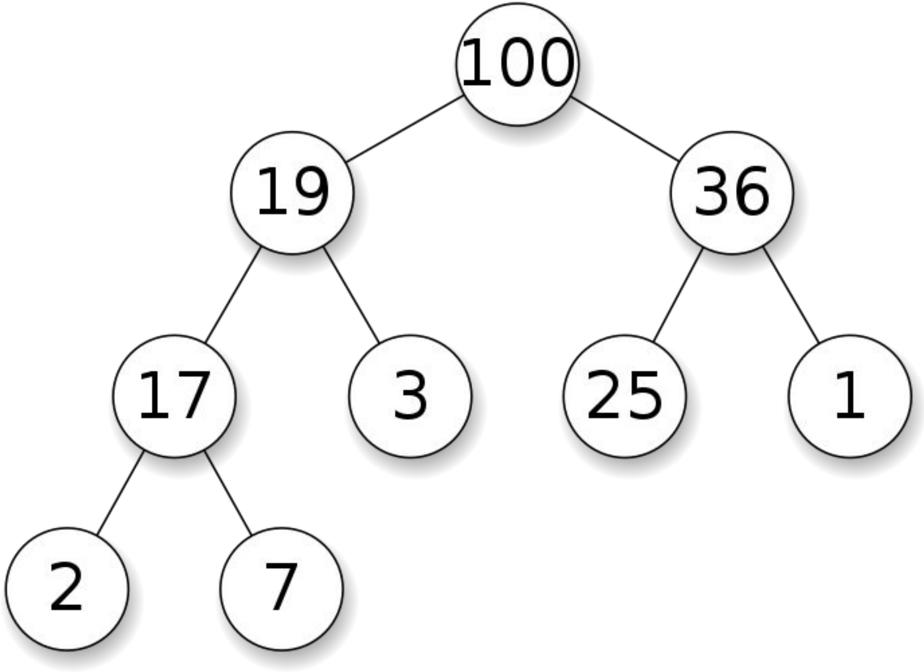
Podemos voltar ao Heap???

Um heap é um tipo específico de árvore binária.

Em um maxheap, cada nó é maior que seus filhos e descendentes. Em um minheap, ocorre exatamente o inverso.

Neste texto, iremos considerar o maxheap, a adaptação para minheap é trivial.

Exemplo de Heap (Maxheap)



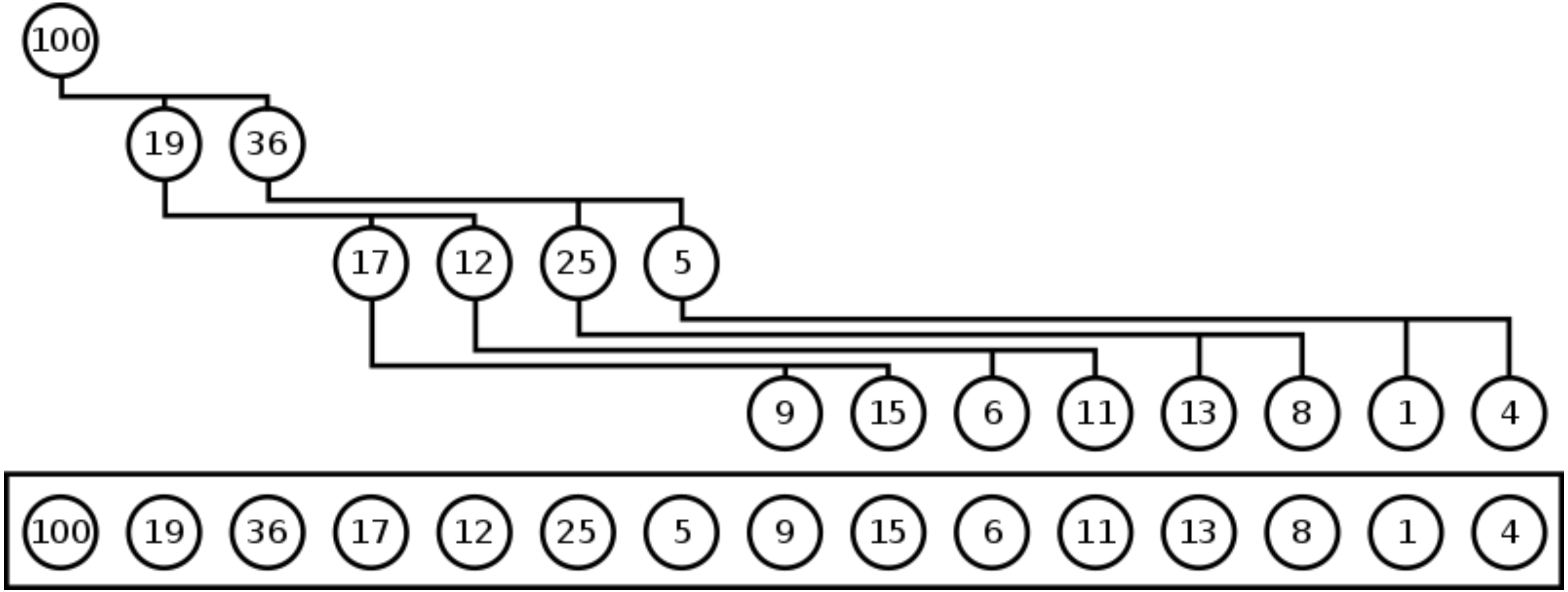
Heaps em Arranjos

Implementação Tradicional de Heaps: Arranjos

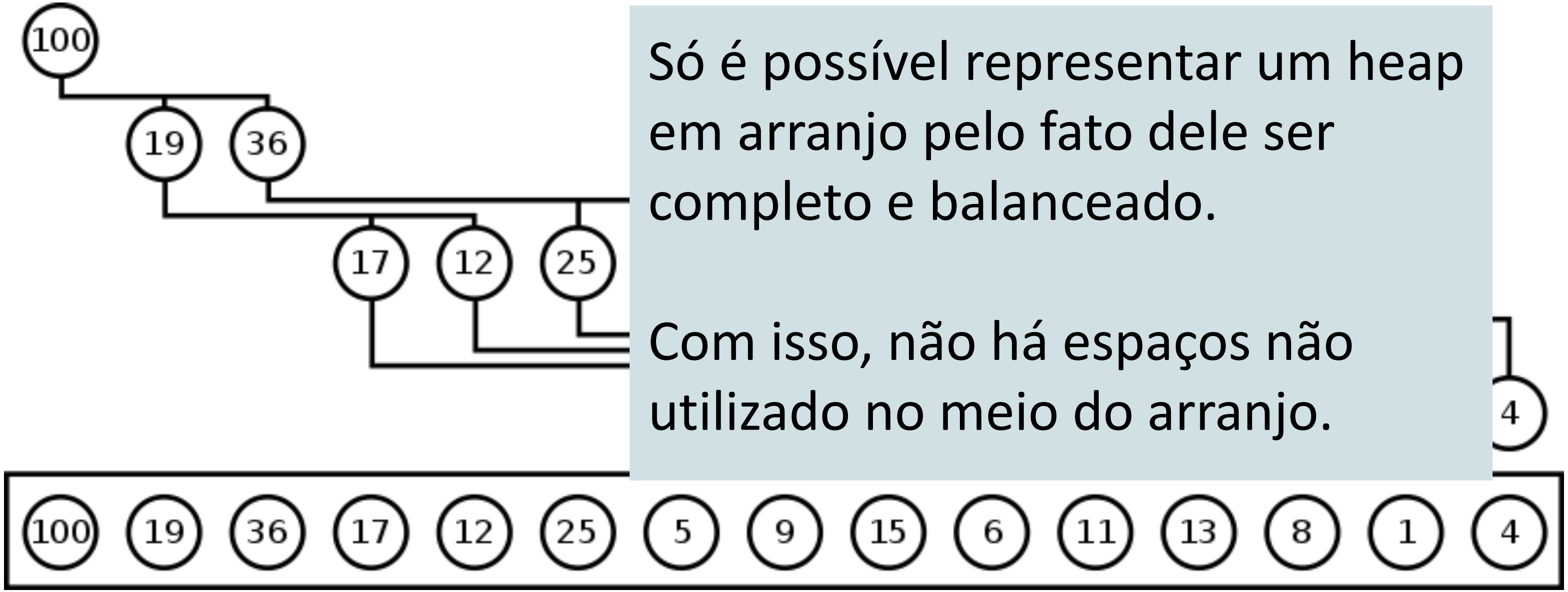
Heaps são tradicionalmente implementados em arranjos, para melhor eficiência das operações.

Mas essa forma de implementação só é adequada (e comum) por conta das características intrínsecas dessa estrutura de dados (ser uma árvore binária completa e balanceada).

Heap em arranjo



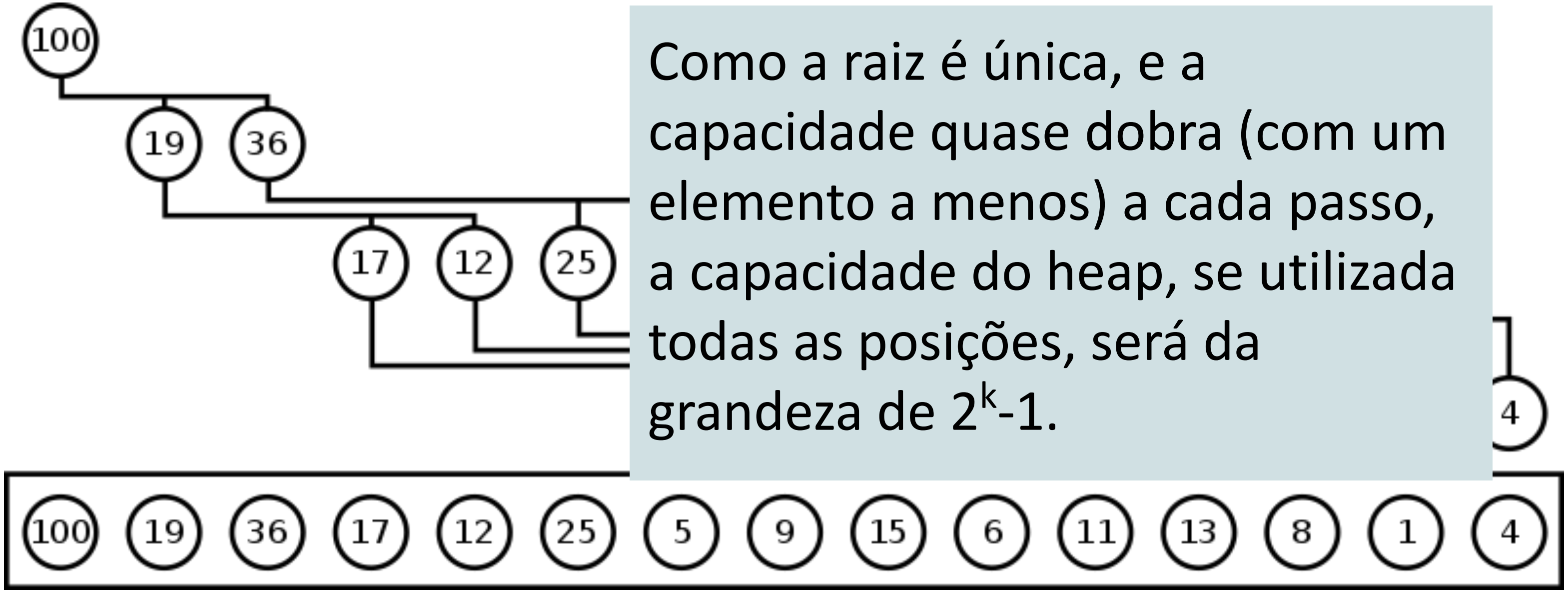
Heap em arranjo



Só é possível representar um heap em arranjo pelo fato dele ser completo e balanceado.

Com isso, não há espaços não utilizado no meio do arranjo.

Heap em arranjo



Como a raiz é única, e a capacidade quase dobra (com um elemento a menos) a cada passo, a capacidade do heap, se utilizada todas as posições, será da grandeza de 2^k-1.

Para que serve isso?

Um heap é uma forma prática e eficiente de implementar filas de prioridade.

Para isso, basta, para retirar o elemento de maior prioridade:

1. remover a raiz
2. substituí-la pelo último elemento
3. reorganizar o heap

Operações básicas em heaps (reorganização)

Corrige descendo: caso um elemento seja menor que um de seus filhos, efetua-se a troca de valores e repete-se o processo no nó filho. **Utilizada na retirada da raiz.**

Corrige subindo: caso um elemento seja maior que seu pai, efetua-se a troca de valores e repete-se o processo no nó pai. **Utilizada na inserção de um novo elemento no heap.**

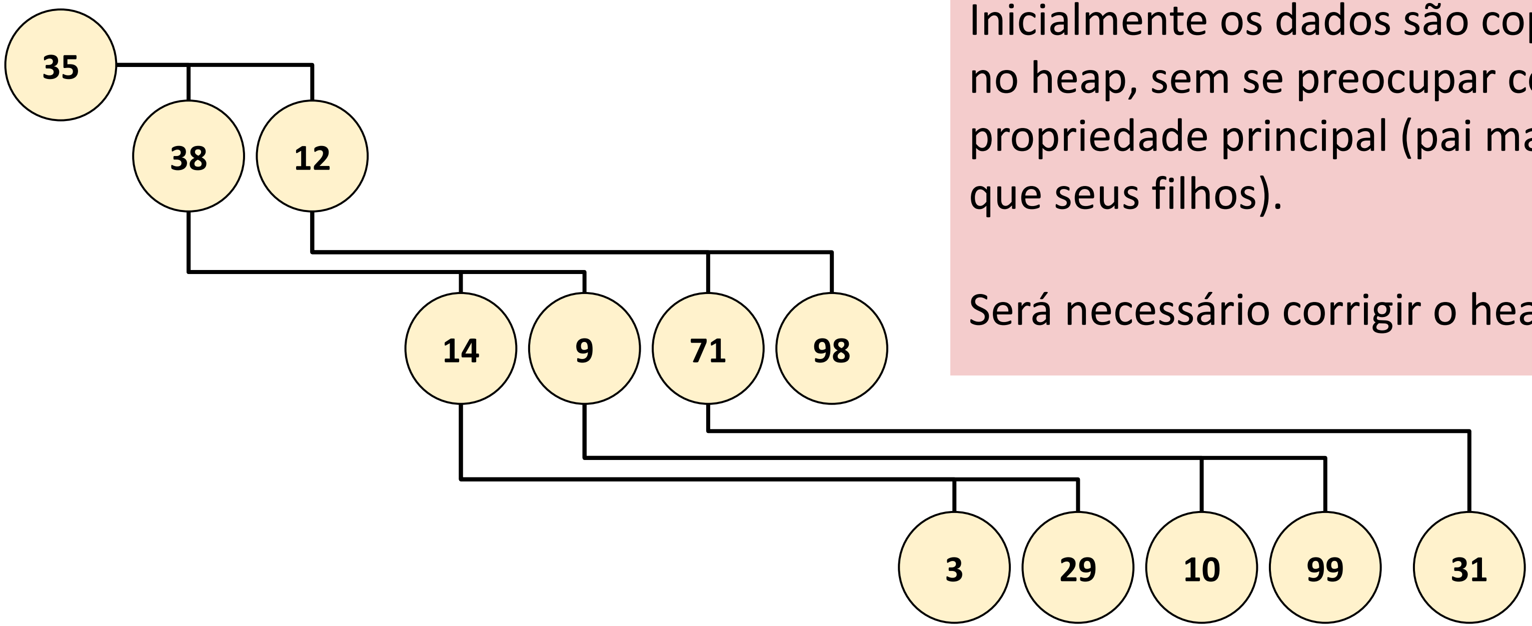
Exemplo 1 - I/XVII

Imagine que pretendamos construir um maxheap a partir dos elementos de um vetor:

35, 38, 12, 14, 9, 71, 98, 3, 29, 10, 99, 31

Precisaríamos de um heap com capacidade teórica para 15 elementos, caso queiramos usar todas as posições.

Exemplo 1 - II/XVII

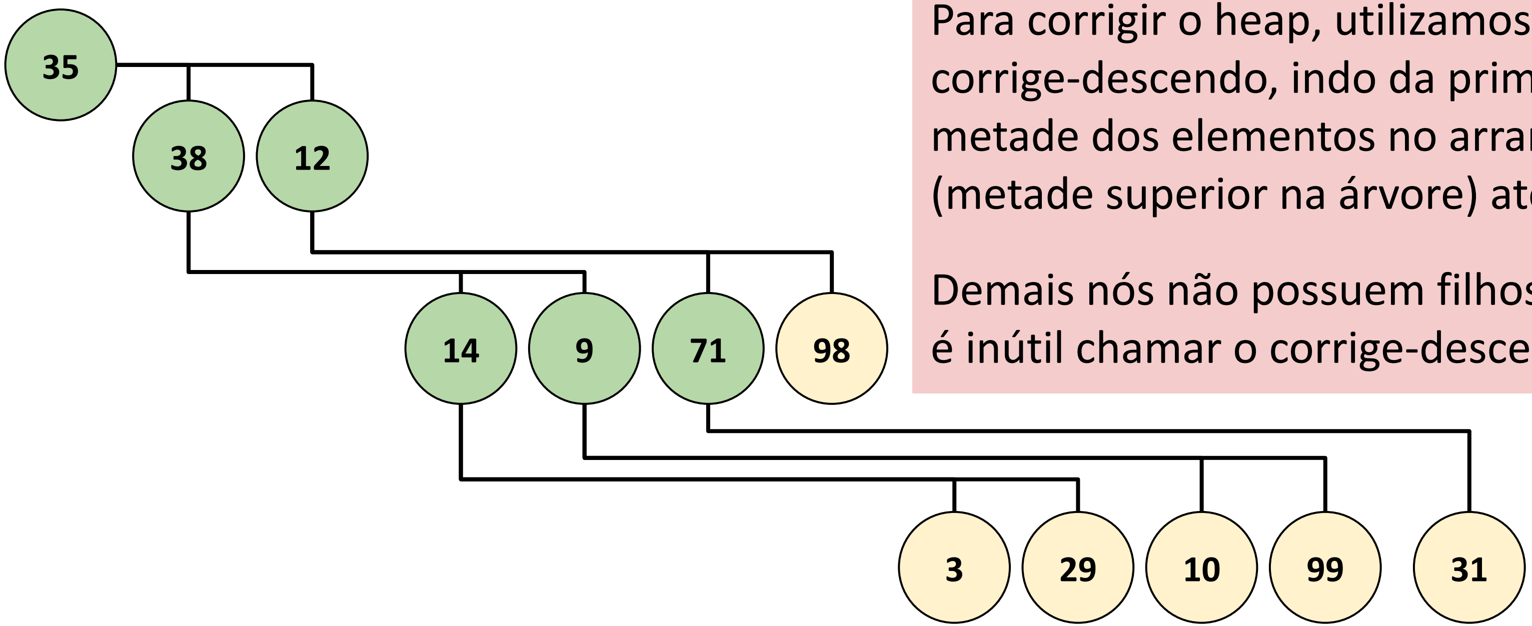


35	38	12	14	9	71	98	3	29	10	99	31

Inicialmente os dados são copiados no heap, sem se preocupar com a propriedade principal (pai maior que seus filhos).

Será necessário corrigir o heap.

Exemplo 1 - III/XVII

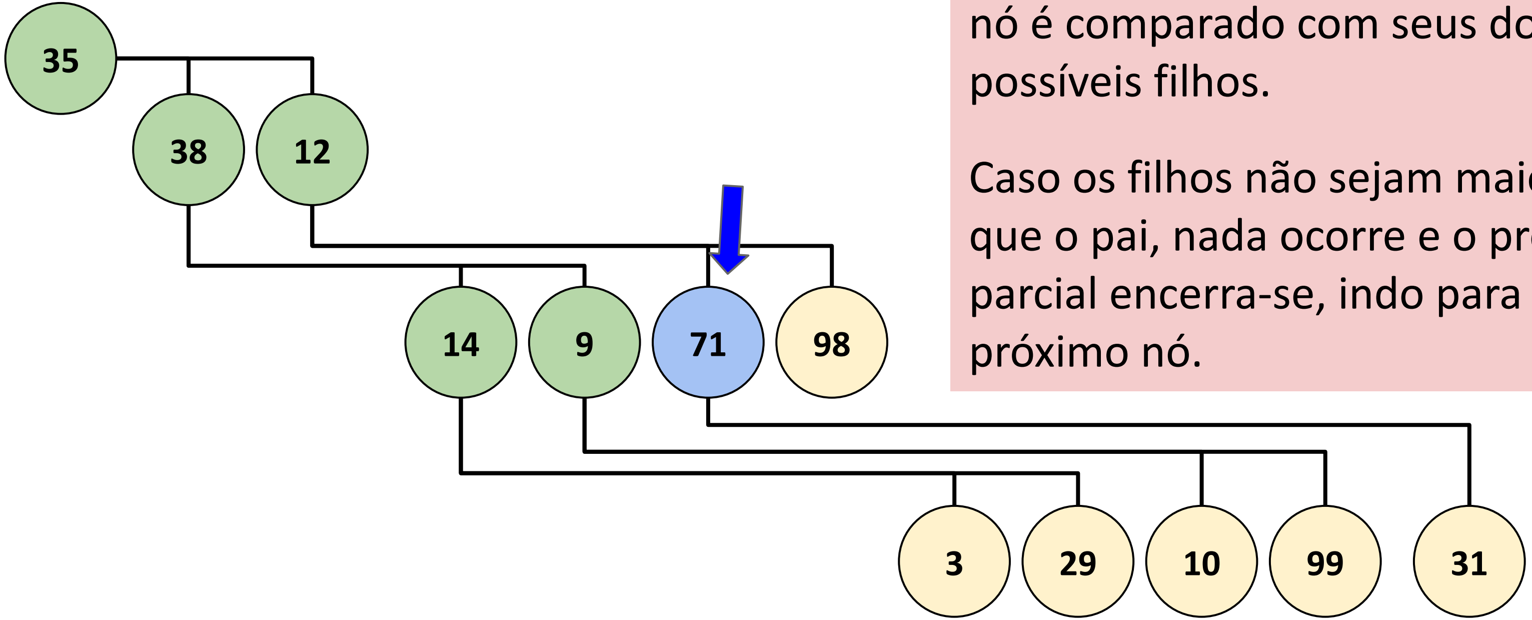


35	38	12	14	9	71	98	3	29	10	99	31

Para corrigir o heap, utilizamos a corrige-descendo, indo da primeira metade dos elementos no arranjo (metade superior na árvore) até a raiz.

Demais nós não possuem filhos, portanto, é inútil chamar o corrige-descendo.

Exemplo 1 - IV/XVII

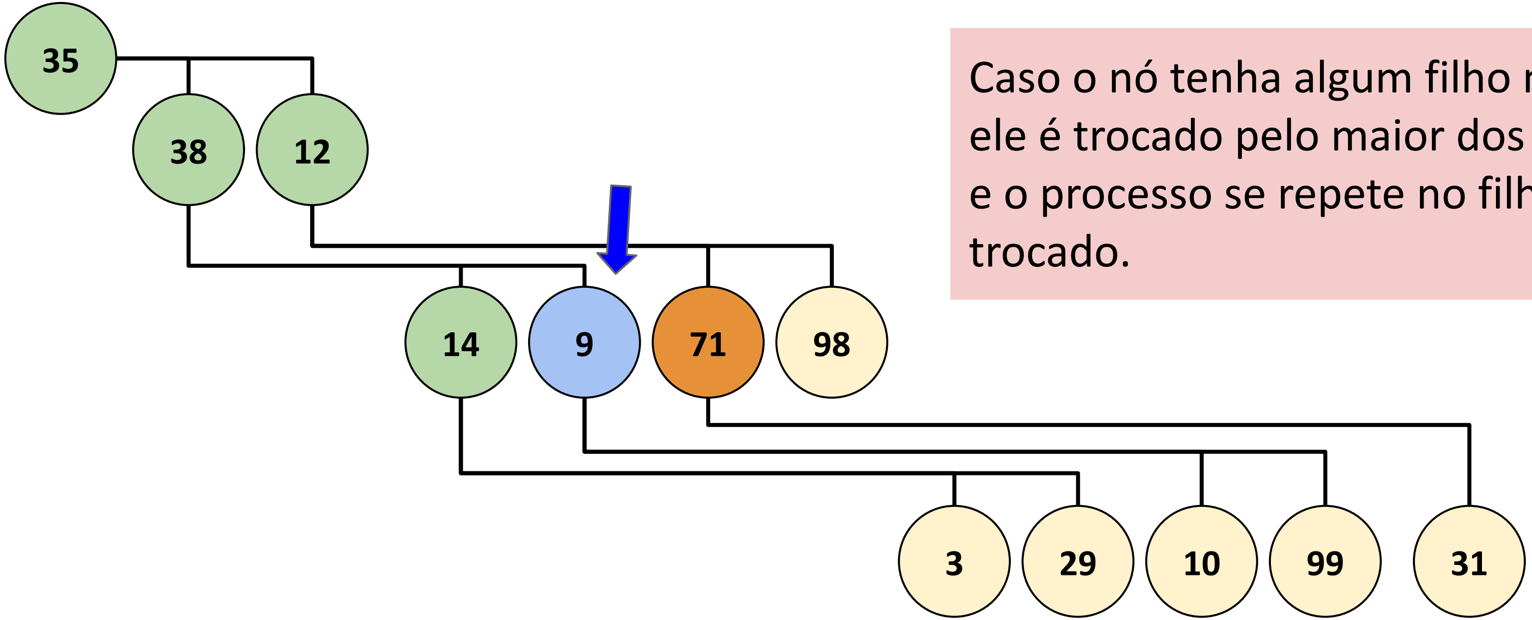


Da metade superior até a raiz, cada nó é comparado com seus dois possíveis filhos.

Caso os filhos não sejam maiores que o pai, nada ocorre e o processo parcial encerra-se, indo para o próximo nó.

35	38	12	14	9	71	98	3	29	10	99	31

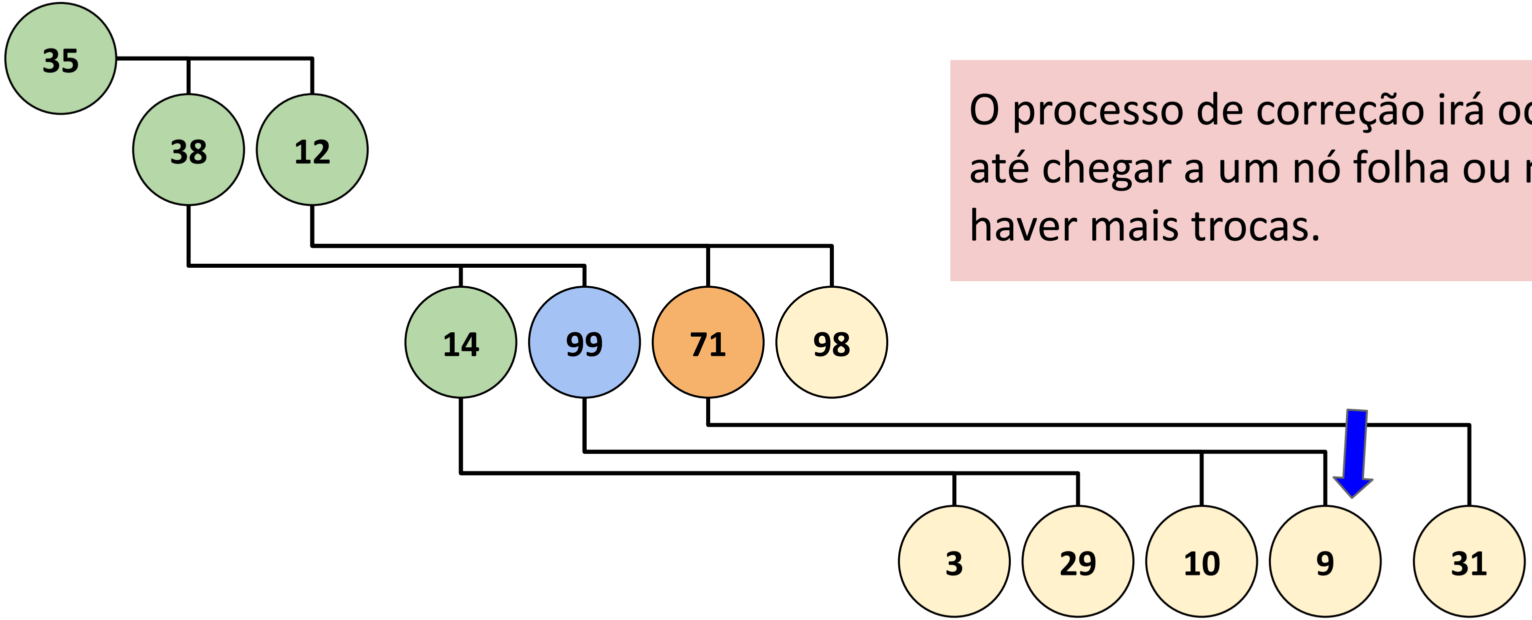
Exemplo 1 - V/XVII



Caso o nó tenha algum filho maior, ele é trocado pelo maior dos filhos e o processo se repete no filho trocado.

35	38	12	14	9	71	98	3	29	10	99	31

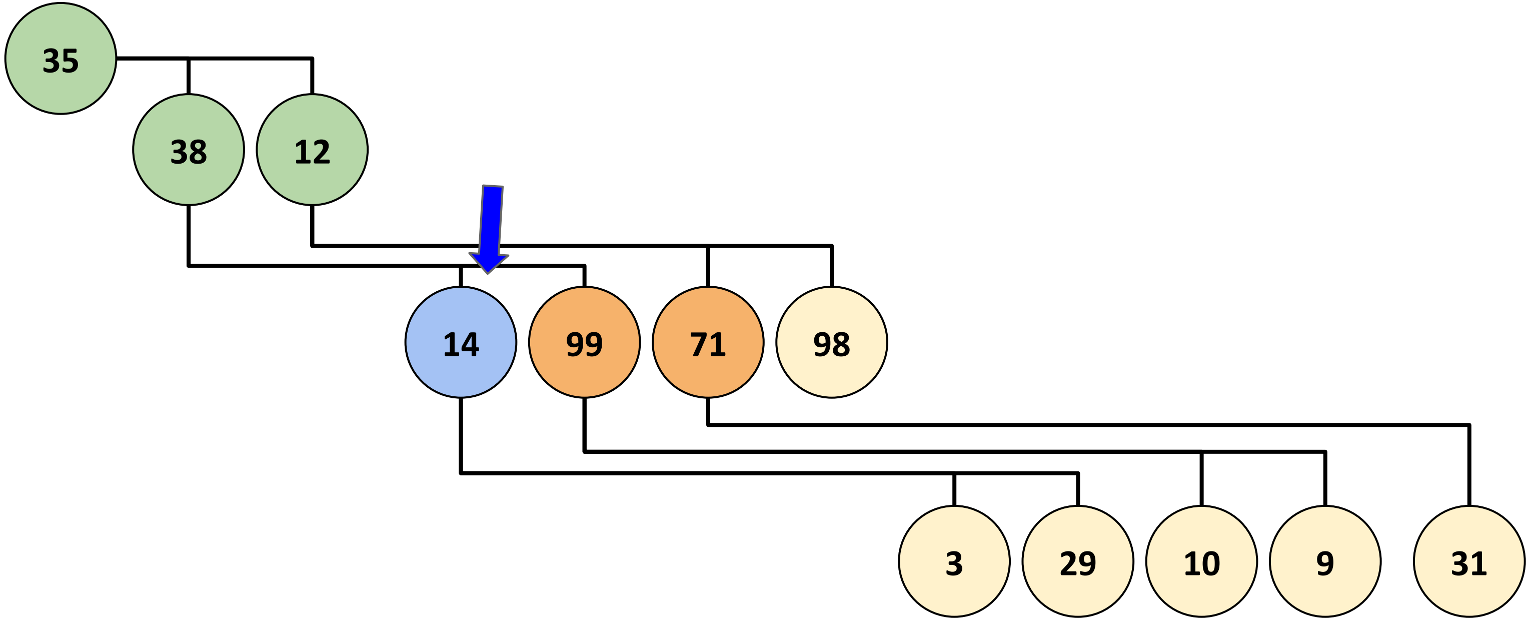
Exemplo 1 - VI/XVII



O processo de correção irá ocorrer até chegar a um nó folha ou não haver mais trocas.

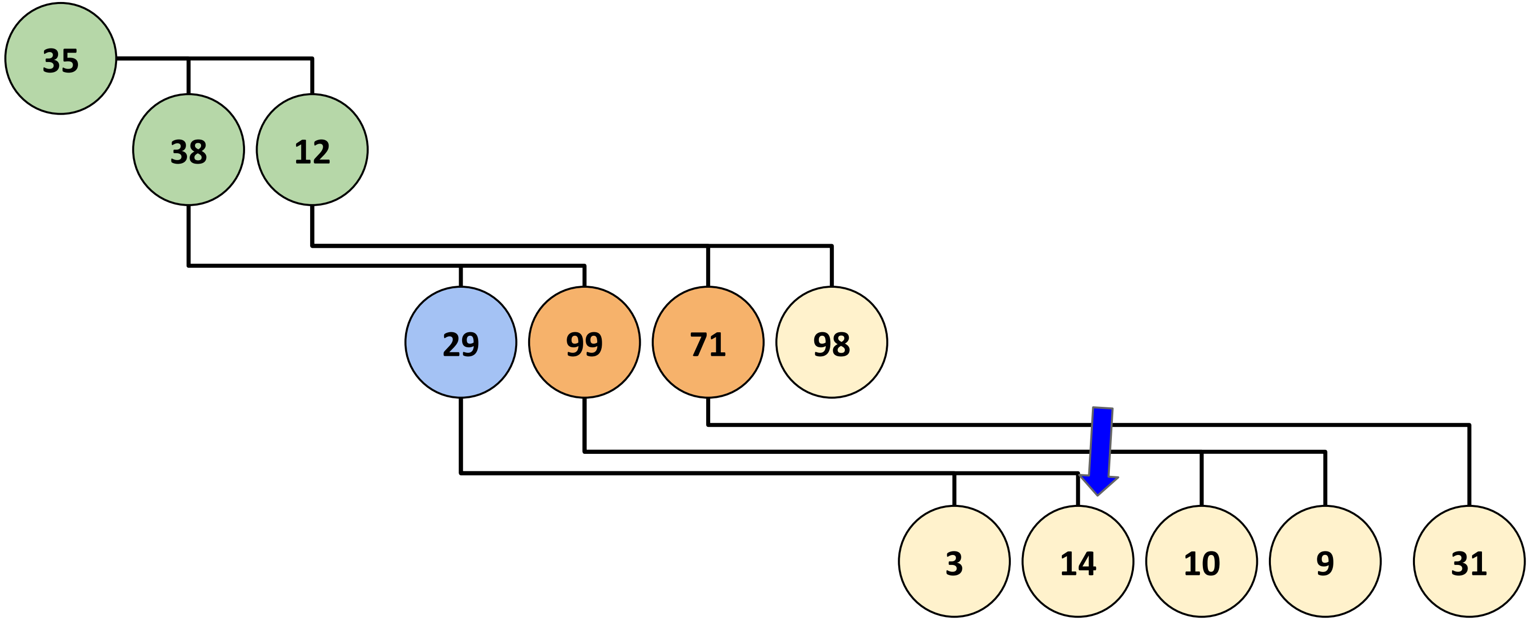
35	38	12	14	99	71	98	3	29	10	9	31

Exemplo 1 - VII/XVII



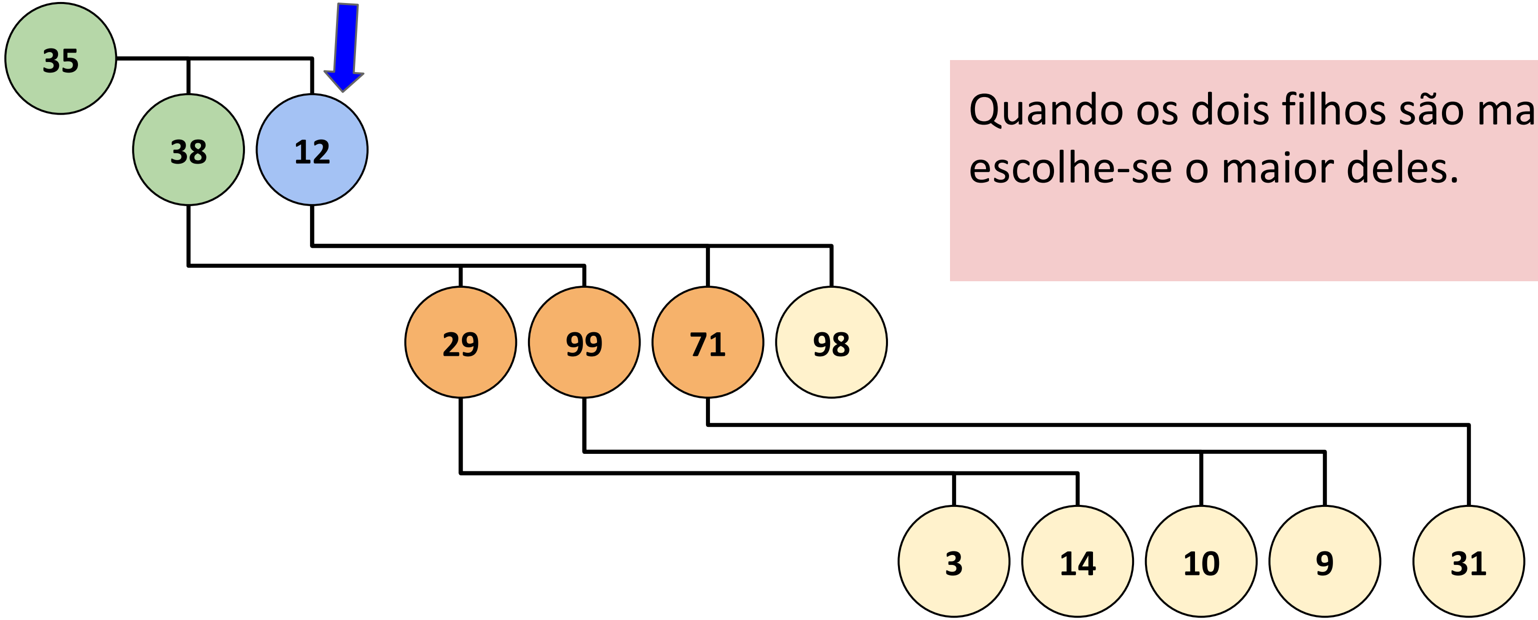
35	38	12	14	99	71	98	3	29	10	9	31

Exemplo 1 - VIII/XVII



35	38	12	29	99	71	98	3	14	10	9	31

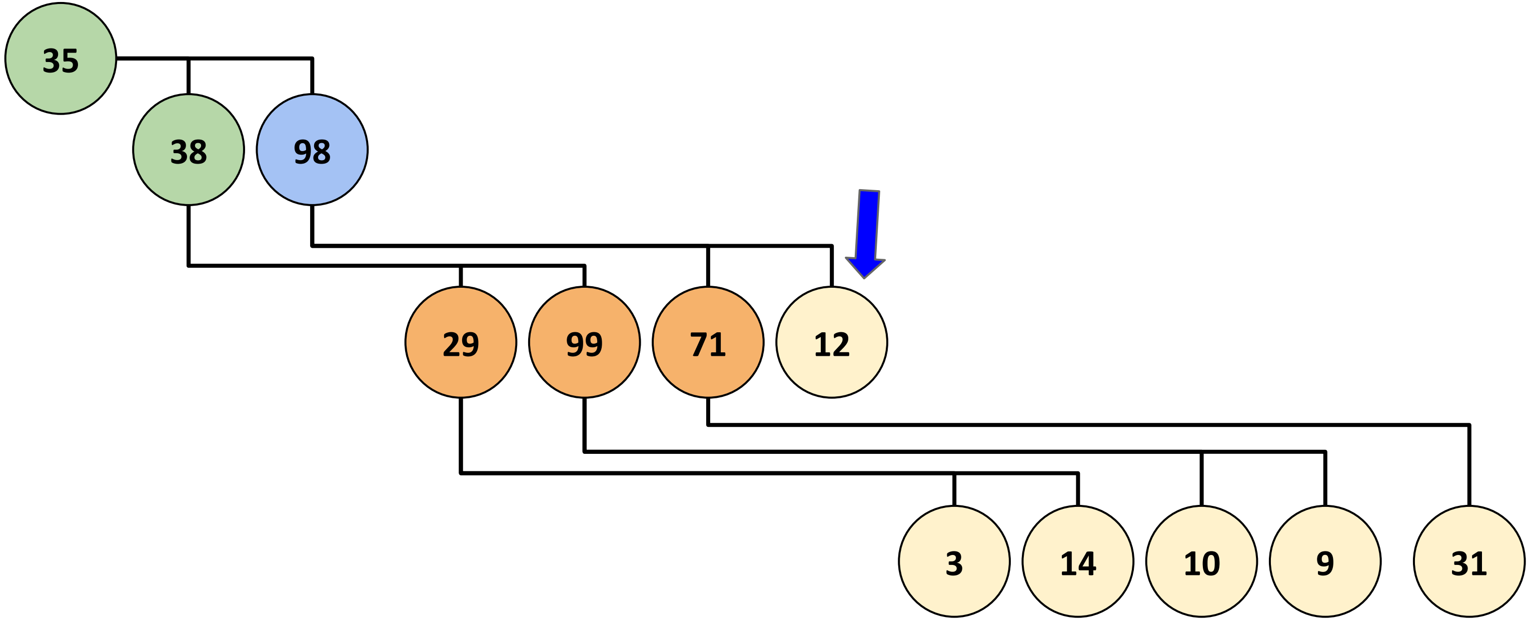
Exemplo 1 - IX/XVII



Quando os dois filhos são maiores, escolhe-se o maior deles.

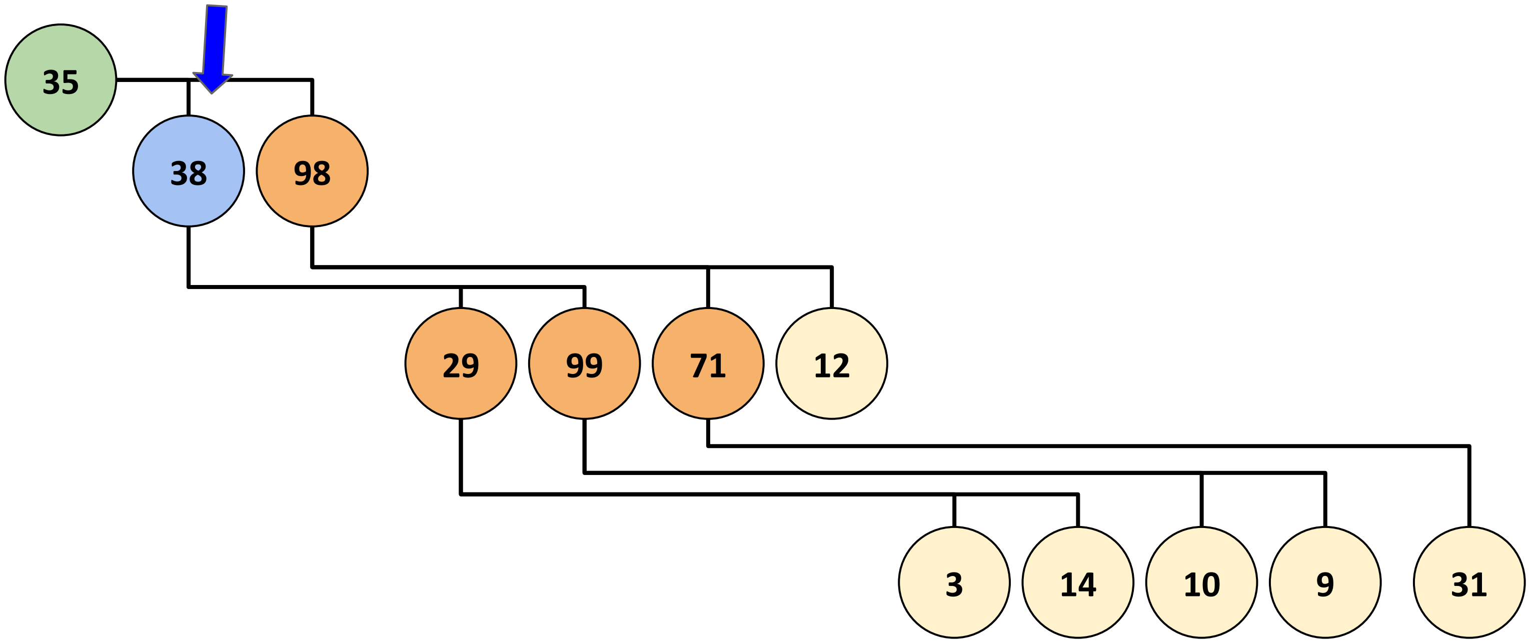
35	38	12	29	99	71	98	3	14	10	9	31

Exemplo 1 - XI/XVII



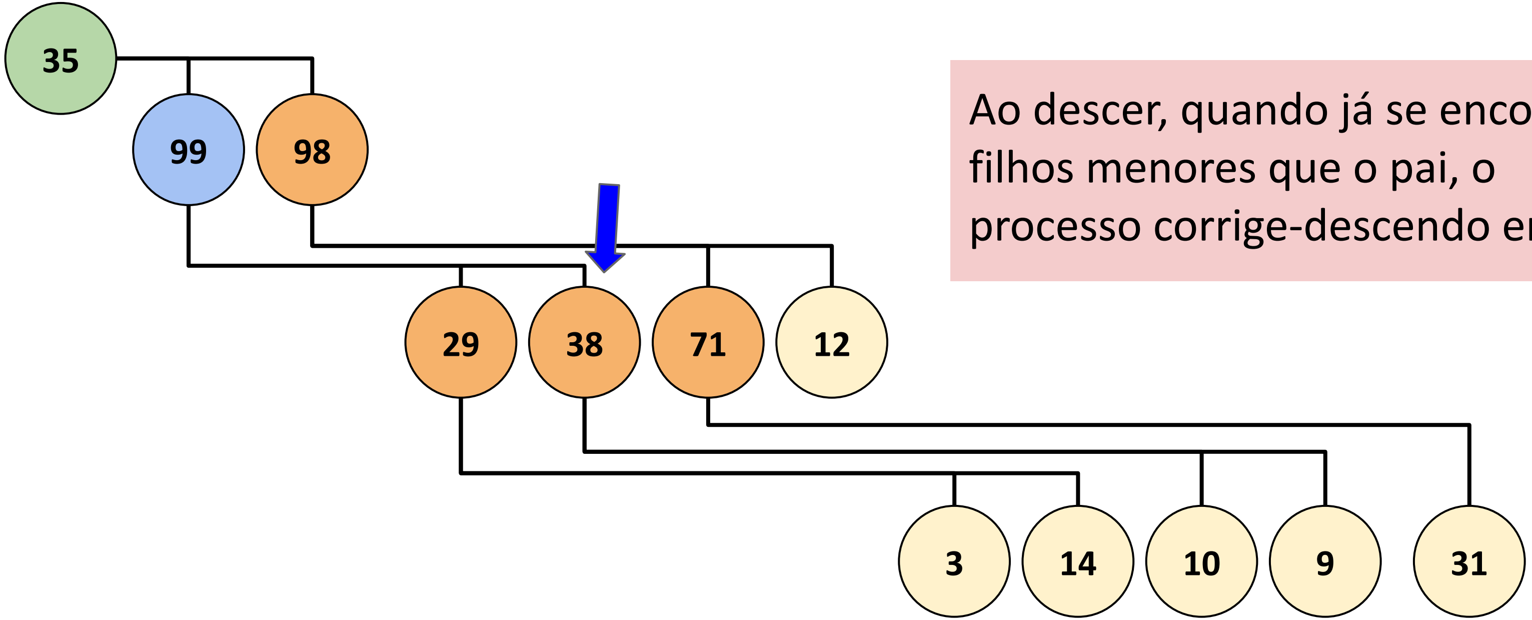
35	38	98	29	99	71	12	3	14	10	9	31

Exemplo 1 - XII/XVII



35	38	98	29	99	71	12	3	14	10	9	31

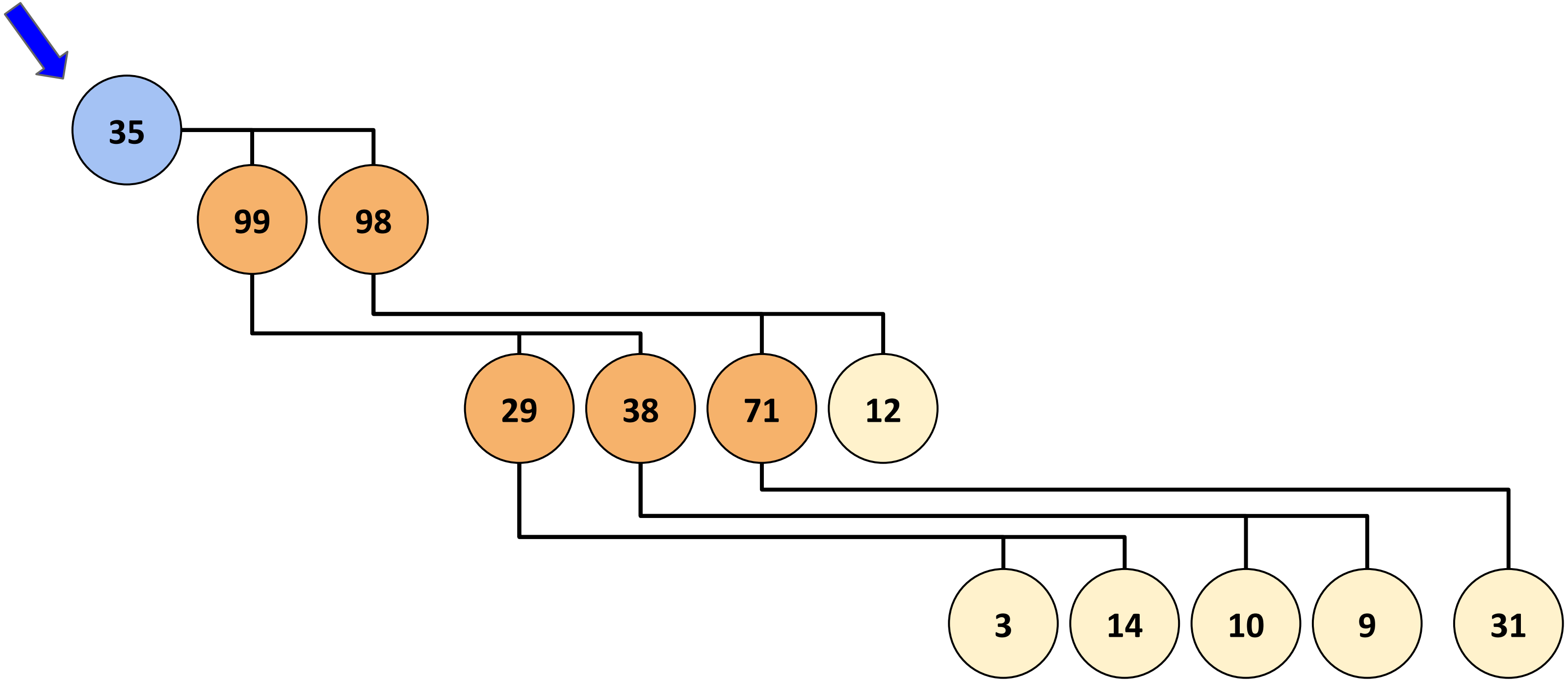
Exemplo 1 - XIII/XVII



Ao descer, quando já se encontra filhos menores que o pai, o processo corrige-descendo encerra.

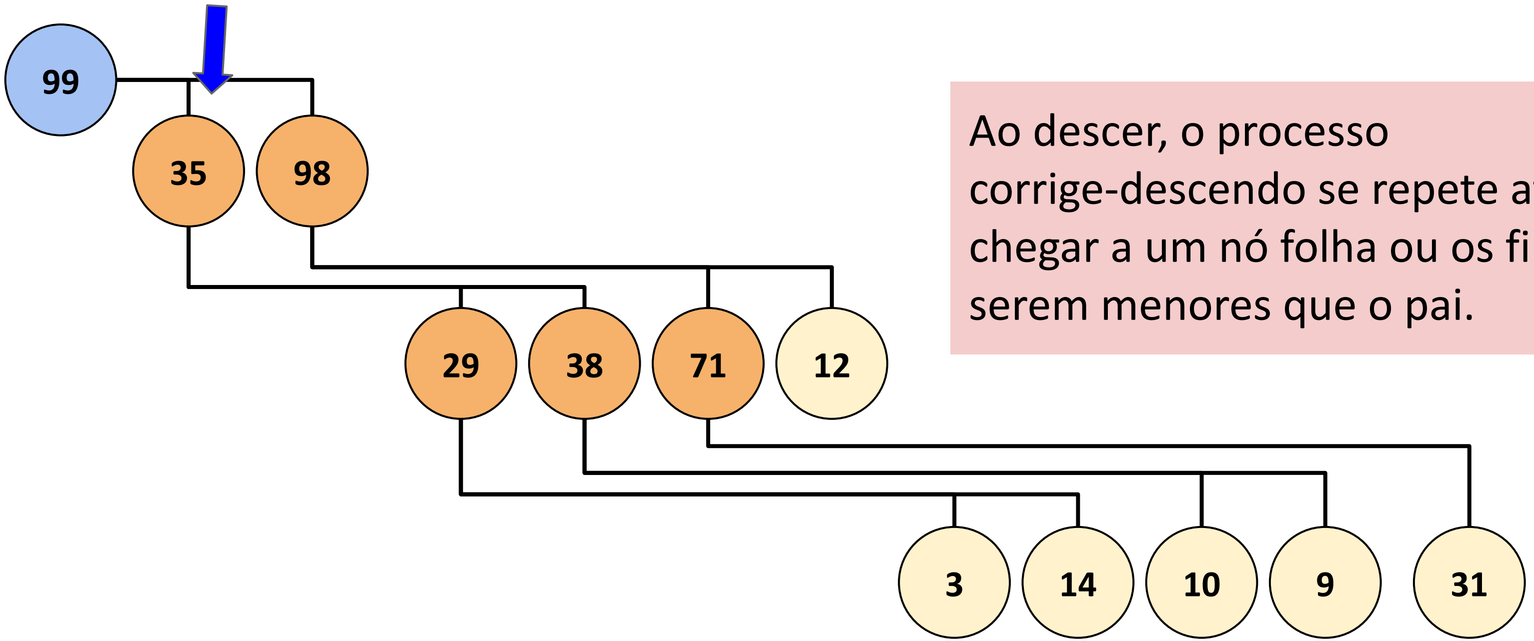
35	99	98	29	38	71	12	3	14	10	9	31

Exemplo 1 - XIV/XVII



35	99	98	29	38	71	12	3	14	10	9	31

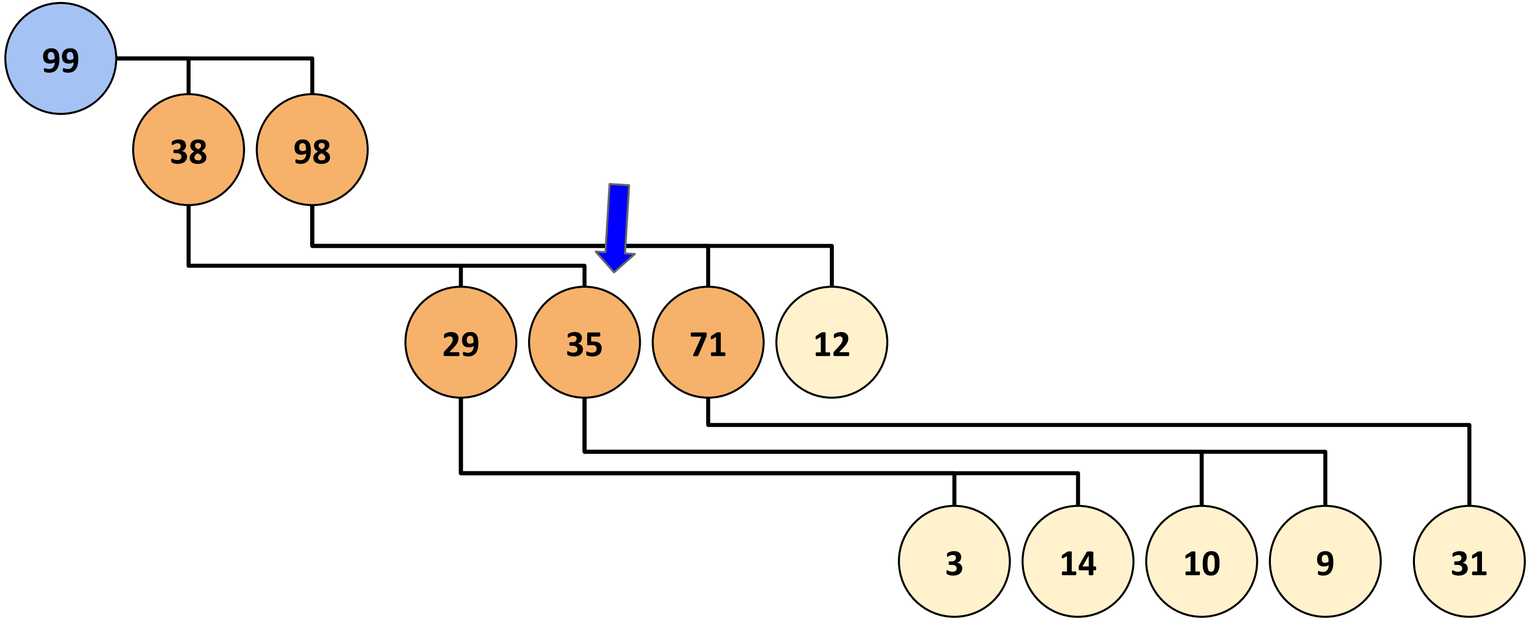
Exemplo 1 - XV/XVII



Ao descer, o processo corrige-descendo se repete até chegar a um nó folha ou os filhos serem menores que o pai.

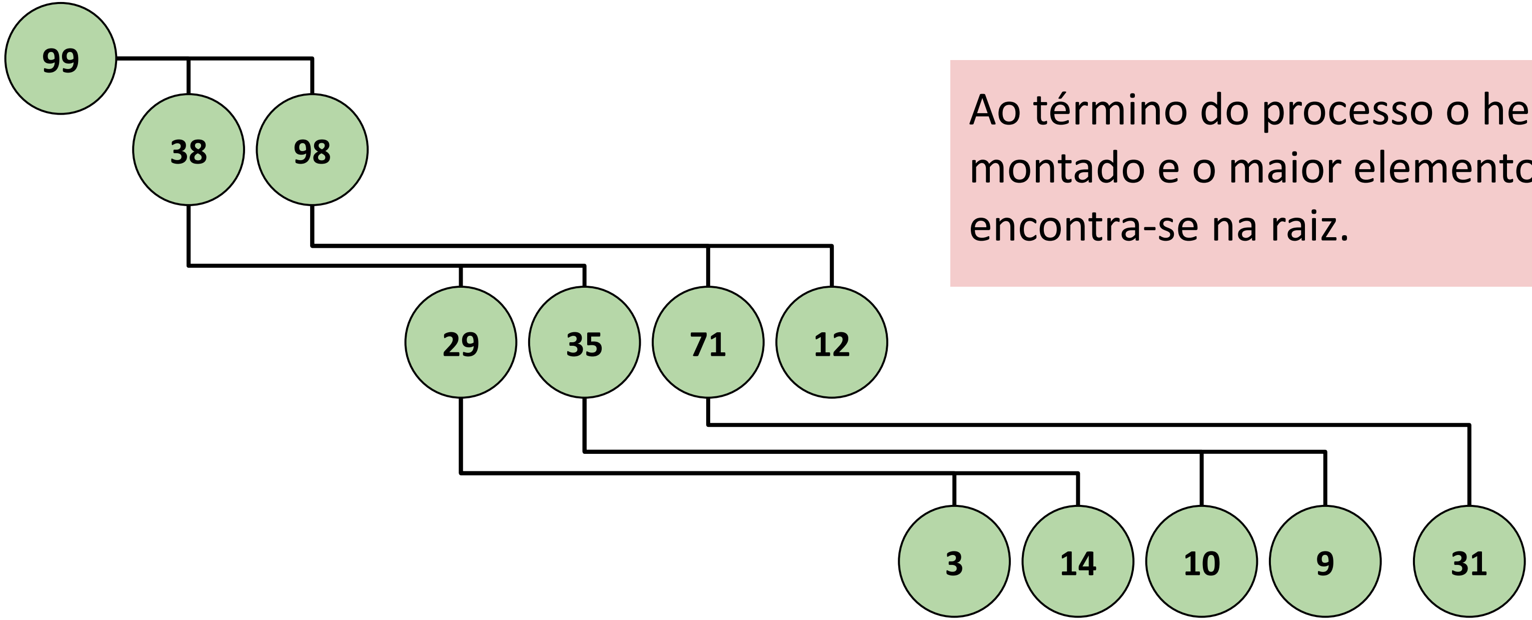
99	35	98	29	38	71	12	3	14	10	9	31

Exemplo 1 - XVI/XVII



99	38	98	29	35	71	12	3	14	10	9	31

Exemplo 1 - XVII/XVII



Ao término do processo o heap está montado e o maior elemento encontra-se na raiz.

99	38	98	29	35	71	12	3	14	10	9	31

Retirada e Inserção de Elementos

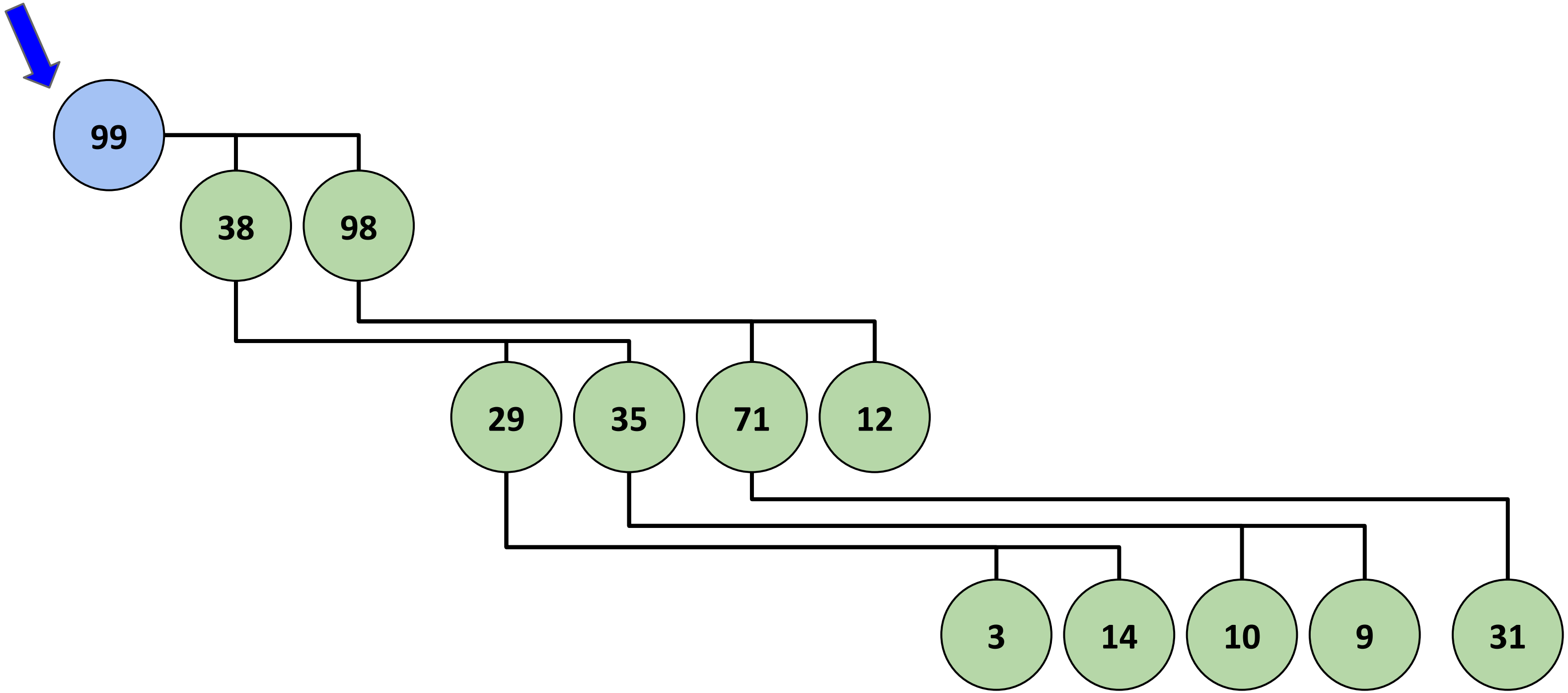
Exemplo 2 - I/XV

Suponha que, com o heap montado, precisemos retirar quatro elementos.

A retirada de um elemento no heap só é possível no topo.

Após a retirada, o heap deve ser reorganizado. Para isso, troca-se a raiz com o último elemento, aplicando-se corrige-descendo na nova raiz.

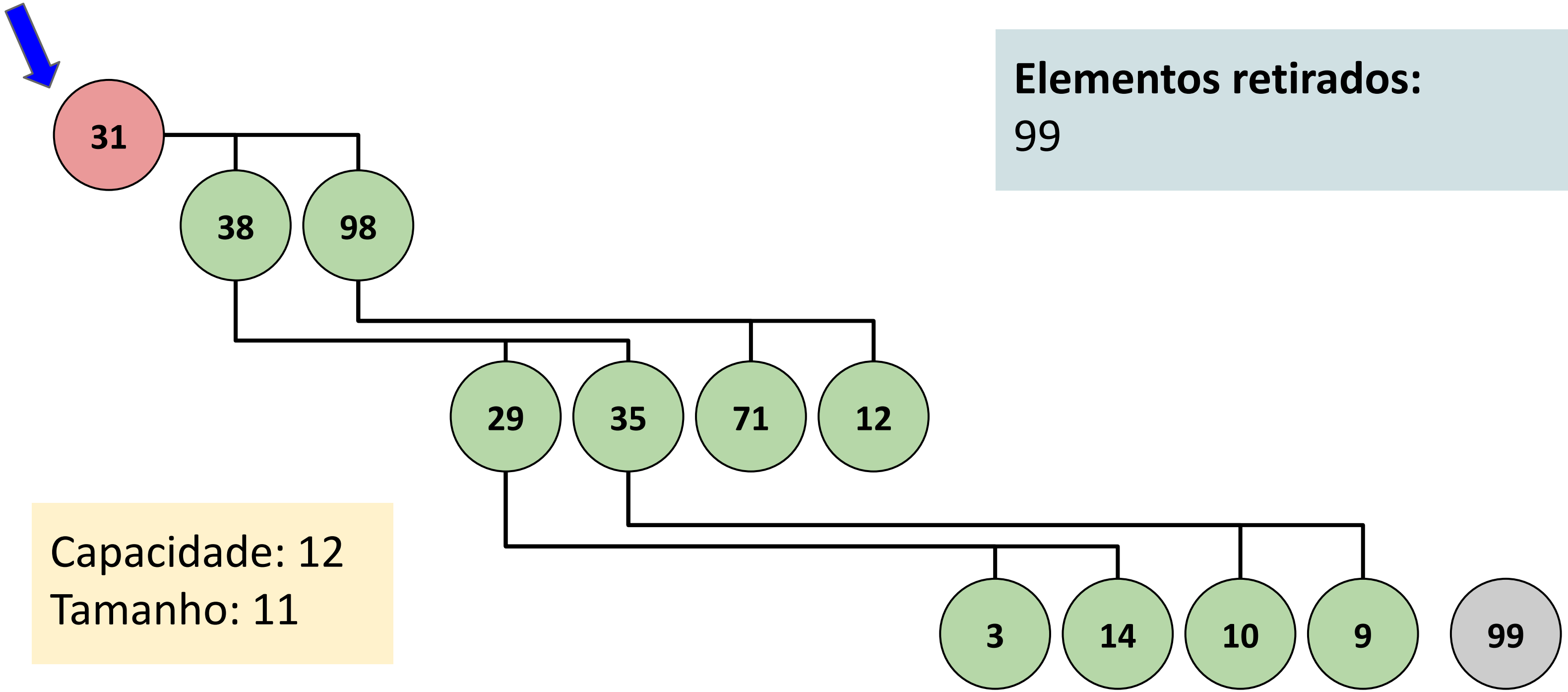
Exemplo 2 - II/XV



99	38	98	29	35	71	12	3	14	10	9	31

Exemplo 2 - III/XV

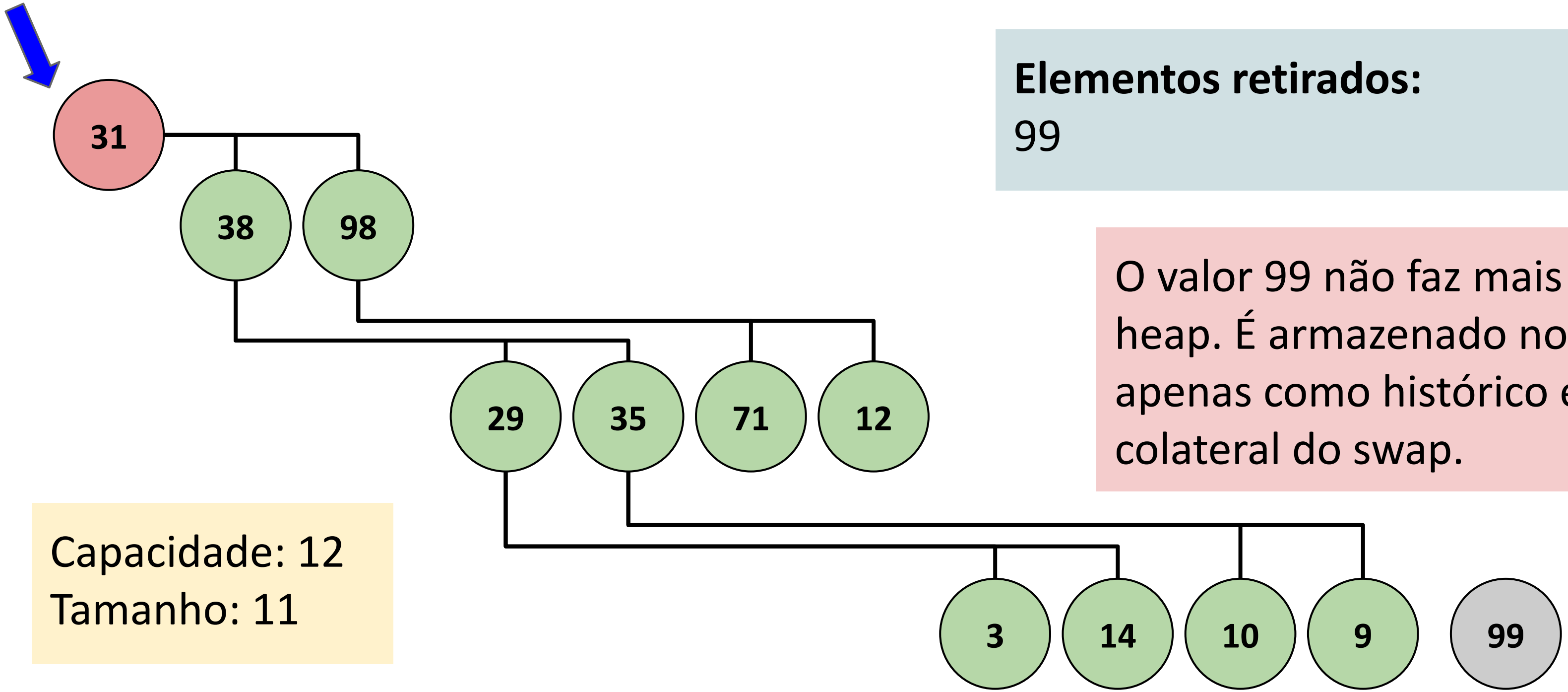
Elementos retirados:
99



Capacidade: 12
Tamanho: 11

31	38	98	29	35	71	12	3	14	10	9	99

Exemplo 2 - IV/XV



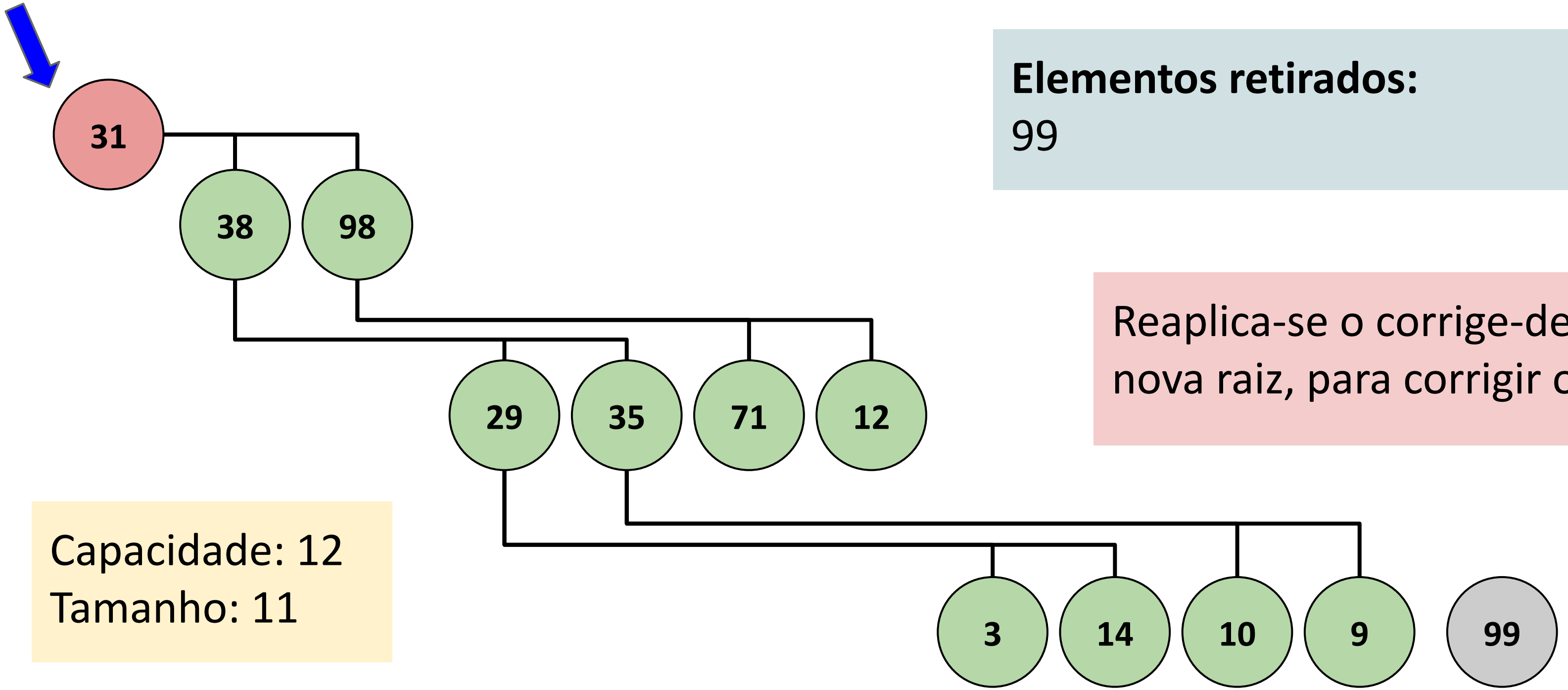
Elementos retirados:
99

O valor 99 não faz mais parte do heap. É armazenado no arranjo apenas como histórico e efeito colateral do swap.

Capacidade: 12
Tamanho: 11

31	38	98	29	35	71	12	3	14	10	9	99

Exemplo 2 - V/XV



Elementos retirados:
99

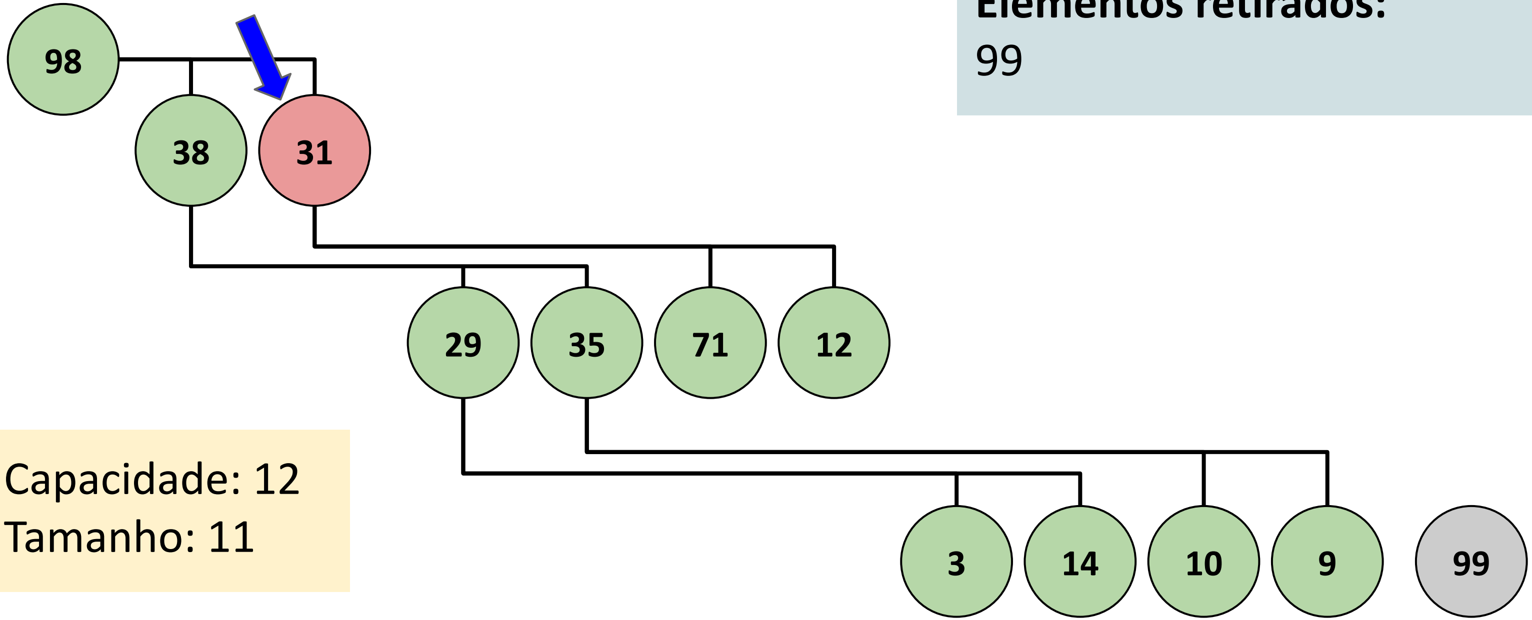
Reaplica-se o corrige-descendo na nova raiz, para corrigir o novo heap.

Capacidade: 12
Tamanho: 11

31	38	98	29	35	71	12	3	14	10	9	99

Exemplo 2 - VI/XV

Elementos retirados:
99

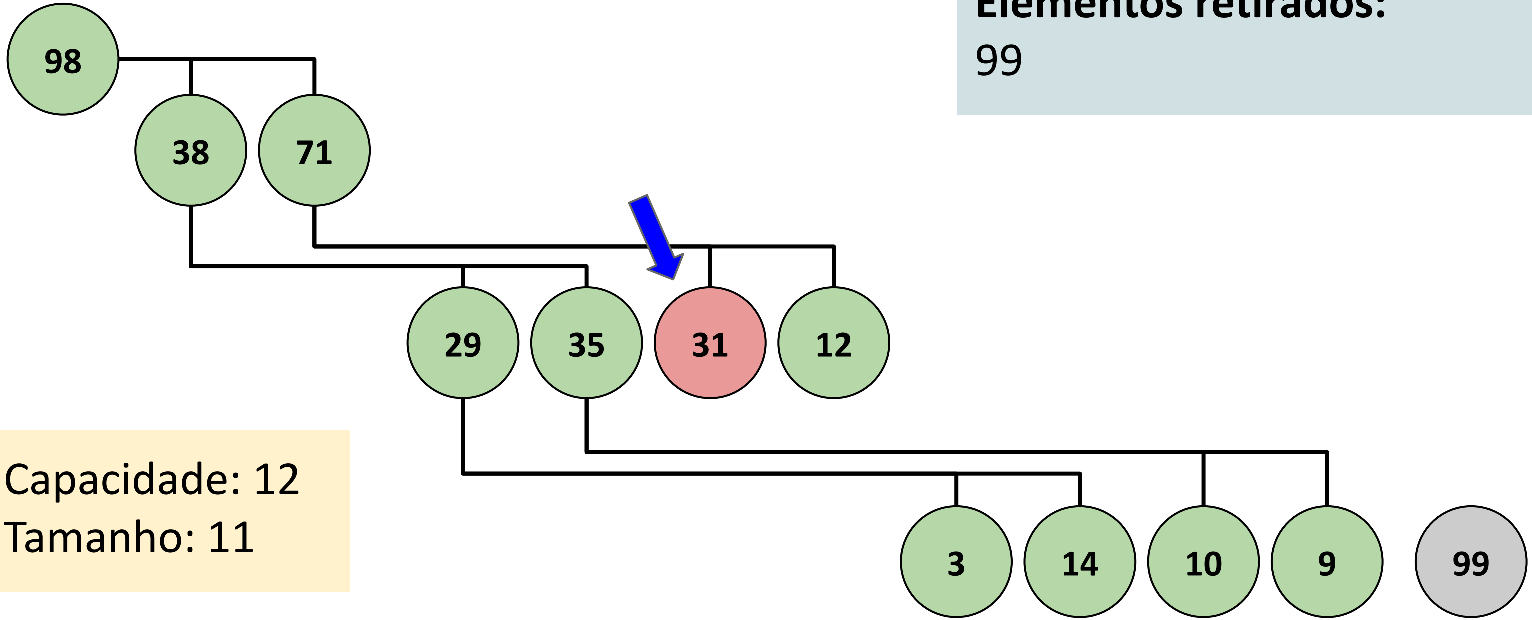


Capacidade: 12
Tamanho: 11

98	38	31	29	35	71	12	3	14	10	9	99

Exemplo 2 - VII/XV

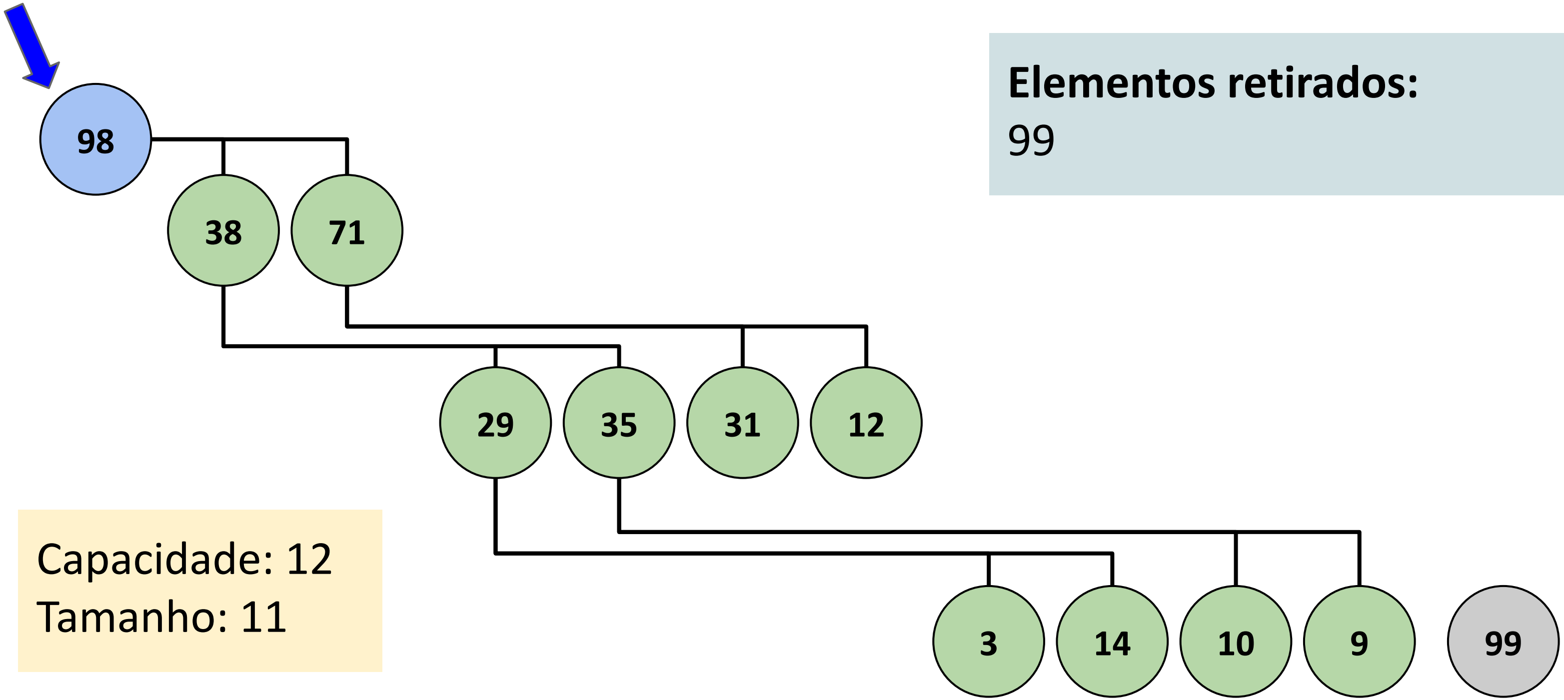
Elementos retirados:
99



Capacidade: 12
Tamanho: 11

98	38	71	29	35	31	12	3	14	10	9	99

Exemplo 2 - VIII/XV

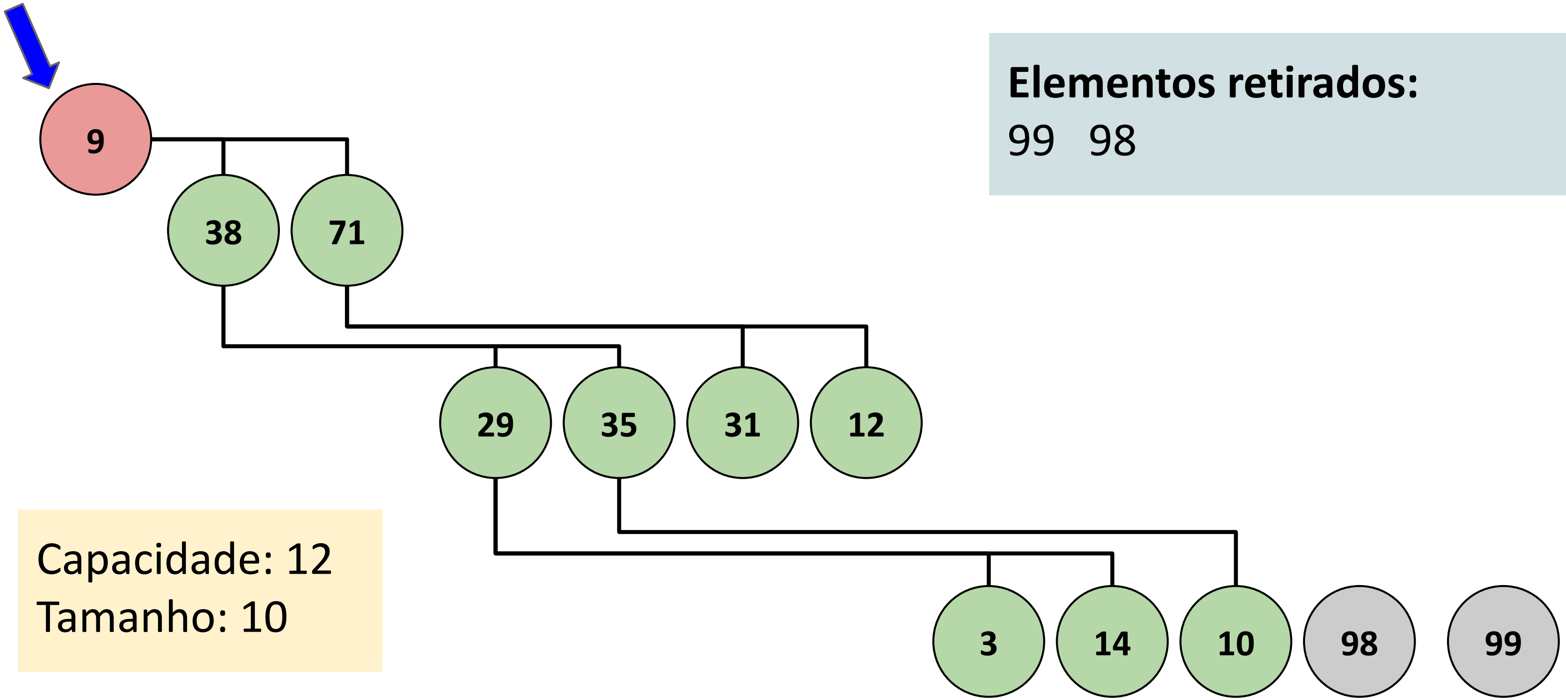


Elementos retirados:
99

Capacidade: 12
Tamanho: 11

98	**38**	**71**	**29**	**35**	**31**	**12**	**3**	**14**	**10**	**9**	**99**

Exemplo 2 - IX/XV



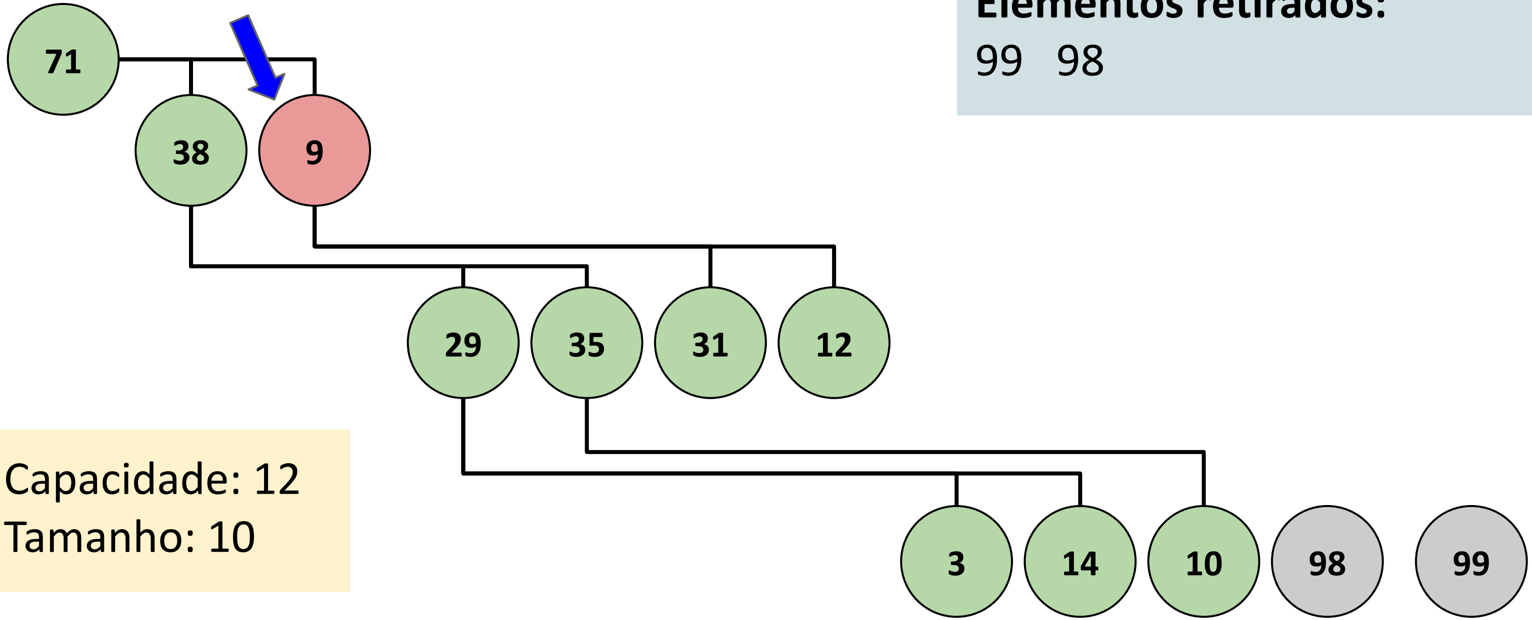
Elementos retirados:
99 98

Capacidade: 12
Tamanho: 10

9	38	71	29	35	31	12	3	14	10	98	99

Exemplo 2 - X/XV

Elementos retirados:
99 98

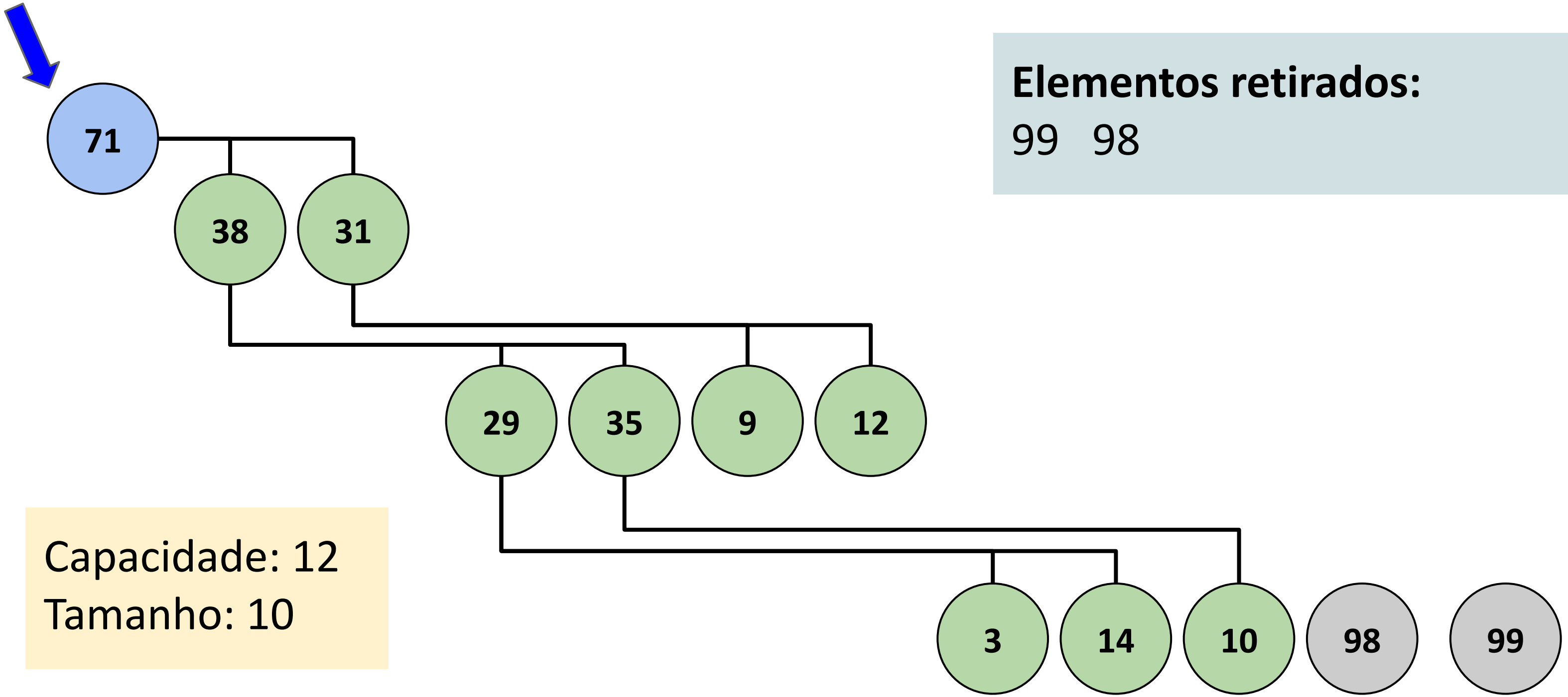


Capacidade: 12
Tamanho: 10

71	38	9	29	35	31	12	3	14	10	98	99

Exemplo 2 - XI/XV

Elementos retirados:
99 98

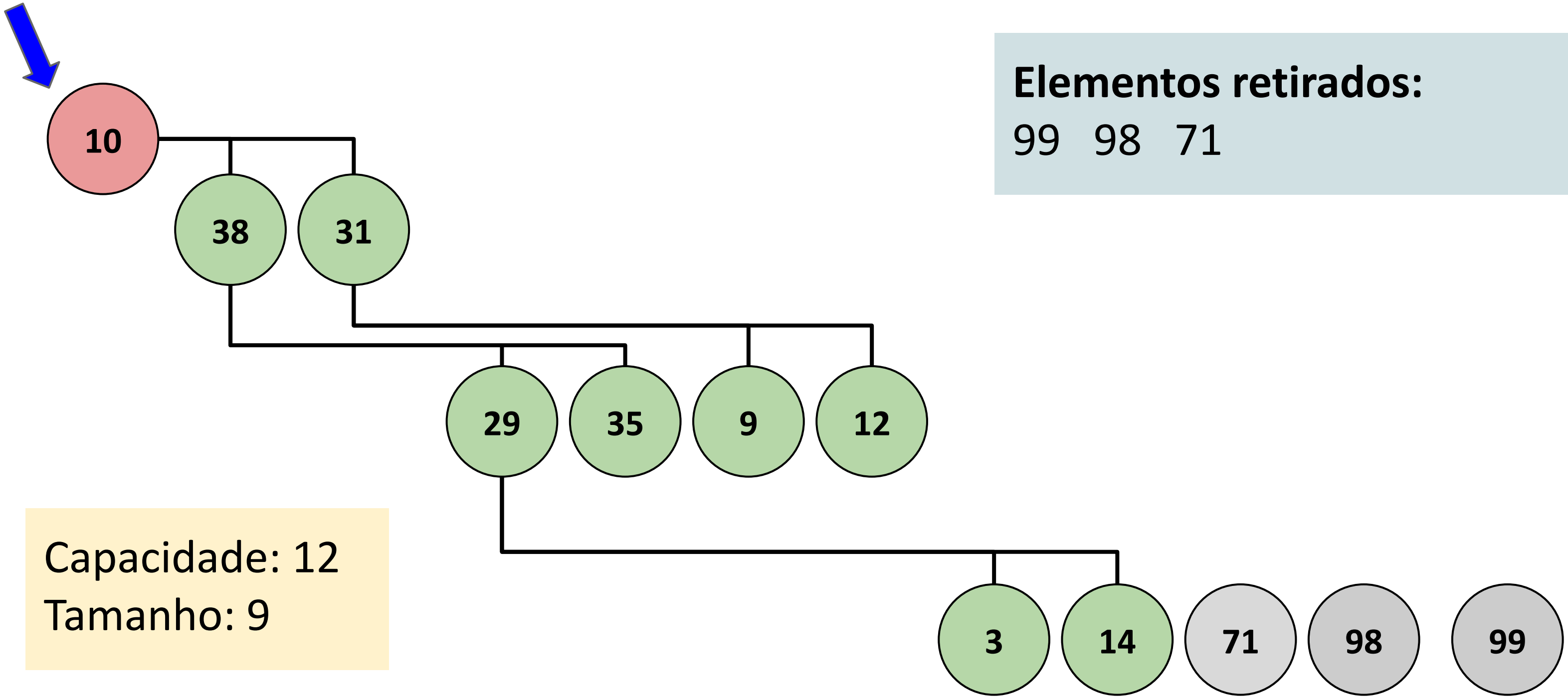


Capacidade: 12
Tamanho: 10

71	38	31	29	35	9	12	3	14	10	98	99

Exemplo 2 - XII/XV

Elementos retirados:
99 98 71



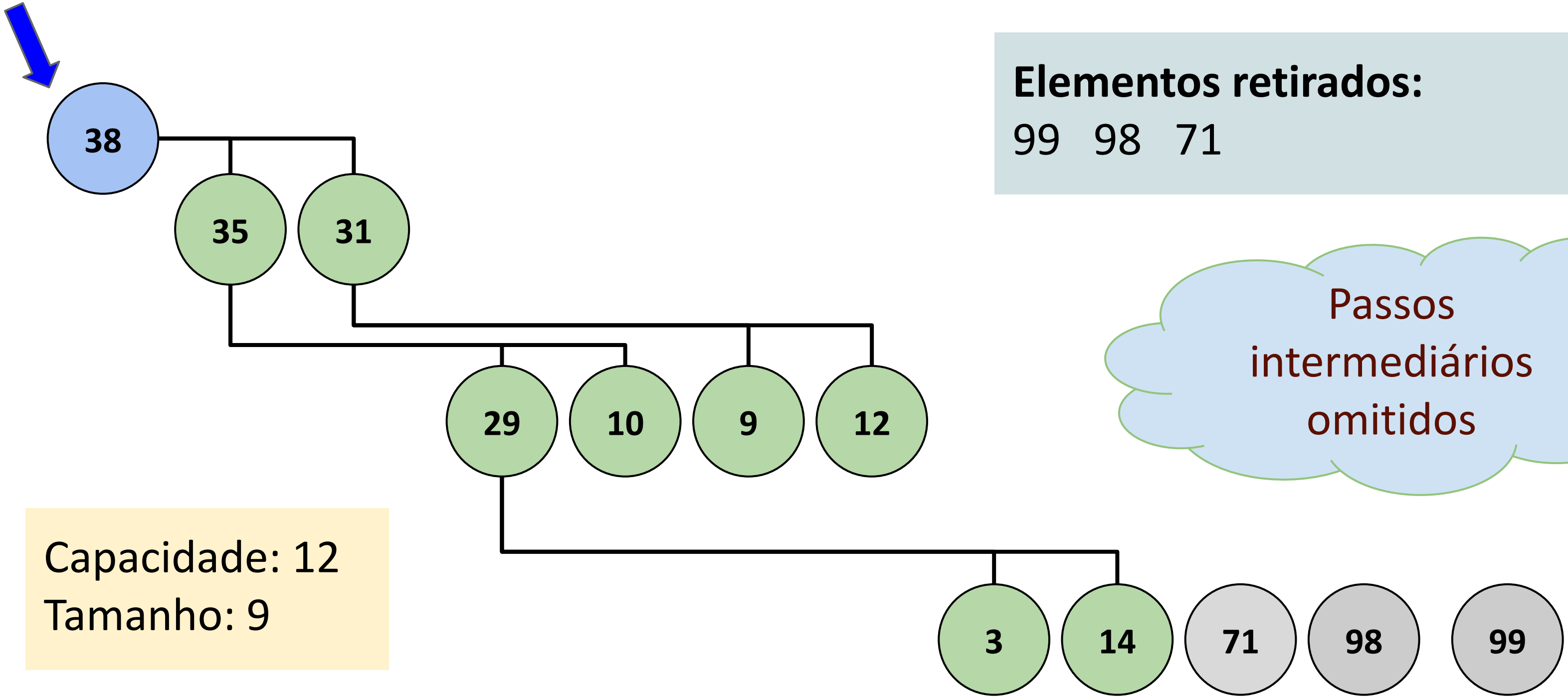
Capacidade: 12
Tamanho: 9

10	38	31	29	35	9	12	3	14	71	98	99

Exemplo 2 - XIII/XV

Elementos retirados:
99 98 71

Passos intermediários omitidos

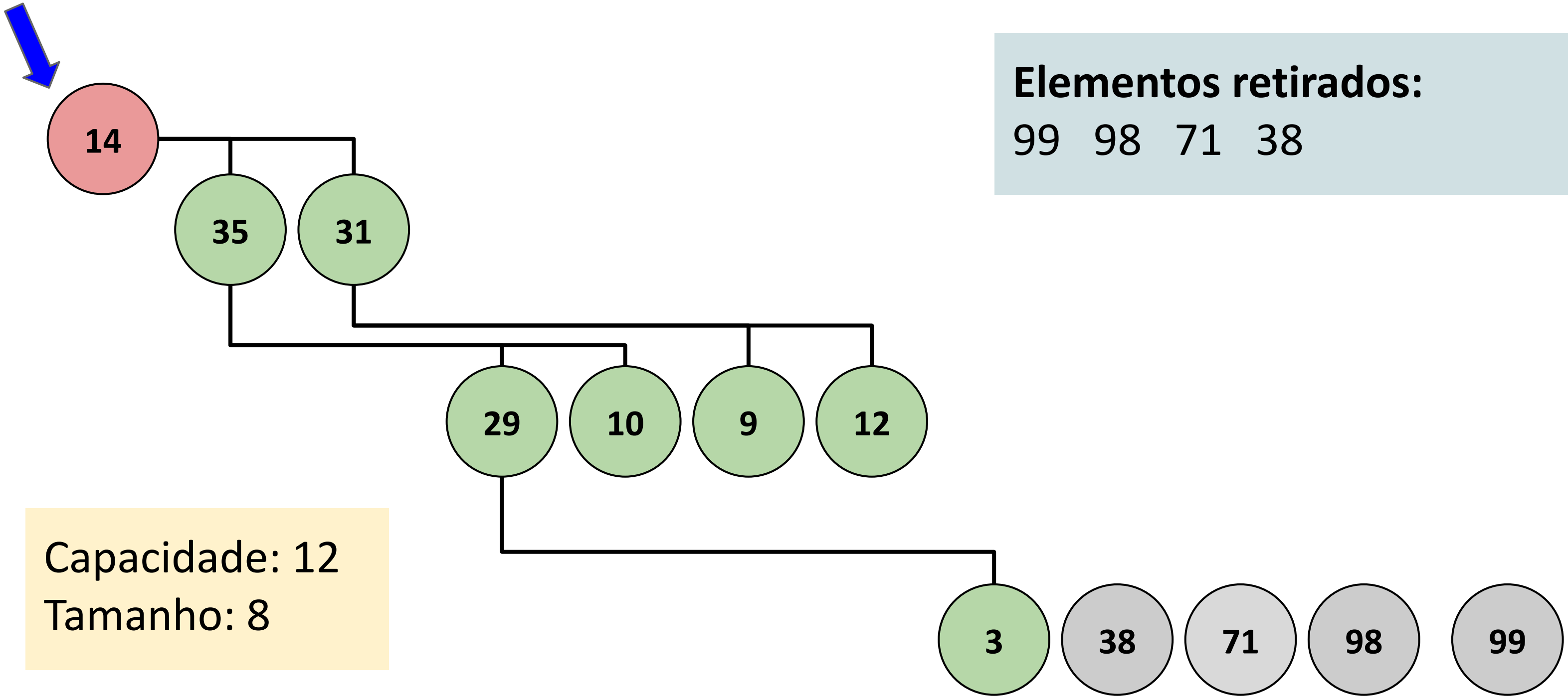


Capacidade: 12
Tamanho: 9

38	35	31	29	10	9	12	3	14	71	98	99

Exemplo 2 - XIV/XV

Elementos retirados:
99 98 71 38



Capacidade: 12
Tamanho: 8

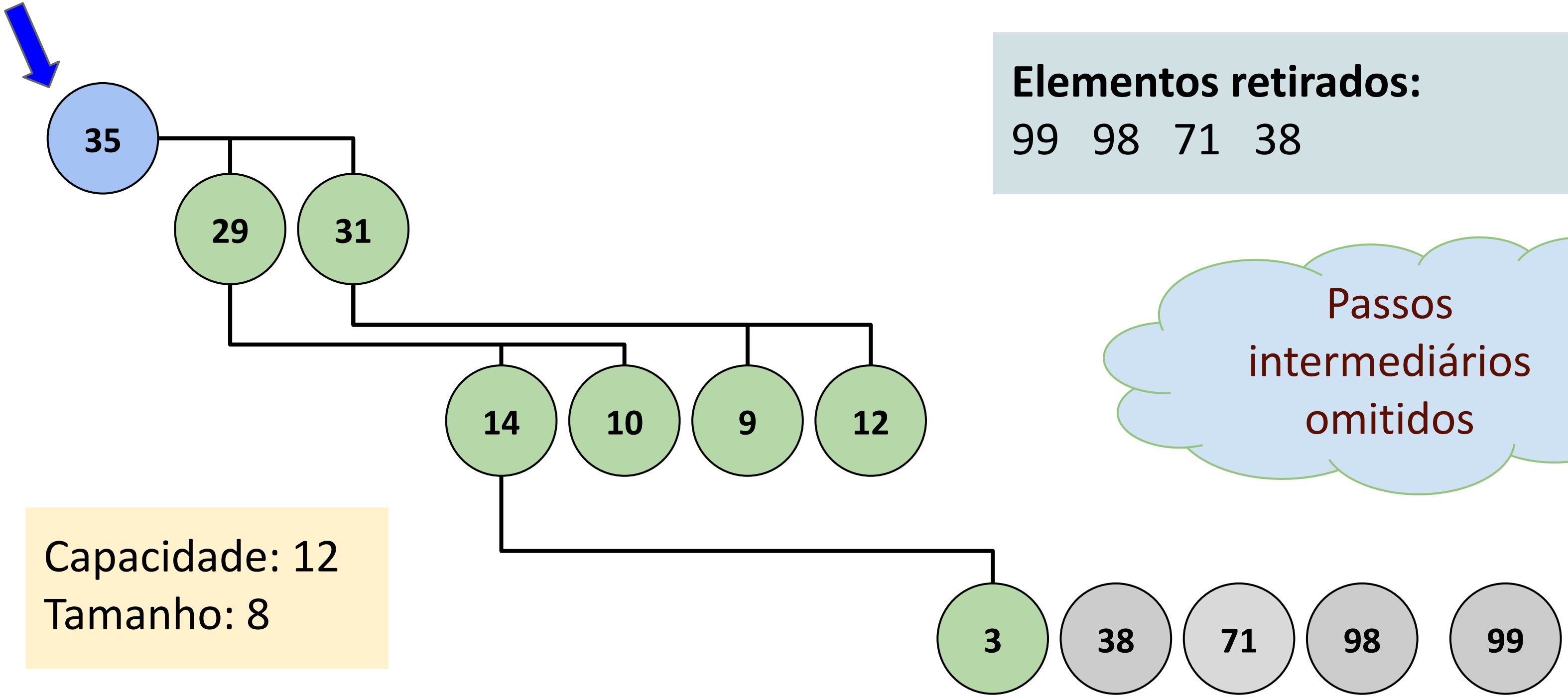
14	35	31	29	10	9	12	3	38	71	98	99

Exemplo 2 - XV/XV

Elementos retirados:
99 98 71 38

Passos intermediários omitidos

Capacidade: 12
Tamanho: 8



35	29	31	14	10	9	12	3	38	71	98	99

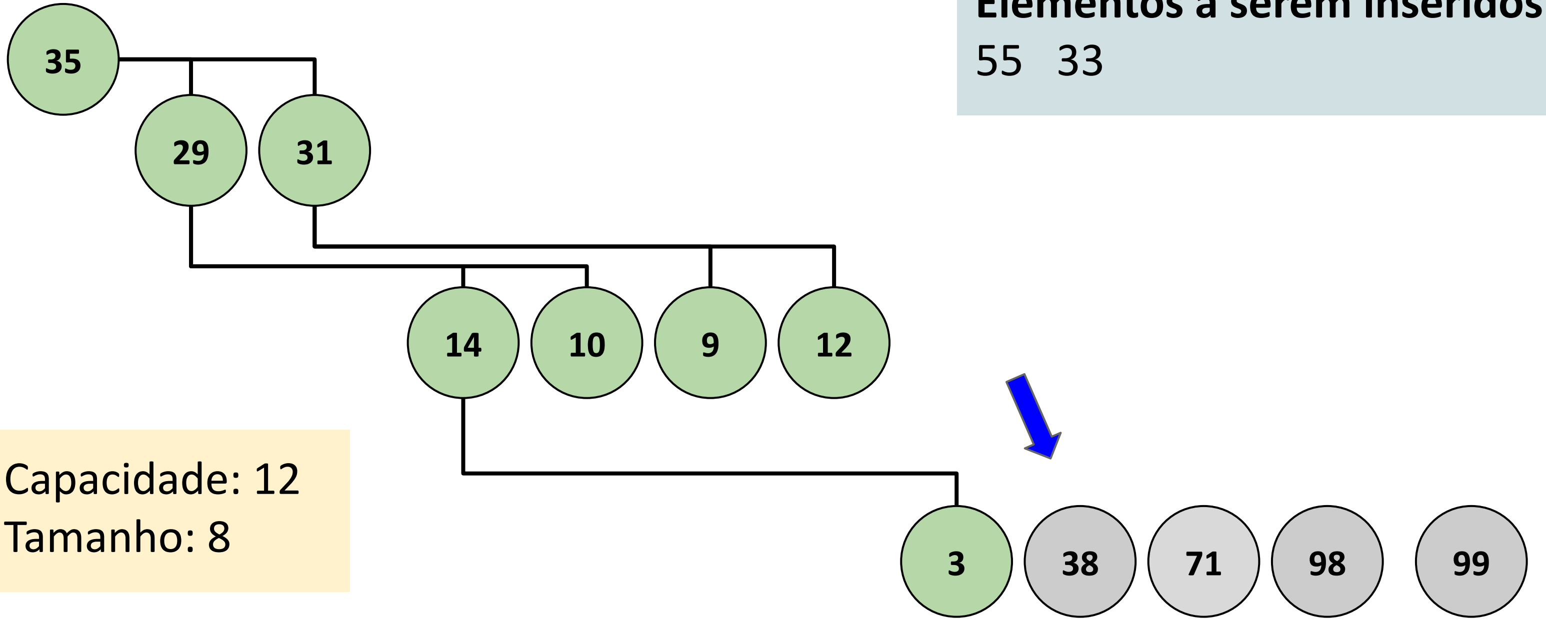
Exemplo 3 - I/XI

Suponha que, com o heap anterior, com capacidade para doze elementos, mas utilizando oito posições, queiramos inserir dois novos elementos: 55 e 33.

A inserção é sempre feita após a posição final do heap, chamando-se o corrige-subindo logo em seguida.

Exemplo 3 - II/XI

Elementos a serem inseridos:
55 33

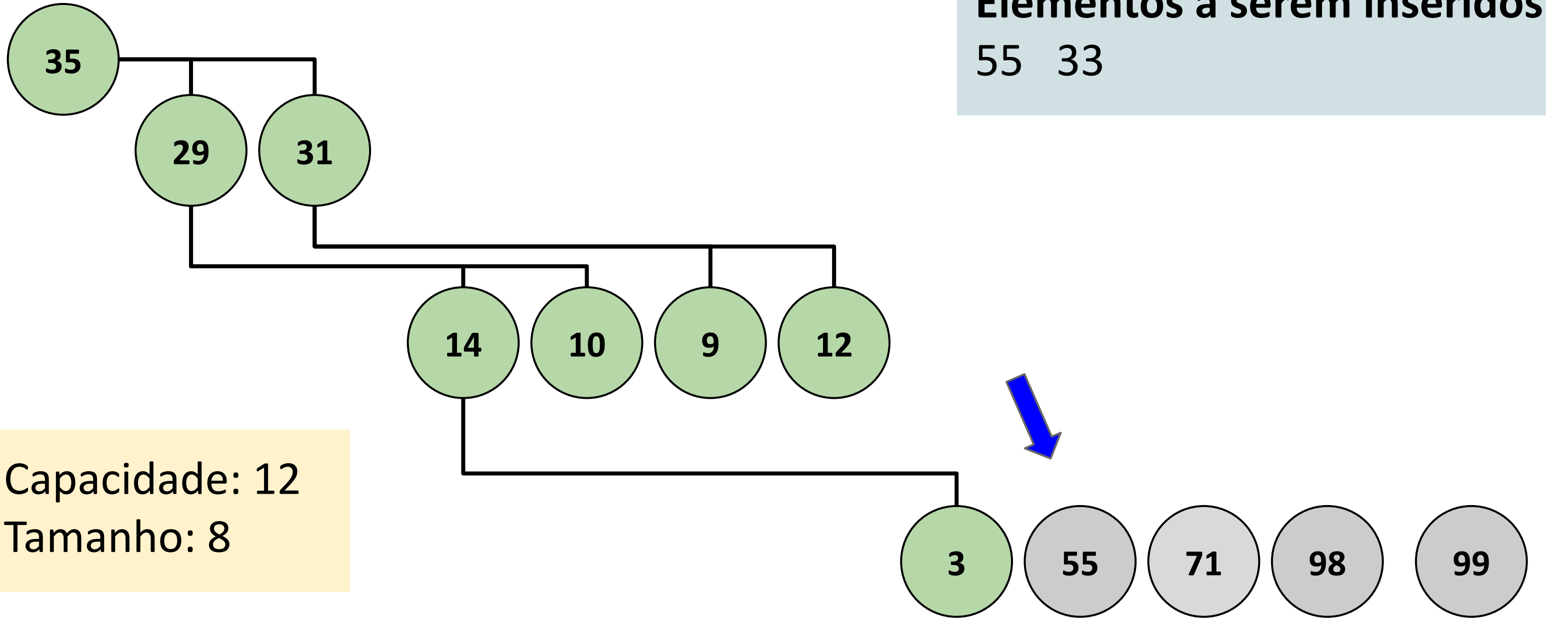


Capacidade: 12
Tamanho: 8

35	**29**	**31**	**14**	**10**	**9**	**12**	**3**	**38**	**71**	**98**	**99**

Exemplo 3 - III/XI

Elementos a serem inseridos:
55 33

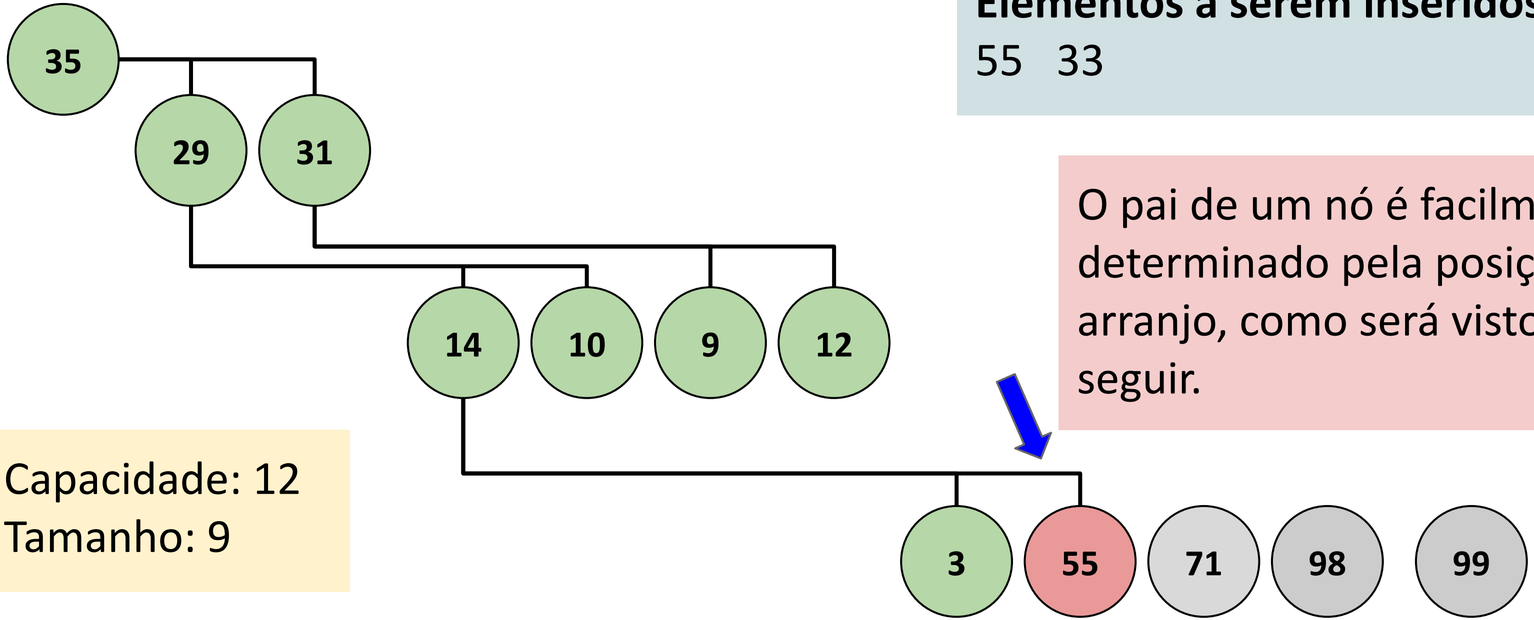


Capacidade: 12
Tamanho: 8

35	29	31	14	10	9	12	3	55	71	98	99

Exemplo 3 - IV/XI

Elementos a serem inseridos:
55 33



O pai de um nó é facilmente determinado pela posição no arranjo, como será visto logo a seguir.

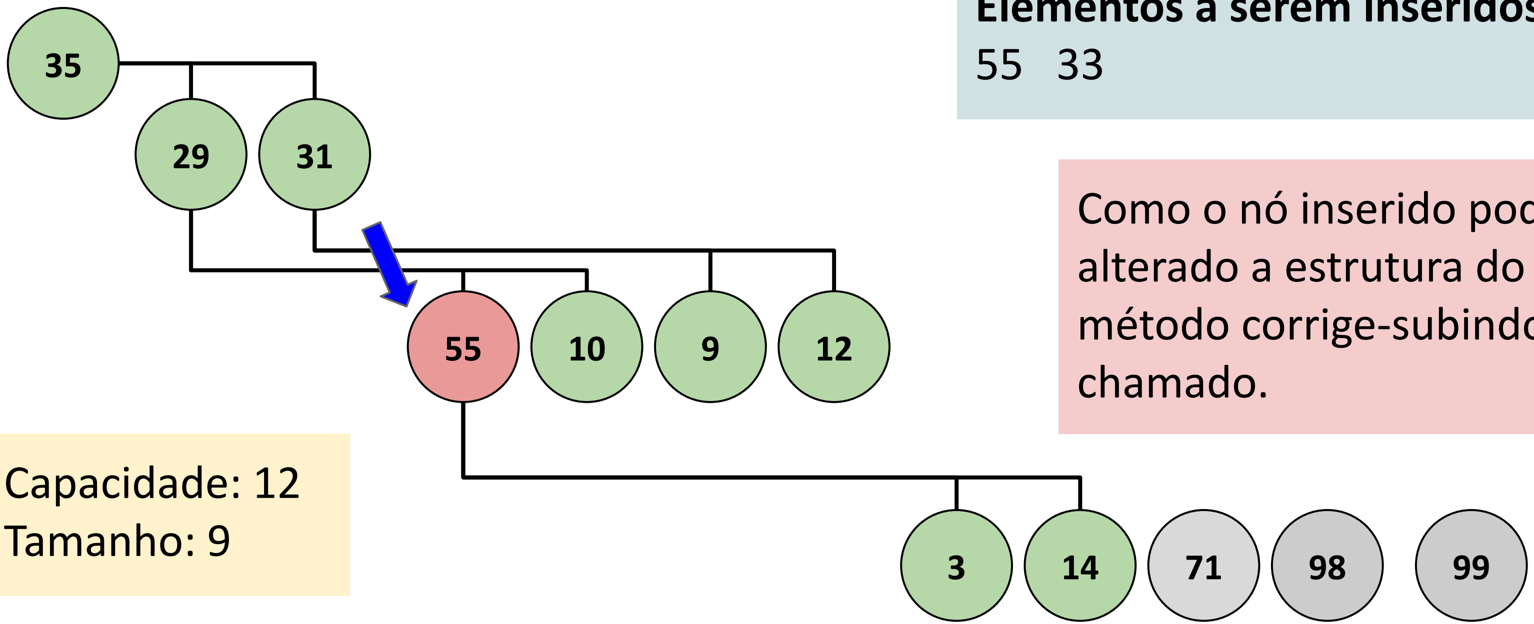
Capacidade: 12
Tamanho: 9

35	29	31	14	10	9	12	3	55	71	98	99

Exemplo 3 - V/XI

Elementos a serem inseridos:
55 33

Como o nó inserido pode ter alterado a estrutura do heap, o método corrige-subindo é chamado.



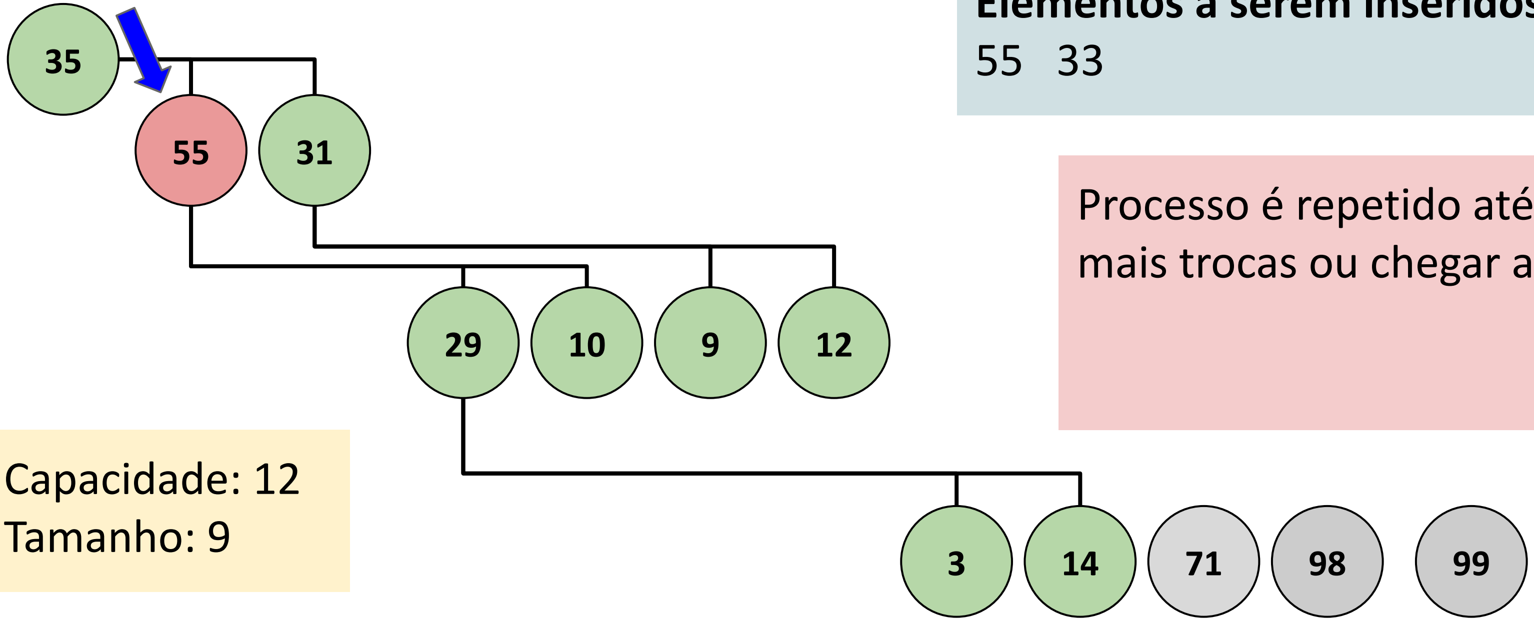
Capacidade: 12
Tamanho: 9

35	29	31	55	10	9	12	3	14	71	98	99

Exemplo 3 - VI/XI

Elementos a serem inseridos:
55 33

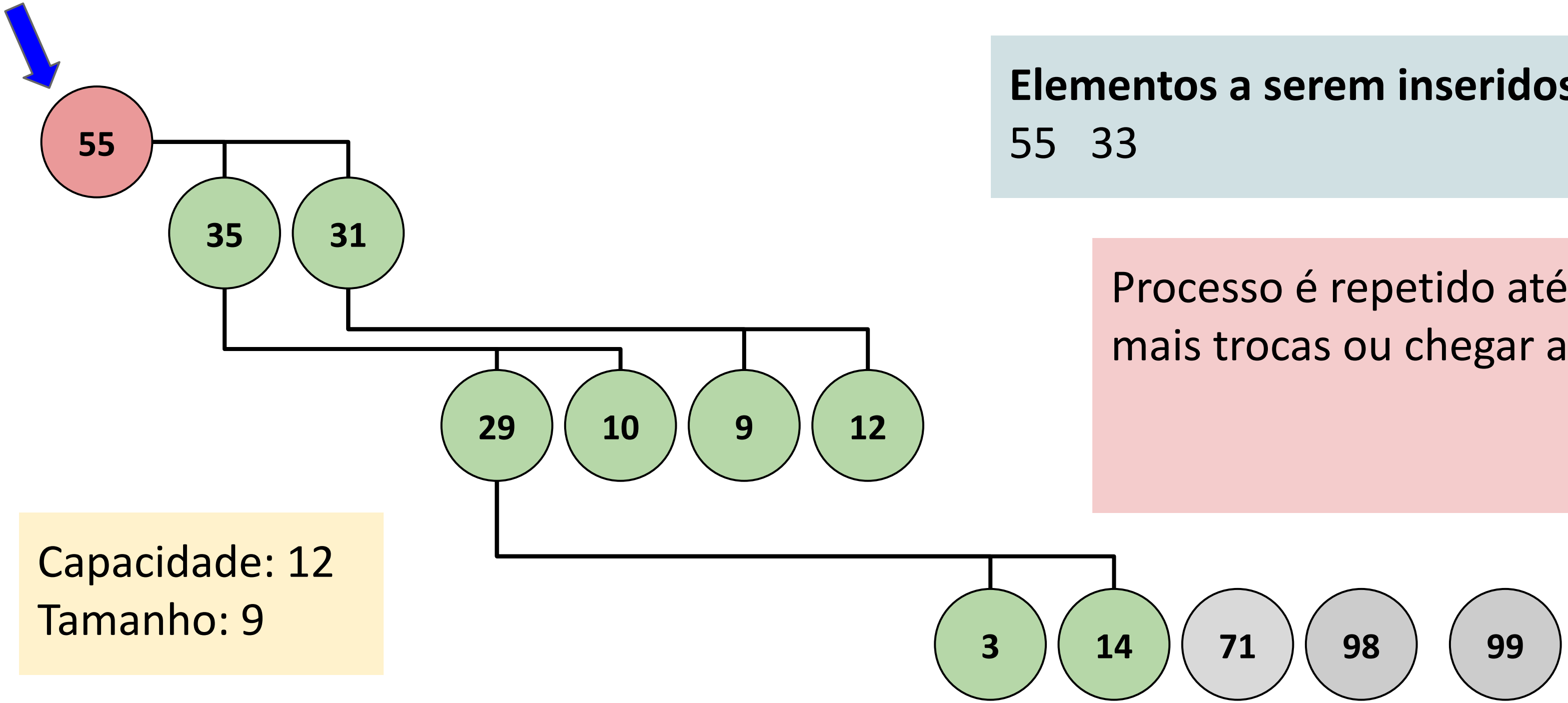
Processo é repetido até não haver mais trocas ou chegar ao raiz.



Capacidade: 12
Tamanho: 9

35	**55**	**31**	**29**	**10**	**9**	**12**	**3**	**14**	**71**	**98**	**99**

Exemplo 3 - VII/XI



Elementos a serem inseridos:
55   33

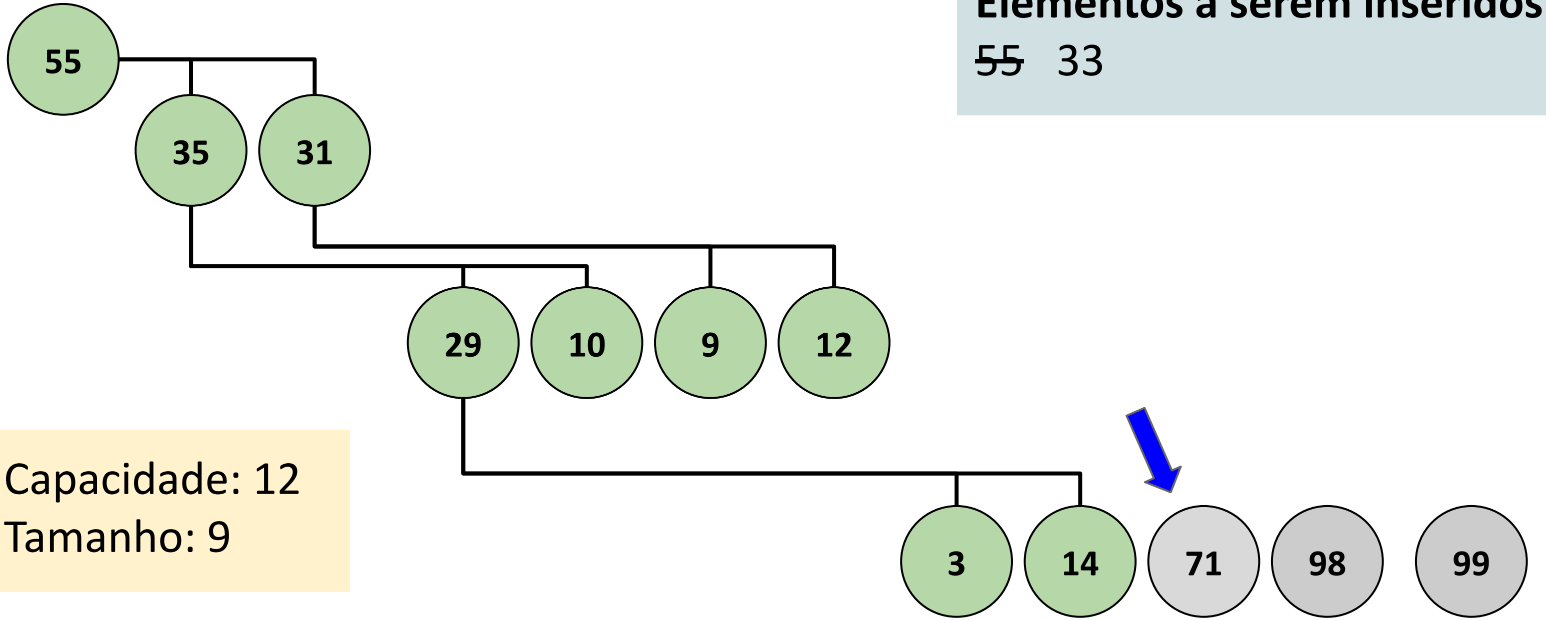
Processo é repetido até não haver mais trocas ou chegar ao raiz.

Capacidade: 12
Tamanho: 9

55	35	31	29	10	9	12	3	14	71	98	99

Exemplo 3 - VIII/XI

Elementos a serem inseridos:
~~55~~ 33

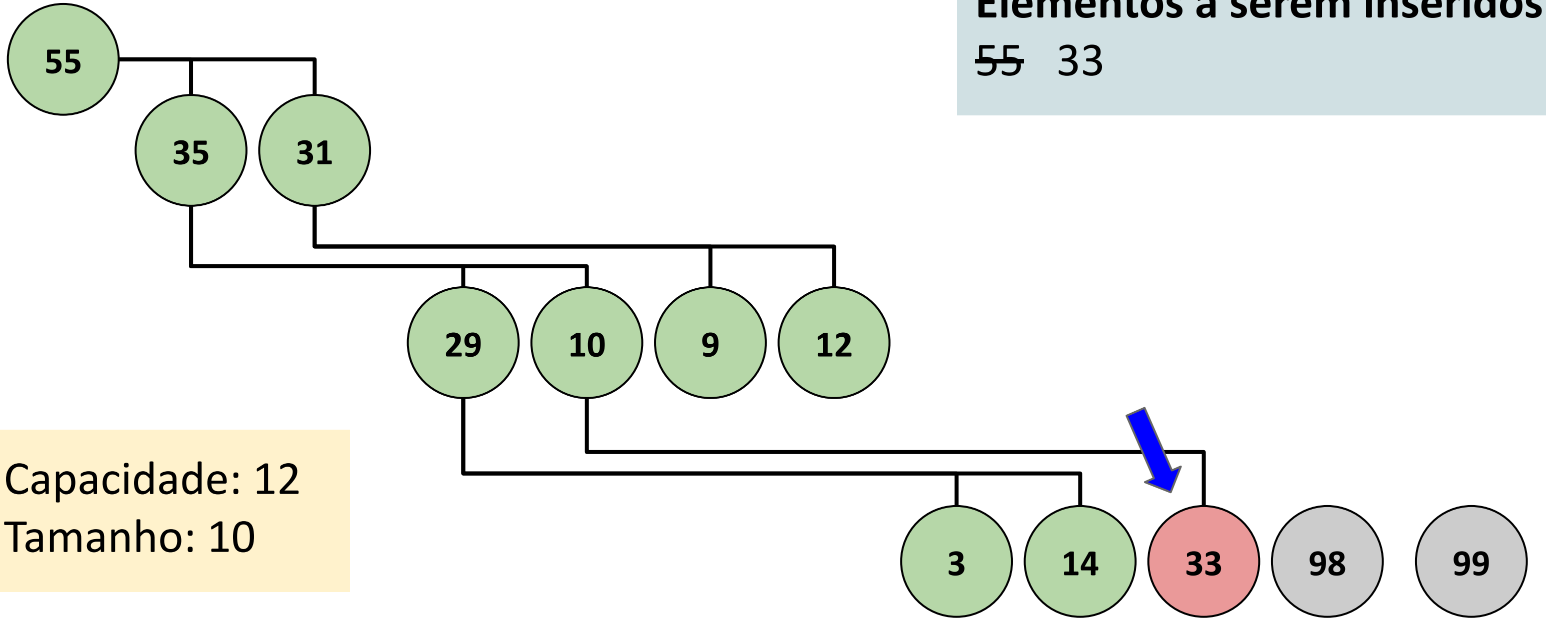


Capacidade: 12
Tamanho: 9

55	**35**	**31**	**29**	**10**	**9**	**12**	**3**	**14**	**71**	**98**	**99**

Exemplo 3 - IX/XI

Elementos a serem inseridos:
~~55~~ 33

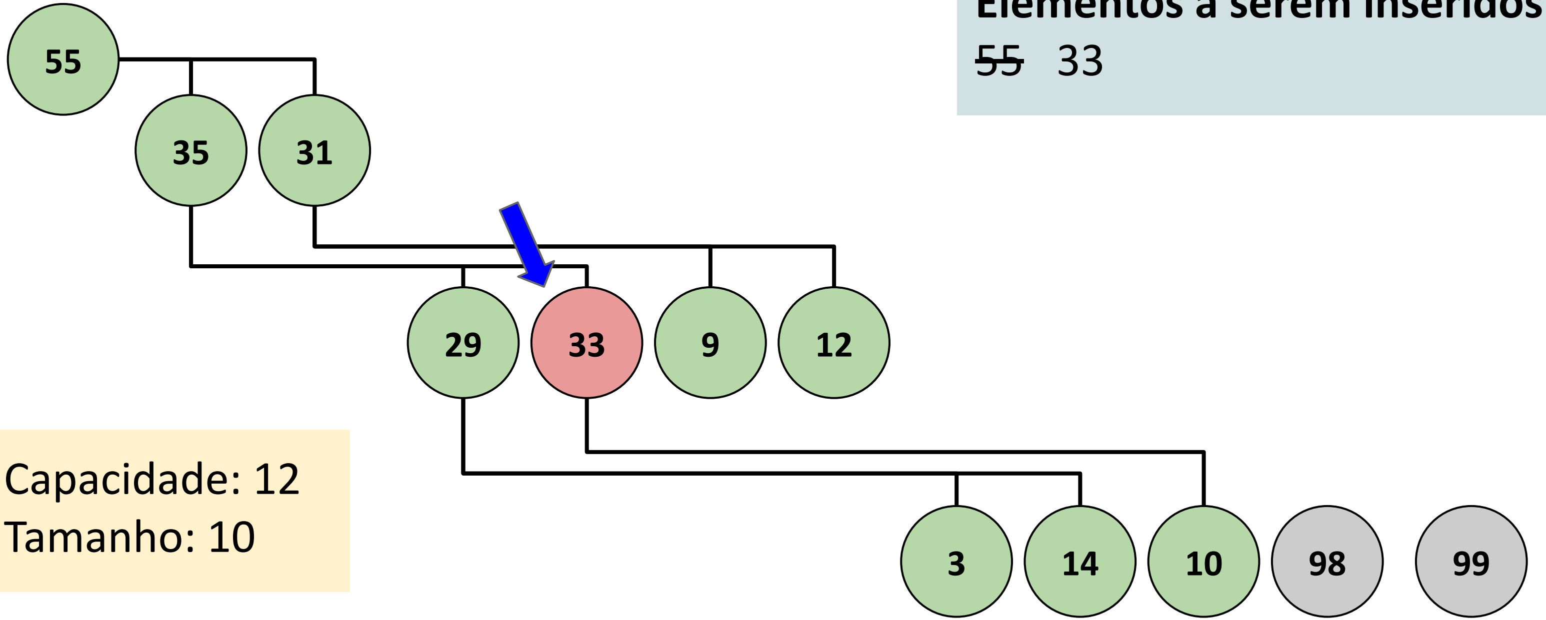


Capacidade: 12
Tamanho: 10

55	35	31	29	10	9	12	3	14	33	98	99

Exemplo 3 - X/XI

Elementos a serem inseridos:
~~55~~ 33



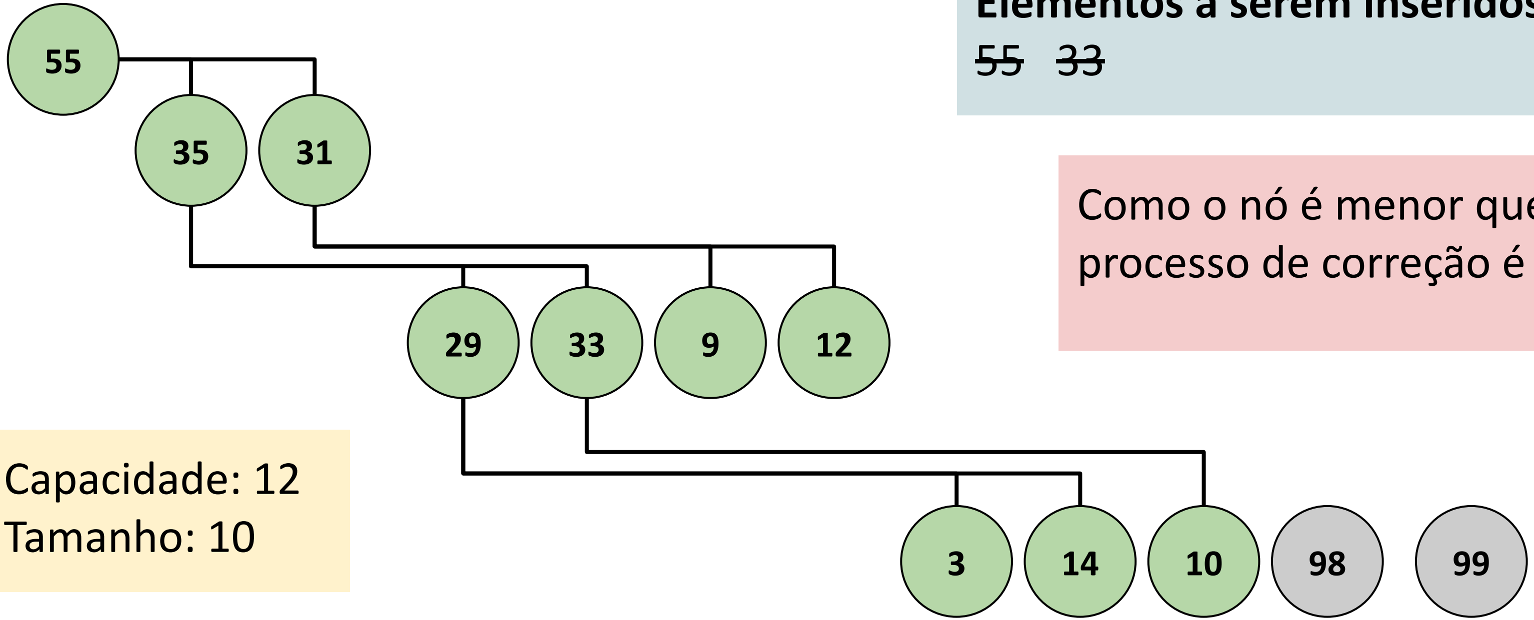
Capacidade: 12
Tamanho: 10

55	**35**	**31**	**29**	**33**	**9**	**12**	**3**	**14**	**10**	**98**	**99**

Exemplo 3 - XI/XI

Elementos a serem inseridos:
~~55~~ ~~33~~

Como o nó é menor que o nó pai, o processo de correção é terminado.



Capacidade: 12
Tamanho: 10

55	35	31	29	33	9	12	3	14	10	98	99

Implementação

Heap - Métodos para Implementação

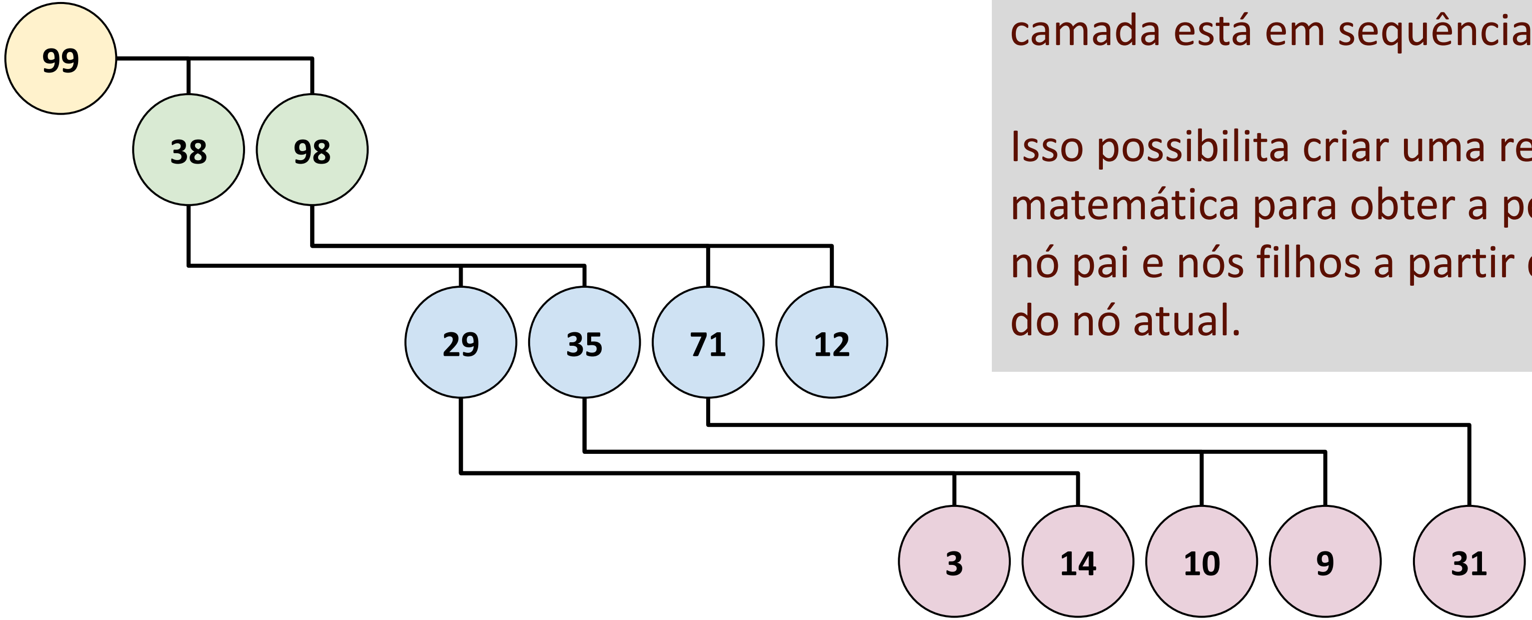
Como foi percebido, os métodos necessários a uma implementação de heap são: corrige-descendo(), corrige-subindo() (desnecessário se não suportar inserção), insere() e retira-raiz().

Quando da construção a partir de vetor, geralmente se utiliza um método auxiliar denominado geralmente de heapify(), constroi-heap() ou arruma().

Heap - Métodos Auxiliares

Como heaps são implementados tradicionalmente em arranjos, é necessários o uso de métodos auxiliares para encontrar o pai de um nó, bem como seu filho à esquerda e filho à direita.

Encontrando Pais e Filhos - I



99	38	98	29	35	71	12	3	14	10	9	31
0	1	2	3	4	5	6	7	8	9	10	11

Observe, no heap ao lado, que cada camada está em sequência no heap.

Isso possibilita criar uma relação matemática para obter a posição dos nó pai e nós filhos a partir da posição do nó atual.

Encontrando Pais e Filhos - II

Considerando-se arranjos começando em posição zero:

pai(i) ← (i-1)/2

esquerdo(i) ← 2i + 1

direito(i) ← 2i + 2

Encontrando Pais e Filhos - II

Algumas implementações de heap, começam a contagem na posição 1, nesse caso:

pai(i) ← (i)/2

esquerdo(i) ← 2i

direito(i) ← 2i + 1

o nó pai e o nó esquerdo são calculados com uma operação a menos.

Encontrando Pais e Filhos - II

Algumas implementações de heap, começam a contagem na posição 1, nesse caso:

pai(i) ← (i)/2

esquerdo(i) ← 2i

direito(i) ← 2i + 1

Quando começando com a posição 1, recomenda-se utilizar a posição 0 para armazenar o tamanho utilizado do heap.

Implementação de Heap - Visão Geral

A implementação de um heap é feita tradicionalmente em arranjos, definidos a partir de uma capacidade.

Adicionalmente, além dos dados no arranjo, é importante armazenar o tamanho utilizado.

Os exemplos de algoritmos a seguir utilizam heaps que, a princípio, podem começar da posição 0 ou 1 indistintamente.

Heap - Criação e Remoção

criarHeap(umaCapacidade):

```
capacidade ← umaCapacidade;
dados ← alocaVetorDeDados(capacidade);
tamanho ← 0;
```

destruirHeap():

```
desalocaVetorDeDados(dados);
```

Heap - Corrige Descendo

```
corrigeDescendo(i):
esq ← esquerdo(i);
dir ← direito(i);
maior ← i;
se ((esq <= FINAL) e (dados[esq] > dados[maior]))
    maior ← esq;
se ((dir <= FINAL) e (dados[dir] > dados[maior]))
    maior ← dir;
se (maior ≠ i) {
    troca(dados[i], dados[maior]);
    corrigeDescendo(maior);
}
```

Heap - Corrige Descendo

FINAL indica a posição final do heap e depende do tamanho (número de elementos no heap).

```
corrigeDescendo(i):
esq ← esquerdo(i);
dir ← direito(i);
maior ← i;
se ((esq <= FINAL) e (dados[esq] > dados[i]))
    maior ← esq;
se ((dir <= FINAL) e (dados[dir] > dados[maior]))
    maior ← dir;
se (maior ≠ i) {
    troca(dados[i], dados[maior]);
    corrigeDescendo(maior);
}
```

Heap - Corrige Descendo

```
corrigeDescendo(i):
esq ← esquerdo(i);
dir ← direito(i);
maior ← i;
se ((esq <= FINAL) e (dados[esq] > dados[i]))
    maior ← esq;
se ((dir <= FINAL) e (dados[dir] > dados[maior]))
    maior ← dir;
se (maior ≠ i) {
    troca(dados[i], dados[maior]);
    corrigeDescendo(maior);
}
```

FINAL indica a posição final do heap e depende do tamanho (número de elementos no heap).

`FINAL vale TAMANHO - 1, para heaps começando em 0 e vale TAMANHO para heaps começando em 1.`

Heap - Construção a partir de Vetor

```
constroiHeap(vetor,tamanho): //heapify()
dados ← copiaDados(vetor);
// INICIO informa posição inicial utilizada no vetor
para todo i de METADE até INICIO {
   corrigeDescendo(i);
}
```

Heap - Construção a partir de Vetor

```
constroiHeap(vetor,tamanho): //heapify()
dados ← copiaDados(vetor);
// INICIO informa posição inicial utilizada no vetor
para todo i de METADE até INICIO {
   corrigeDescendo(i);
}
```

METADE indica a posição do último nó não folha no heap.

`METADE = FINAL/2` para heaps começando em 1
`METADE = (FINAL-1)/2` para heaps começando em `0`

Heap - Construção a partir de Vetor

```
constroiHeap(vetor,tamanho): //heapify()
dados ← copiaDados(vetor);
// INICIO informa posição inicial utilizada no vetor
para todo i de METADE até INICIO {
    corrigeDescendo(i);
}
```

`FINAL vale TAMANHO – 1, para heaps começando em 0 e vale TAMANHO para heaps começando em 1.`

Considerando que o valor de FINAL depende se o armazenamento começa em 0 ou 1:

`METADE = TAMANHO/2` para heaps começando em 1
`METADE = (TAMANHO-2)/2` para heaps começando em `0`

Heap - Retirada da Raiz

<u>retiraRaiz():</u>

```
se (tamanho < 1)
   gerarErro(erroTamanho);
aux ← dados[INICIO];
troca(dados[INICIO],dados[FINAL]);
tamanho--;
corrigeDescendo(INICIO);
efetuaAcao(aux);
```

Heap - Corrige Subindo

```
corrigeSubindo(i):
p ← pai(i);
se ((p ≥ INICIO) e (dados[i] > dados[p])) {
   troca(dados[i],dados[p]);
   corrigeSubindo(p);
}
```

Heap - Corrige Subindo

```
corrigeSubindo(i):
p ← pai(i);
se ((p ≥ INICIO) e (da
    troca(dados[i],dados
    corrigeSubindo(p);
}
```

O teste (p ≥ INICIO) é desnecessário na maioria das linguagens de programação, caso se utilize heap começando em 0. Isso ocorre porque o pai da raiz vai ser calculado como:

raiz(0) = (0-1)/2 = -1/2.

A maioria das linguagens (incluindo C/C++) vai arredondar esse valor para 0 ao fazer divisão inteira.

Heap - Inserção

insere(valor):

```
se (tamanho = capacidade)
   geraErro(erroInsercao);
heap[FINAL+1] ← valor;
corrigeSubindo(FINAL+1);
tamanho++;
```

Heap - Outros Métodos

Considera-se quebra de estrutura acessar diretamente o arranjo de dados em um heap. Assim, um método para impressão dos dados só seria aceito em implementações didáticas ou para depuração.

Em algumas situações, é permitido o acesso (mas sem retirada) ao elemento raiz, espiando-o.

Um heap também pode ser utilizado para ordenar vetores, através do método heapsort.

Heapsort

O heapsort é um método de ordenação que consiste na construção de um heap a partir dos dados de um vetor. Pode-se usar um maxheap ou minheap, dependendo como os dados são trabalhados.

No heapsort, até que o heap fique vazio, os elementos do heap são retirados um por um, produzindo uma ordenação dos dados.

É possível encontrar vários algoritmos e implementações em que a estrutura fica implícita.

Torneio

Torneio - I

Um torneio é uma árvore estritamente binária na qual cada nó não folha (pai) contém uma cópia do maior elemento entre seus dois filhos.

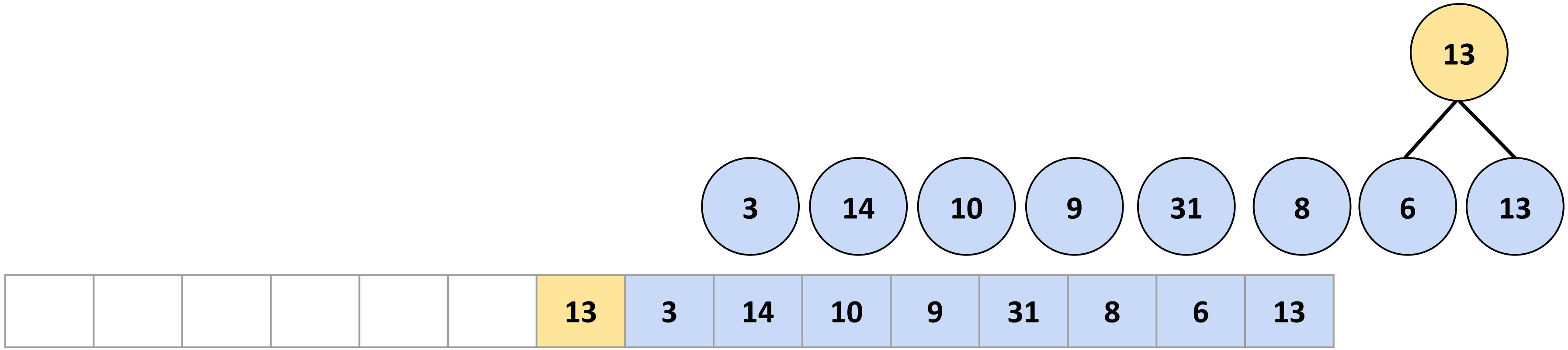
O conteúdo das folhas de um torneio determina o conteúdo de todos os seus nós.

Torneio - II

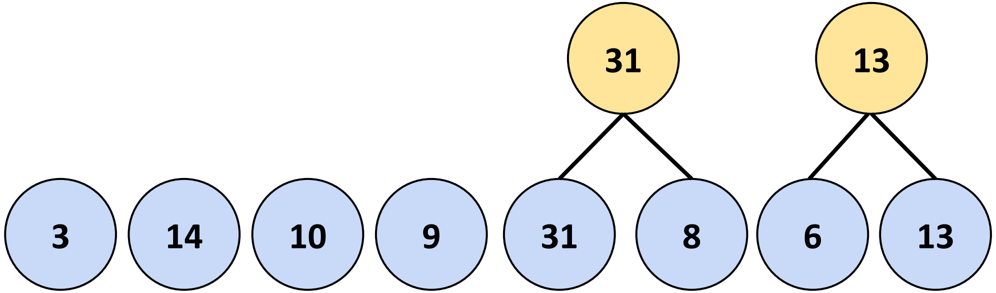
A representação de um torneio através de um MaxHeap permite identificar a classificação dos 1º e 2º lugares com facilidade, por exemplo.

A ordem de classificação é obtida retirando-se a raiz e reorganizando os elementos do torneio, mas mantendo a classificação parcial já realizada.

Torneio - III

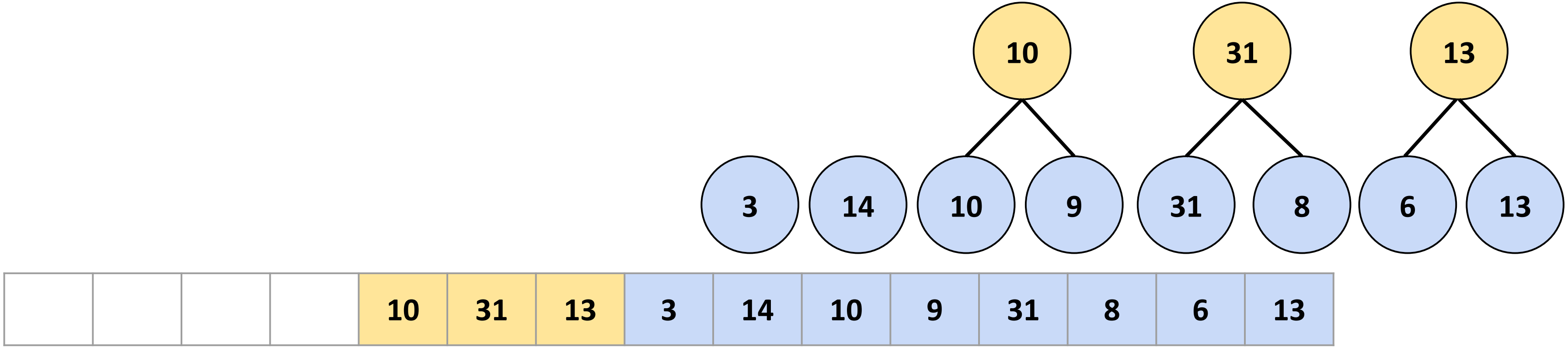


Torneio - IV

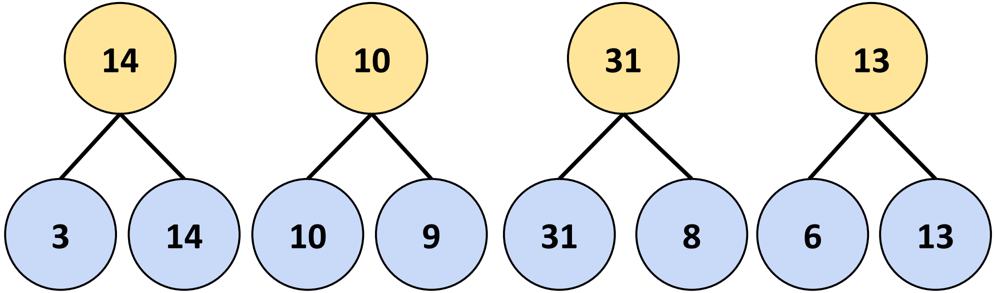


					31	13	3	14	10	9	31	8	6	13

Torneio - V

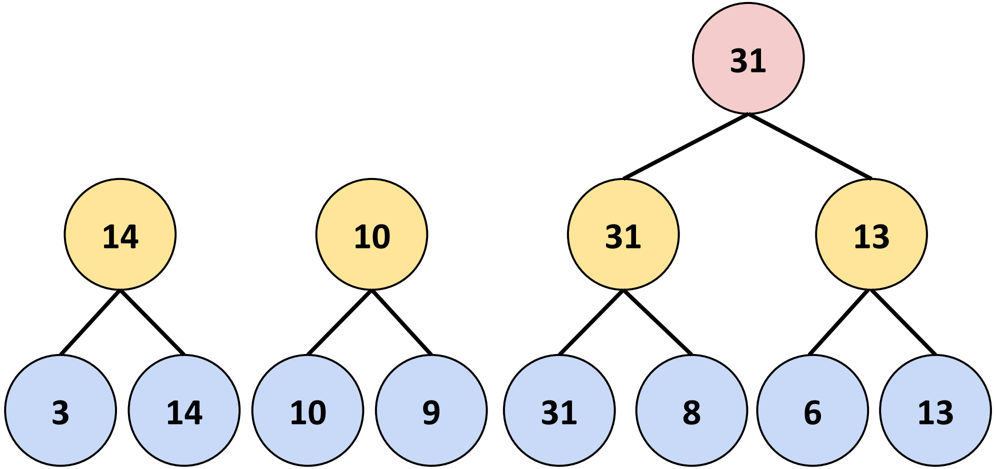


Torneio - VI



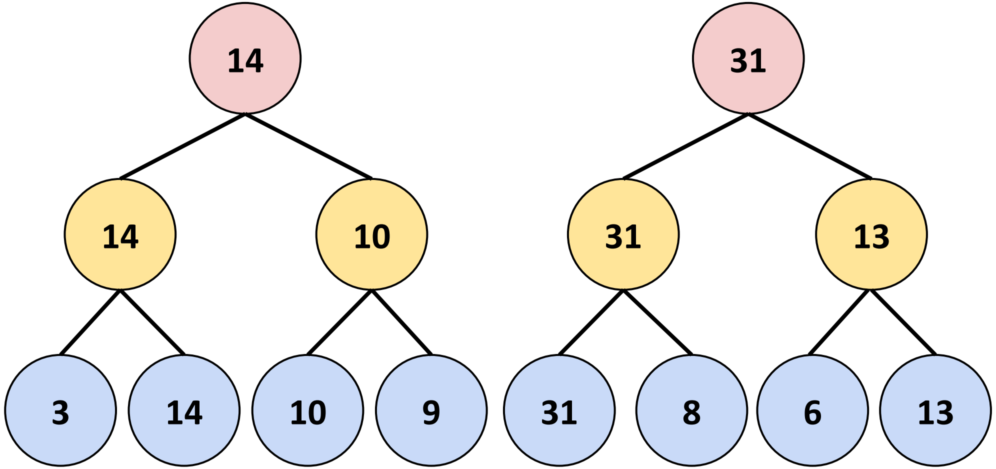
			14	10	31	13	3	14	10	9	31	8	6	13

Torneio - VII



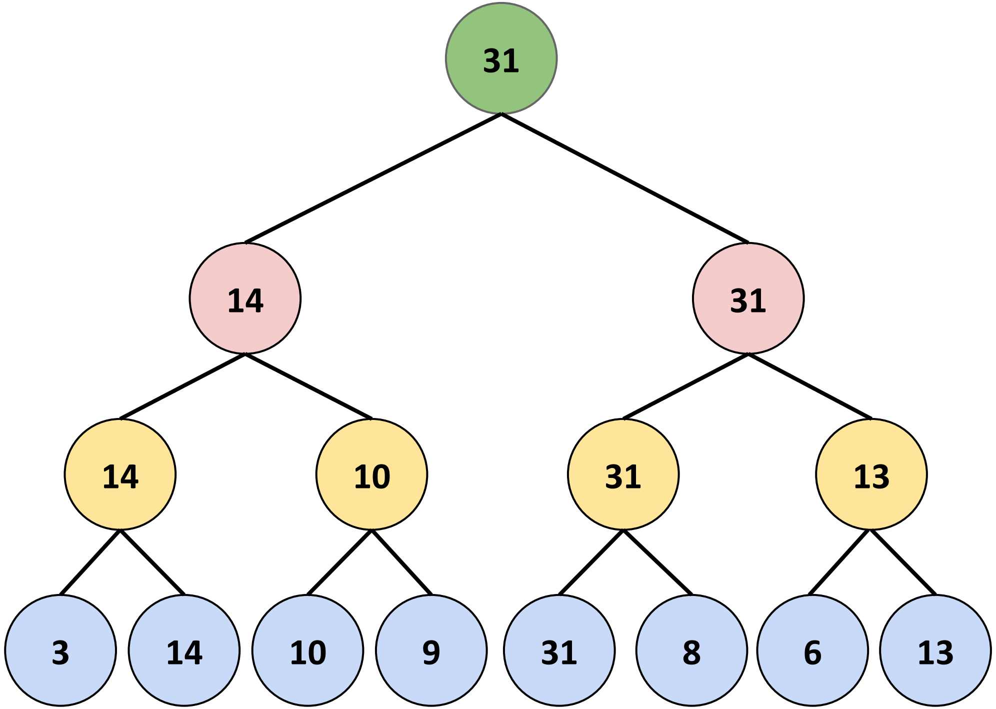
		31	14	10	31	13	3	14	10	9	31	8	6	13

Torneio - VIII



	14	31	14	10	31	13	3	14	10	9	31	8	6	13

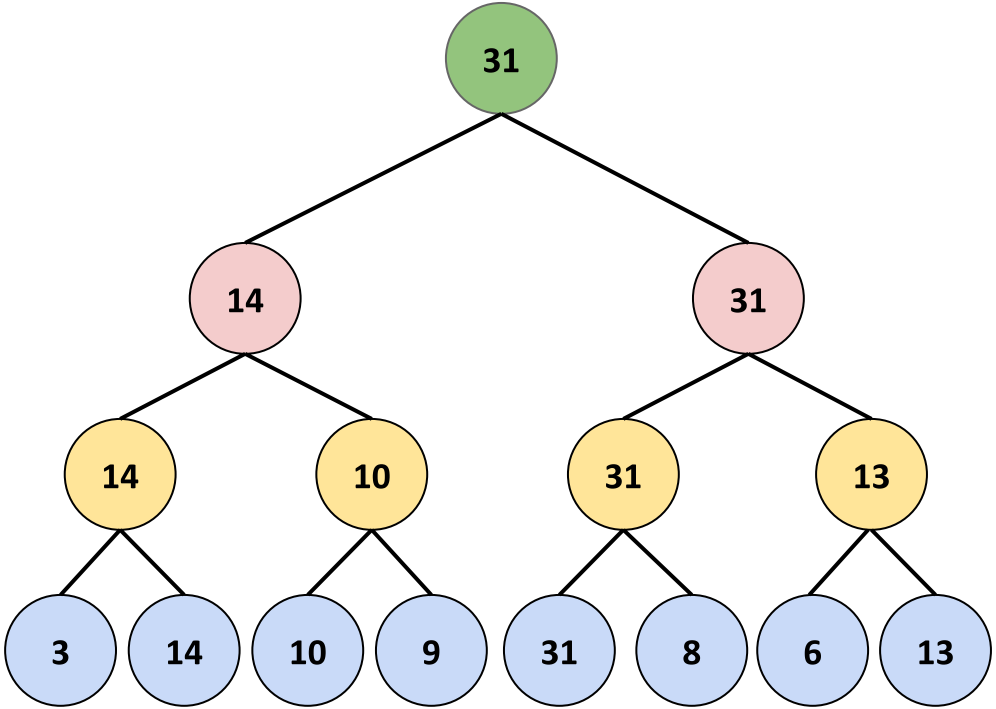
Torneio - IX



31	14	31	14	10	31	13	3	14	10	9	31	8	6	13

Torneio - X

É fácil apontar, após a construção, o campeão e o vice-campeão do torneio. O campeão está na raiz e o vice-campeão é o filho da raiz que não tem o mesmo valor.



31	14	31	14	10	31	13	3	14	10	9	31	8	6	13

Torneio Incompleto

Torneio Incompleto - I

Como pode ser percebido, fica bastante simples implementar um torneio quando os elementos representam uma potência de 2. O heap ocupa todas as posições possíveis de nós folha.

E quando não há elementos suficientes para ocupar todas as posições?

Torneio Incompleto - II

Como em um torneio os dados estão todos contidos nas folhas, a alocação do vetor de dados precisa alocar espaço para os dados e os possíveis nós nas camadas intermediárias.

Se as folhas estivessem todas cheias, a capacidade total a ser alocada seria $2^k - 1$. Nesse caso, os dados ocupariam metade do espaço, ou seja 2^{k-1}.

Torneio Incompleto - III

Assim o número de possíveis pais é dado pelo cálculo da potência de 2 menos uma unidade, tal que esse cálculo seja maior ou igual ao número de dados iniciais:

```
numPais ← 1;
enquanto (numPais < numDados)
  numPais ← numPais * 2;
capacidade ← numPais - 1  + numDados;
alocaVetorDeDados(capacidade);
inicioVetorDeDados ← capacidade - tam;
copiaVetorDeDados();
```

Torneio Incompleto - III

Assim o número de possíveis possíveis pais é dado pelo cálculo da potência de 2 menos uma unidade, tal que esse cálculo seja maior ou igual ao número de dados iniciais:

```
numPais ← 1;
enquanto (numPais < numDados)
  numPais ← numPais * 2;
capacidade ← numPais - 1  + numDados;
alocaVetorDeDados(capacidade);
inicioVetorDeDados ← capacidade - tam;
copiaVetorDeDados();
```

A cópia do vetor de dados é feita a partir do espaço reservado aos nós dos possíveis pais.

Torneio incompleto - IV

Após cópia dos dados, é necessário agora arrumar o torneio. A função arruma(), assim como no maxheap, consiste em chamar uma função auxiliar do meio até o início para ir corrigindo os elementos.

No caso do torneio, não há troca de valores, mas cópia. Além disso, como os dados estão apenas nas folhas, não há necessidade de descer aos níveis mais baixos como na corrigeDescendo() do heap. Assim, vamos chamar esse método de copiaMaior().

Método arruma()

<u>*arruma():*</u>

```
para todo i de inicioVetorDeDados até INICIO {
    copiaMaior(i);
}
```

Método copiaMaior()

```
copiaMaior(i):
esq ← esquerdo(i);
dir ← direito(i);
se (esq <= FINAL)
    se ((dir <= FINAL) e (dados[dir] > dados[esq]))
        maior ← dir;
    senão
        maior ← esq;
    heap[i] ← heap[maior];
senão
    heap[i] ← INVALIDO;
```

Método copiaMaior()

INVALIDO é uma forma de informar que o valor naquele nó não pode ser usado.
Caso o torneio seja apenas de valores positivos (o mais usual), basta configurar INVALIDO como -1.

copiaMaior(i):

```
esq ← esquerdo(i);
dir ← direito(i);
se (esq <= FINAL)
    se ((dir <= FINAL) e (dados[dir] > dados[esq]))
        maior ← dir;
    senão
        maior ← esq;
    heap[i] ← heap[maior];
senão
    heap[i] ← INVALIDO;
```

Torneio Incompleto - Exemplo (1)

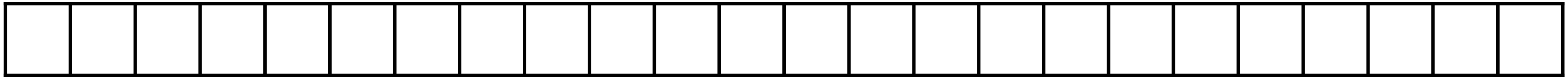
Considere o seguinte exemplo: 9 times vão competir pela copa de jokempô. A pontuação foi determinada por caracteres, em ordem crescente.

F	A	G	T	M	C	B	I	L

Torneio incompleto - Exemplo (2)

Para nove times, precisamos de um arranjo com 24 posições:
2^k (*potência de dois que comporta a quantidade de elementos*) - 1 + a quantidade de elementos

$$16 - 1 + 9 = 24$$



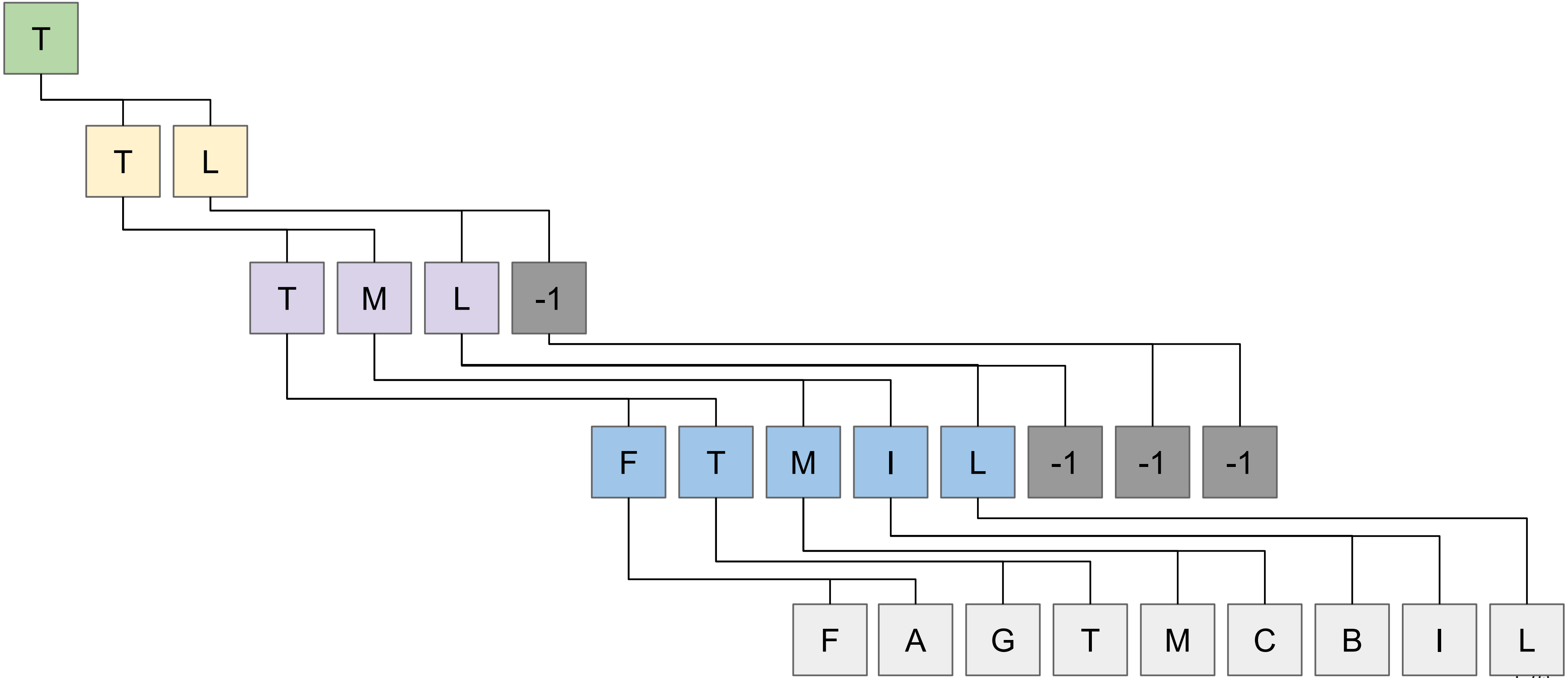
Torneio incompleto - Exemplo (3)

O preenchimento do arranjo começa na porção final, que representa o conjunto de folhas (ou filhos).

```
inicioVetorDeDados ← numPais;
copiaVetorDeDados();
```

														F	A	G	T	M	C	B	I	L

Torneio incompleto - Exemplo (4)



Torneio incompleto - Exemplo (5)

Após aplicação do método arruma(), o vetor de dados ficará da seguinte forma:

T	T	L	T	M	L	-1	F	T	M	I	L	-1	-1	-1	F	A	G	T	M	C	B	I	L

Suporte para Inserção - I

Usualmente, torneios são construídos a partir de dados em vetores.

Entretanto, apesar de um pouco mais trabalhoso, não é complicado adicionar inserção em torneios.

Nesse caso é necessário criar o vetor com a capacidade desejada (de maneira similar ao tamanho do vetor) e marcar posições não utilizadas como inválidas.

Suporte para Inserção - II

A inserção é feita sempre na primeira folha disponível, cuja posição é possível de encontrar usando tamanho e posição da primeira folha (inicioVetorDeDados).

Após a inserção, é necessário ir copiando o valor inserido até a raiz ou até encontrar um valor que seja maior que ele. O funcionamento é similar à corrigeSubindo(), mas sem trocas.

Método insere()

```
insere(valor):
se (tamanho = capacidade)
   geraErro(erroInsercao);
heap[tamanho+inicioDados] ← valor;
copiaSubindo(tamanho+inicioDados);
tamanho++;
```

Método copiaSubindo()

<u>copiaSubindo(i):</u>

```
p ← pai(i);
se ((dados[i] > dados[p])) {
    dados[p] ← dados[i];
    corrigeSubindo(p);
}
```

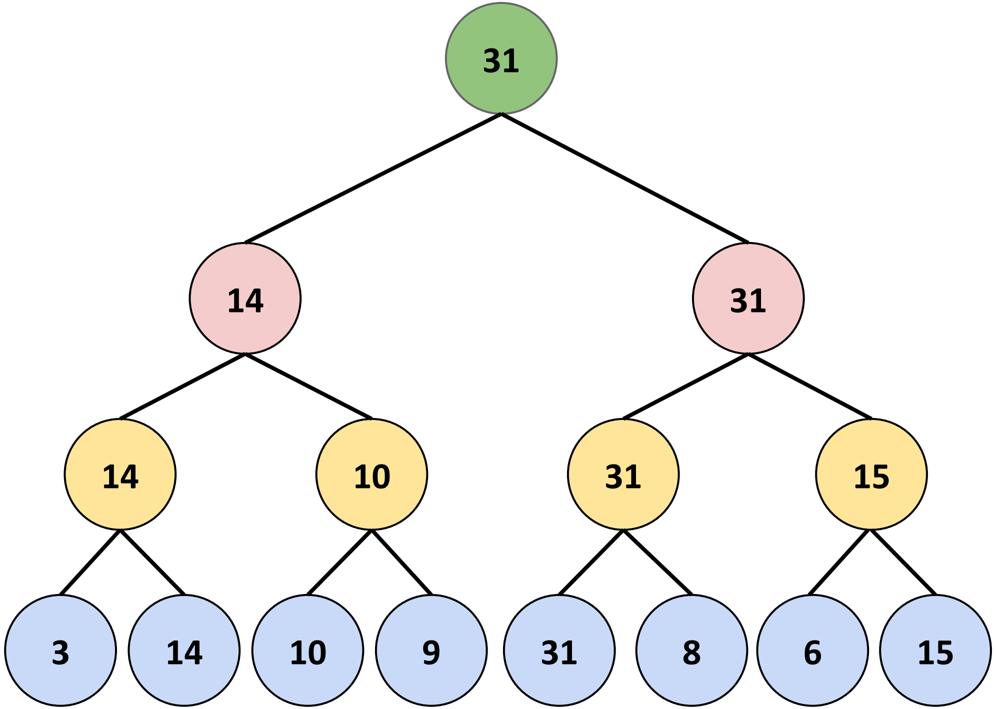
Retirada de Elementos - I

Em geral, torneios são produzidos para fácil obtenção do vencedor, sendo também fácil obter o segundo colocado. Assim, a maioria das aplicações de torneios não utilizam a retirada de elementos.

Caso seja necessário, é preciso avaliar o que essa retirada significa para o problema em questão, uma vez que diferentes caminhos podem ser seguidos após a retirada da raiz.

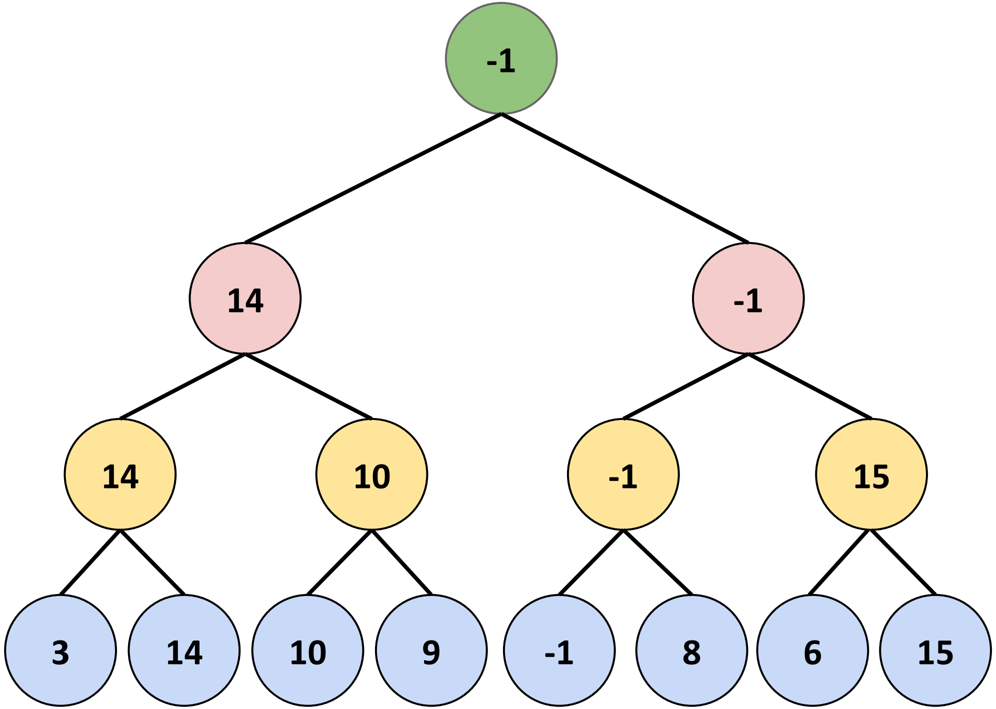
Retirada de Elementos - II

Por exemplo, no torneio ao lado, o segundo colocado é o 14, mas o segundo maior valor é o 15. Para que o 15 suba ao topo, caso seja esse o objetivo da retirada, é necessário refazer o torneio com os elementos restantes.



Retirada de Elementos - III

Caso seja necessário refazer o torneio, basta marcar os nós com o valor retirado como inválidos (-1 na figura) e reconstruir o torneio usando a função arruma().



Sobre o Material

Sobre este material

Material produzido coletivamente, principalmente pelos seguintes professores do DCC/UFLA:

- Joaquim Quinteiro Uchôa
- Juliana Galvani Greghi
- Renato Ramos da Silva

Inclui contribuições de outros professores do setor de Fundamentos de Programação do DCC/UFLA.