

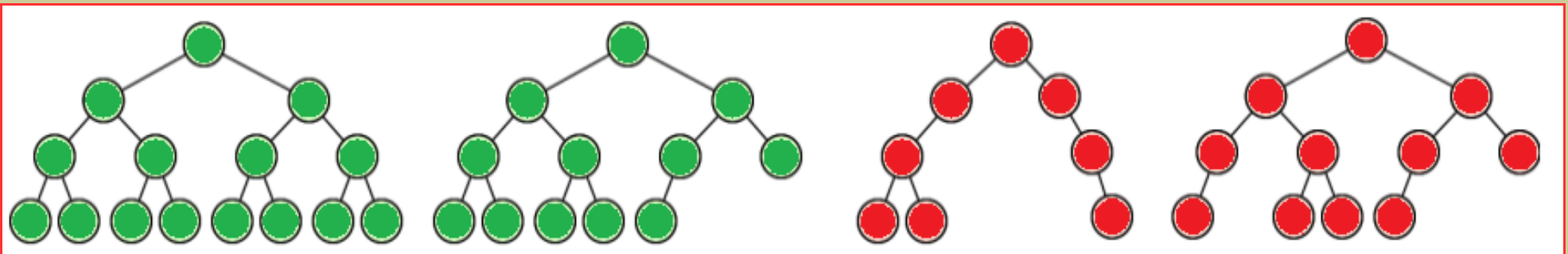
Filas de Prioridade

Heap Binário

Conceitos gerais

Árvore binária completa

- Uma **árvore binária completa** é uma árvore onde
 - todos os níveis (exceto possivelmente o último) estão totalmente preenchidos com nós, e
 - todos os nós estão o mais à esquerda possível
- Uma **árvore binária completa** é uma árvore muito balanceada
- Exemplos

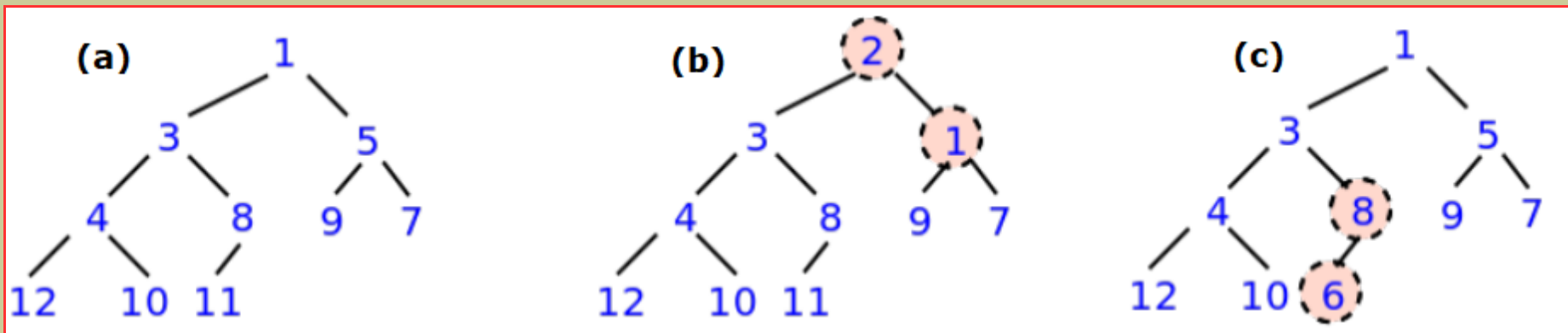


Árvores completas

Árvores não completas

Definição

- Um **Heap binário** é uma árvore com as seguintes propriedades
 - é uma árvore binária completa
 - todos os descendentes de um nó têm prioridade mais baixa do que esse nó
 - qualquer subárvore é um **Heap**
- Um **Heap binário** pode ser de dois tipos, conforme o elemento com *maior prioridade* tem associado o *menor* valor ou o *maior* valor
 - **minHeap**: o pai tem sempre *menor* valor que os seus descendentes
 - **maxHeap**: o pai tem sempre *maior* valor que os seus descendentes
- Exemplos: miniHeap **(a)** e árvore não Heap **(b)** e **(c)**



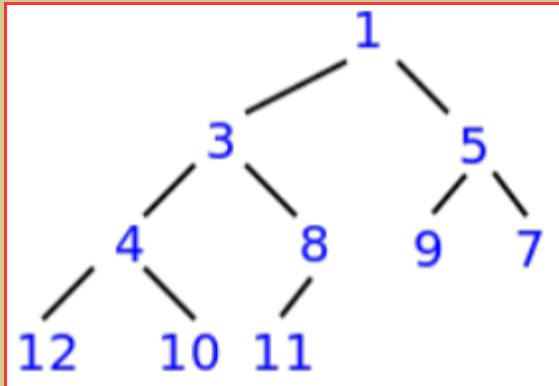
Implementação computacional

Mapeamento num array

- Maneira mais fácil de se implementar um **Heap** é usar um *array* que *implicitamente* representa a árvore
 - os elementos aparecem num *array* numa *ordem em largura* (de cima para baixo e da esquerda para a direita)
 - como a **raiz** está na **posição 1** do *array* (**índice 1**), então
 - os **filhos** do nó que está no **índice k** estão nos **índices 2.k** e **2.k+1**
 - o **pai** do nó que está no **índice k** está no **índice k/2** (divisão inteira)
 - **usando a linguagem C**, como a **raiz** está na **posição 1** do *array* (**índice 0**), então
 - os **filhos** do nó que está no **índice k** estão nos **índices 2.k+1** e **2.k+2**
 - o **pai** do nó que está no **índice k** está no **índice (k-1)/2** (divisão inteira)

Mapeamento num array

- Exemplo (linguagem C)



0	1	2	3	4	5	6	7	8	9
1	3	5	4	8	9	7	12	10	11

- os filhos do nó que está no **índice 2** (nó 5) estão nos **índices**
 - $2 \cdot 2 + 1 = 5$ (nó 9) e
 - $2 \cdot 2 + 2 = 6$ (nó 7)
- o pai do nó que está no **índice 5** (nó 9) é o nó que está no **índice**
 - $(5-1)/2 = 2$ (nó 5)
- Como a árvore é completa, então o *array* fica com **posições (índices)** consecutivas preenchidas

Operações auxiliares - minHeap

- **criar()**

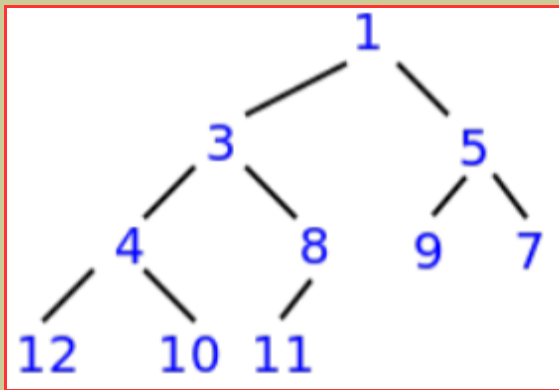
- cria um minHeap vazio: um *array* com 0 elementos

- **vazio(H)**

- verifica se o minHeap H é vazio: se o tamanho de um array é 0 ou não

Operação principal consultar(H) - minHeap

- Como cada nó tem valor menor (maior prioridade) que os seus filhos, o nó de menor valor está garantidamente na raiz da árvore (índice 0 do *array*)



0	1	2	3	4	5	6	7	8	9
1	3	5	4	8	9	7	12	10	11

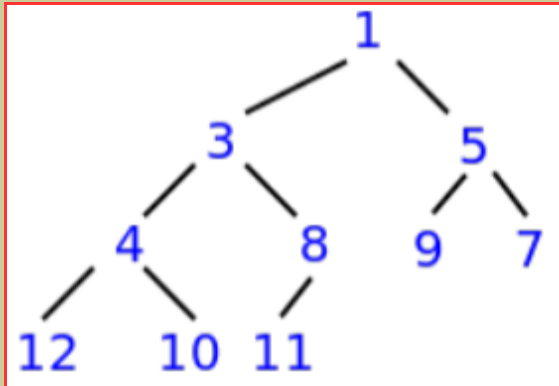
- Ordem de complexidade: $O(1)$
- Operação que devolve o elemento na posição 1 (índice 0) do array

Operação principal remover(H) - minHeap

- Extrair o elemento que se encontra na raiz da árvore (posição 0 do *array*)
- Remover o nó que está na raiz da árvore (posição 0 do *array*)
 - copiar para a posição 0 (nó raiz) o valor do último nó do *array* (e último nó da árvore, que é o nó mais à direita do último nível)
 - remover aquele nó do *array* (último nó da árvore) – a árvore continua completa
 - repor as propriedades de Heap:
 - R** ← o elemento da posição 0 do Heap (ou seja, a raiz)
 - se R** é uma folha **ou** tem menor valor (maior prioridade) que os seus filhos **então**
terminar
 - senão**
 - J** ← o filho de R com menor valor (ou seja, o com maior prioridade)
 - trocar a informação do nó **R** pela do nó **J**
 - repor as propriedades de Heap à subárvore com raiz em **J**
- Ordem de complexidade: $O(\log n)$

Operação principal remover(H) - minHeap

- Exemplo

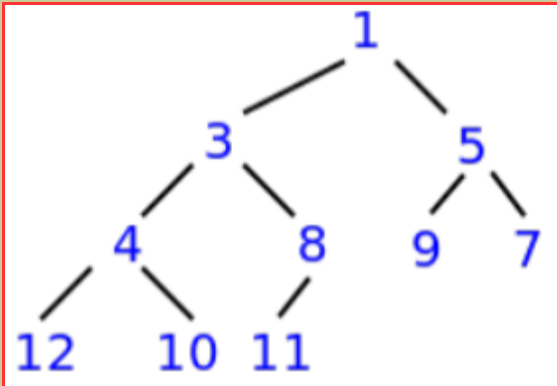


0	1	2	3	4	5	6	7	8	9
1	3	5	4	8	9	7	12	10	11

- retirar o valor do nó raiz (valor **1**)
- copiar o valor do último nó da árvore (com valor 11) para o nó raiz, e
- remover este nó (último da árvore) da árvore
- repor as *propriedades de Heap*, descendo o elemento da raiz (agora com o **11**)

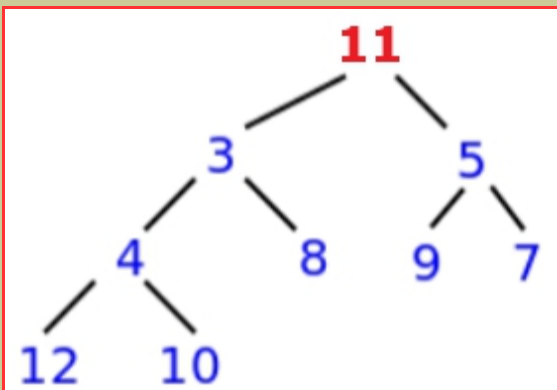
Operação principal remover(H) - minHeap

- Exemplo (cont.)



0	1	2	3	4	5	6	7	8	9
1	3	5	4	8	9	7	12	10	11

- retirar o valor do nó raiz (valor **1**)
- copiar o valor do último nó da árvore (com valor 11) para o nó raiz, e
- remover este nó (último da árvore) da árvore

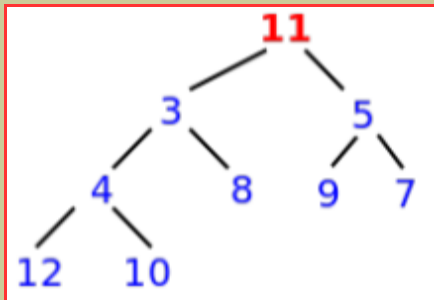


0	1	2	3	4	5	6	7	8
11	3	5	4	8	9	7	12	10

Operação principal `remove(H)` - minHeap

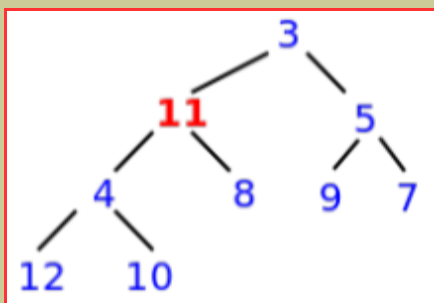
- Exemplo (cont.)

- repor as *propriedades de Heap*, descendo o elemento da raiz



0	1	2	3	4	5	6	7	8
11	3	5	4	8	9	7	12	10

- **11** ($H[0]$) troca com o menor dos seus filhos (**3** = $H[2 \times 0 + 1]$ e **5** = $H[2 \times 0 + 2]$), pois é maior do que o menor deles



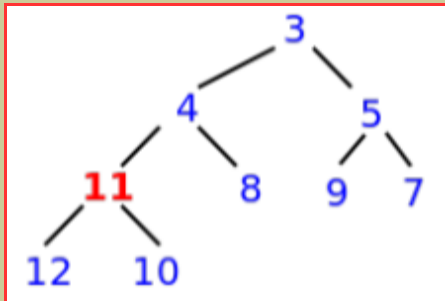
0	1	2	3	4	5	6	7	8
3	11	5	4	8	9	7	12	10

- **11** ($H[1]$) troca com o menor dos seus filhos (**4** = $H[2 \times 1 + 1]$ e **8** = $H[2 \times 1 + 2]$), pois é maior do que o menor deles

Operação principal `remove(H)` - minHeap

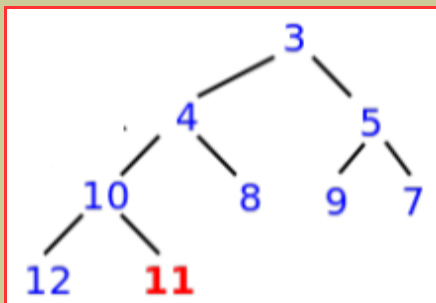
- Exemplo (cont.)

- repor as *propriedades de Heap*, descendo o elemento da raiz



0	1	2	3	4	5	6	7	8
3	4	5	11	8	9	7	12	10

- **11** ($H[3]$) troca com o menor dos seus filhos (**12** = $H[2 \times 3 + 1]$ e **10** = $H[2 \times 3 + 2]$), pois é maior do que o menor deles



0	1	2	3	4	5	6	7	8
3	4	5	10	8	9	7	12	11

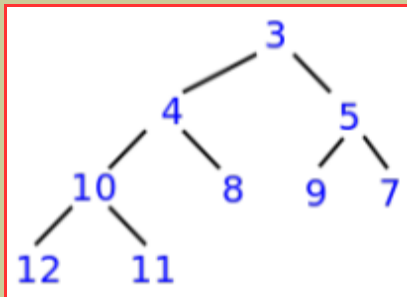
Operação principal inserir(x, H) - minHeap

- Inserir o novo nó na última posição livre na árvore (posição imediatamente a seguir à última posição), que é a primeira livre do *array* - a árvore continua completa
- Repor as propriedades de Heap, “subindo” o novo nó
 - **enquanto** o novo nó não atingir a raiz **e** o seu pai for de menor prioridade (maior valor, no caso de minHeap), **fazer** o seguinte:
 - trocar o novo nó com o seu pai
 - atualizar o novo nó, passando a ser o seu pai
- Ordem de complexidade: $O(\log n)$

Operação principal inserir(x, H) - minHeap

- Exemplo

- inserir um novo nó com o valor **2** no seguinte minHeap:



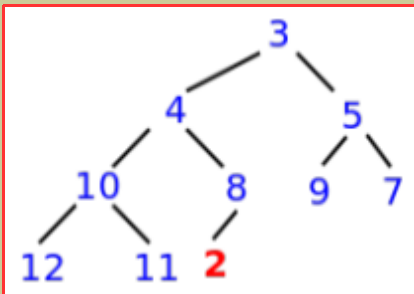
0	1	2	3	4	5	6	7	8
3	4	5	10	8	9	7	12	11

- inserir o novo elemento (**2**) na última posição da Heap
 - alocar memória para mais um elemento do array, e
 - colocar o valor **2** na última posição do array
- repor as *propriedades de Heap*
 - subir o elemento na árvore até encontrar um pai com menor valor

Operação principal inserir(x, H) - minHeap

- Exemplo

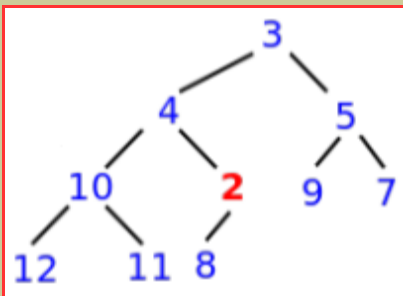
- inserir o novo elemento (**2**) na última posição da Heap



0	1	2	3	4	5	6	7	8	9
3	4	5	10	8	9	7	12	11	2

- repor as *propriedades de Heap*

- **2** (H[9]) troca com o seu pai (**8** = H[4], $4 = (9-1)/2$), pois é menor do que ele



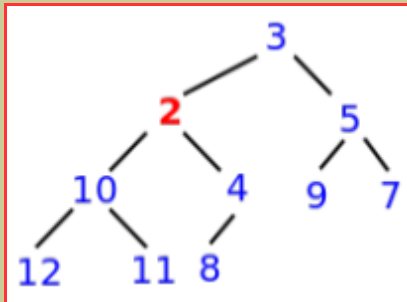
0	1	2	3	4	5	6	7	8	9
3	4	5	10	2	9	7	12	11	8

- **2** (H[4]) troca com o seu pai (**4** = H[1], $1 = (4-1)/2$), pois é menor do que ele

Operação principal inserir(x, H) - minHeap

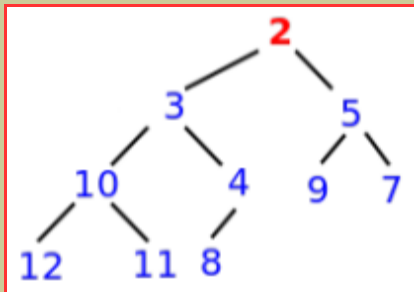
- Exemplo

- repor as *propriedades de Heap*



0	1	2	3	4	5	6	7	8	9
3	2	5	10	4	9	7	12	11	8

- **2** (H[1]) troca com o seu pai (**3** = H[0], $0 = (1-1)/2$), pois é menor do que ele



0	1	2	3	4	5	6	7	8	9
2	3	5	10	4	9	7	12	11	8

Eficiência computacional

Ordens de complexidade das operações principais

Estrutura Abstrata de Dados	inserir	remover	consultar
Lista não ordenada (array)	$O(1)$	$O(n)$	$O(n)$
Lista ordenada (array)	$O(n)$	$O(1)$	$O(1)$
Lista não ordenada (listas ligadas)	$O(1)$	$O(n)$	$O(n)$
Lista ordenada (listas ligadas)	$O(n)$	$O(1)$	$O(1)$
Árvore binária pesquisa	$O(n)$	$O(n)$	$O(n)$
Árvore binária pesquisa balanceada	$O(\log n)$	$O(\log n)$	$O(\log n)$
Heap binário	$O(\log n)$	$O(\log n)$	$O(1)$
Heap binomial			
Heap de Fibonacci			

Outros algoritmos

HeapSort

- Um Heap binário sugere um **algoritmo de ordenação**
- Para ordenar **n** elementos, basta fazer o seguinte
 - criar um Heap com os **n** elementos
 - retirar um a um os **n** elementos do *Heap* e colocá-los num *array*
- Como os elementos saem por ordem de prioridade, vão aparecer no *array* por ordem crescente
- Este processo implica **n** inserções, seguidas de **n** remoções
 - como o custo de cada operação é logarítmico,
 - então o custo total será de $O(n \cdot \log n)$
- Este algoritmo (na sua essência) é conhecido como **HeapSort**