

SCC0202 – Algoritmos e Estruturas de Dados I

Filas de prioridade e Heaps

Prof.: Dr. Rudinei Goularte

(rudinei@icmc.usp.br)

Instituto de Ciências Matemáticas e de Computação - ICMC

Sala 4-229

Conteúdo



- TAD Fila de Prioridade
- Heaps
- Implementação em Arranjo

Fila de Prioridade

- Relembrando:
 - ▣ **Pilhas e filas** => Elementos processados em função da sequência na qual foram inseridos
- **Filas de Prioridade**
 - ▣ Elementos são processados de acordo com sua importância, independente do momento em que entraram na fila
 - ▣ Exemplos:
 - atendimento preferencial em estabelecimentos em geral
 - importância de processos em sistemas operacionais
 - substituição de páginas menos utilizadas na memória
 - filas de atendimento para análise de sinais em sistemas de controle (sensores prioritários)

TAD Fila de Prioridade

- Armazena Itens
- Item: par (chave, informação)
- Operações principais
 - ▣ desenfileirar(F): remove e retorna o item com maior (menor) prioridade da fila F
 - ▣ enfileirar(F, x): insere um item $x = (k, i)$ com chave k
- Operações auxiliares
 - ▣ proximo(F): retorna o item com maior (menor) chave da fila F, sem removê-lo
 - ▣ vazia(F), cheia(F)

TAD Fila de Prioridade

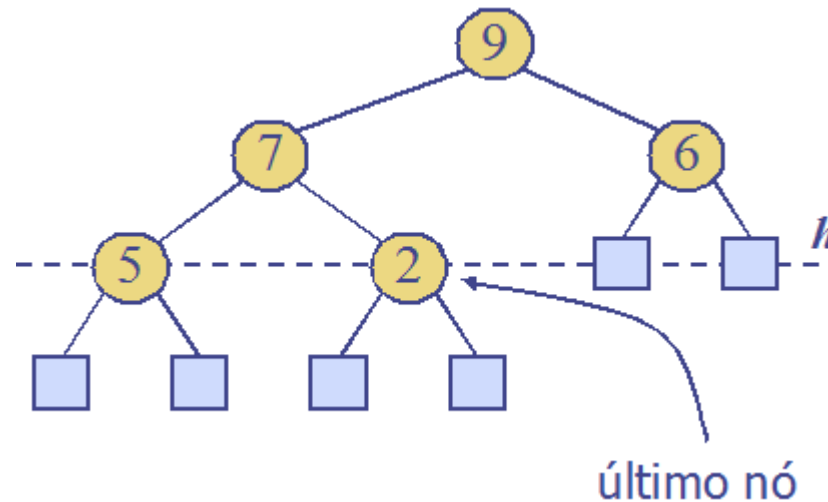
- Diferentes Realizações (implementações)
 - ▣ Estáticas
 - Lista sequencial (arranjo) ordenada
 - Lista sequencial (arranjo) não ordenada
 - **Heap em arranjo**
 - ▣ Dinâmicas
 - Lista encadeada ordenada
 - Lista encadeada não ordenada
 - Heap encadeada
- Cada realização possui vantagens e desvantagens

TAD Fila de Prioridade

- Uma das escolhas diretas seria usar uma lista ordenada
 - ▣ inserção é $O(n)$
 - ▣ remoção é $O(1)$
 - ▣ próximo é $O(1)$
- Outra seria usar uma fila (será?) ou lista não-ordenada
 - ▣ inserção é $O(1)$
 - ▣ remoção é $O(n)$
 - ▣ próximo é $O(n)$
- Portanto uma abordagem mais rápida precisa ser pensada quando grandes conjuntos de dados são considerados

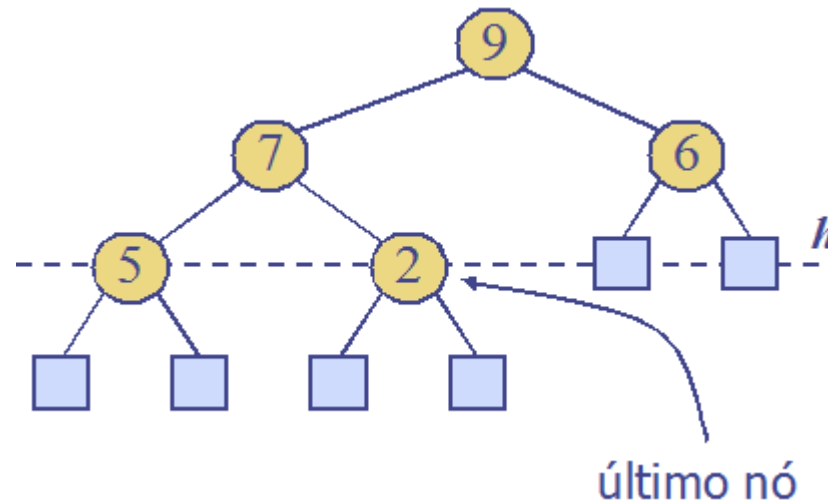
Heaps

- Uma heap é uma árvore binária que satisfaz as propriedades
 - ▣ **Ordem:** para cada nó v , exceto o nó raiz, tem-se que
 - $\text{chave}(v) \leq \text{chave}(\text{pai}(v))$ - heap máxima
 - $\text{chave}(v) \geq \text{chave}(\text{pai}(v))$ - heap mínima



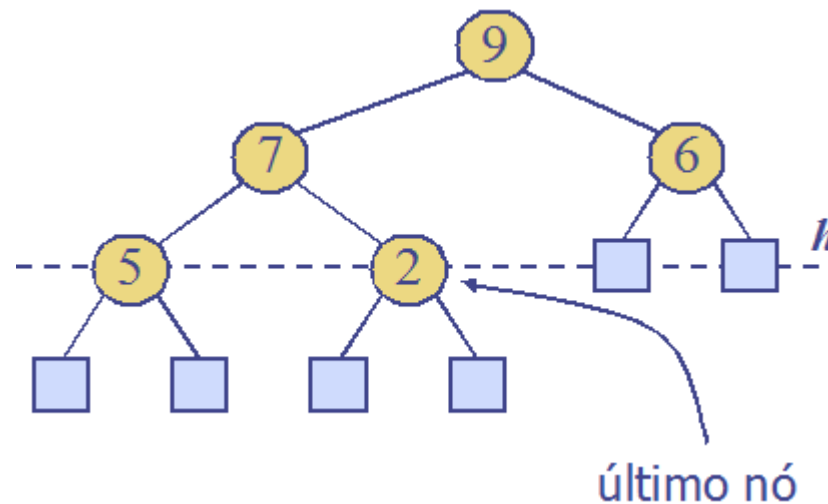
Heaps

- Uma heap é uma árvore binária que satisfaz as propriedades
 - ▣ **Completeness**: é completa, i.e., se h é a altura
 - Todo nó folha está no nível h ou $h - 1$
 - O nível $h - 1$ está totalmente preenchido
 - As folhas do nível h estão todas mais a esquerda



Heaps

- Convenciona-se aqui
 - ▣ Último nó: nó interno mais à direita de profundidade h
 - ▣ (está no nível h , último nível)

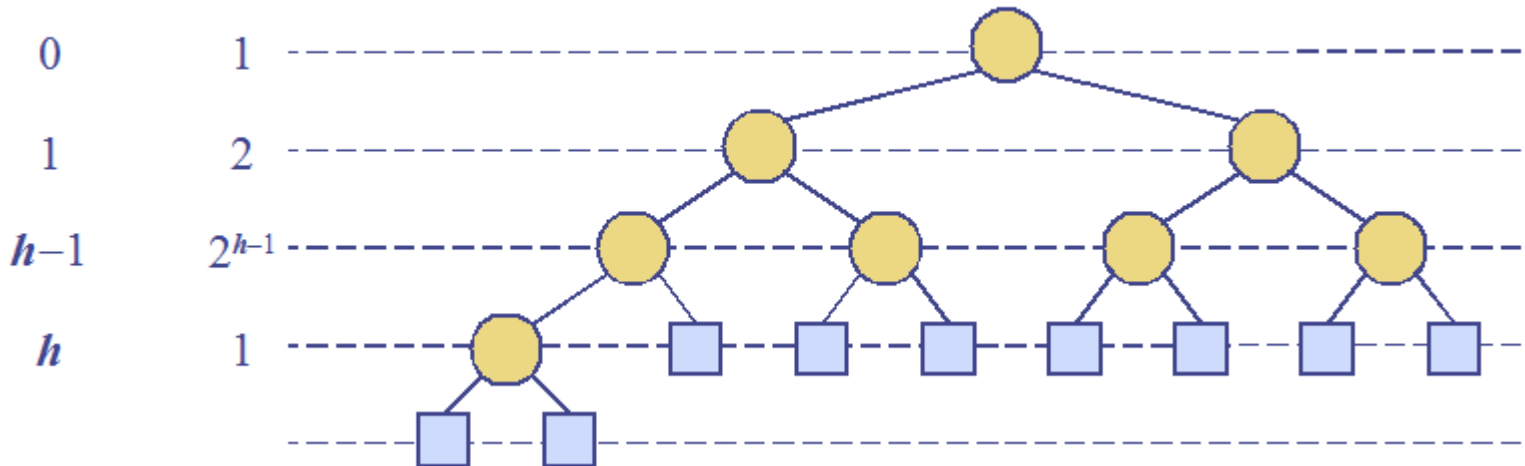


Altura de uma Heap

□ Teorema

- ▣ Uma heap armazenando n nós possui altura h de ordem $O(\log n)$.

prof. no. chaves



Altura de uma Heap

□ Prova

▣ Dado que existem 2^i chaves na profundidade $i = 0, \dots, h-1$ e ao menos 1 chave na profundidade h , tem-se:

$$n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$$

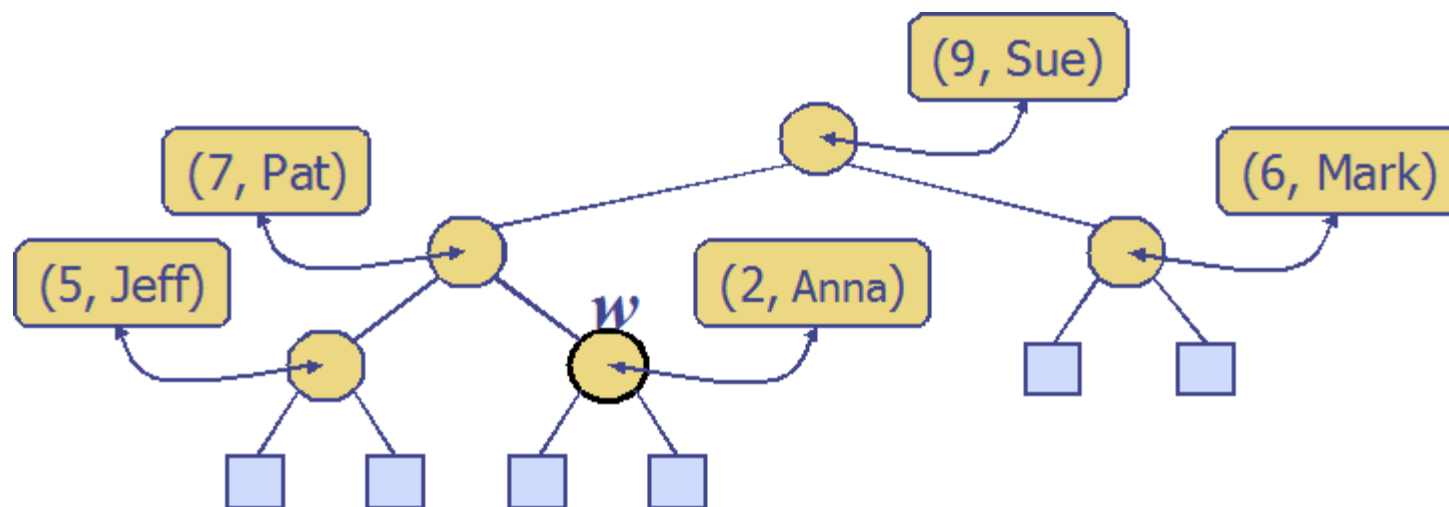
□ Isso é uma Progressão Geométrica (PG) com razão $q = 2$. Dado que a soma de um PG pode ser calculada por

$$S_k = \frac{a^k \times q - a_1}{q - 1}, \text{ temos } n \geq (2^{h-1} \times 2 - 1) + 1 = 2^h$$

□ Logo, $n \geq 2^h$, i.e., $h \leq \log_2 n \rightarrow h \text{ é } O(\log n)$

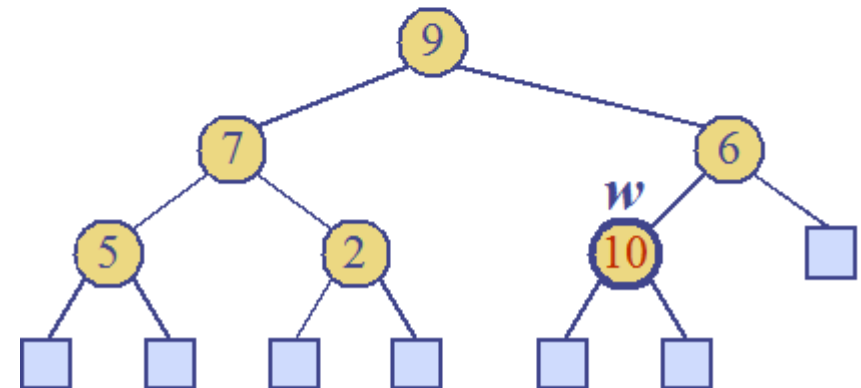
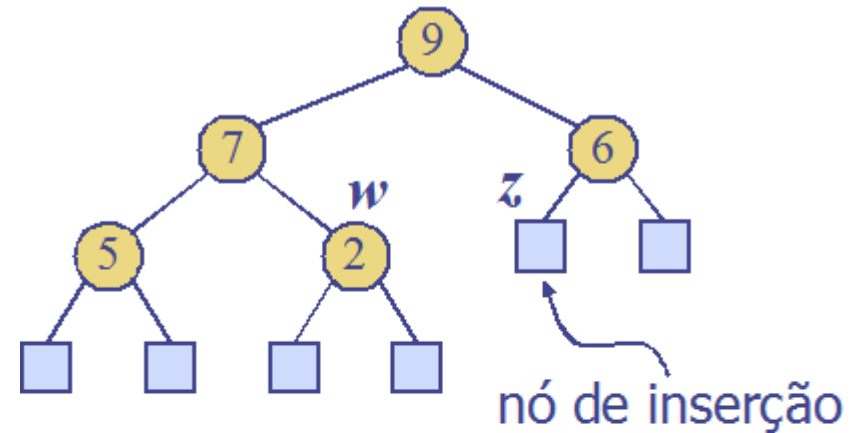
Filas de Prioridade com Heaps

- Armazena-se um Item (chave, informação) em cada nó
- Mantém-se o controle sobre a localização do último nó (w)
- Remove-se sempre o Item armazenado na raiz, devido à propriedade de ordem da heap
 - ▣ Heap mínima: menor chave na raiz da heap
 - ▣ Heap máxima: maior chave na raiz da heap



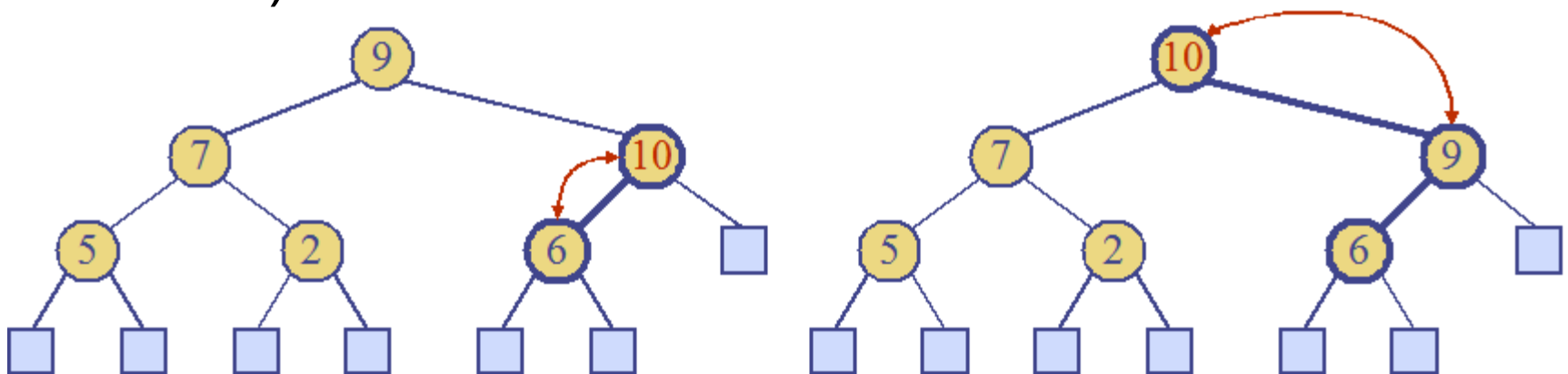
Inserção

- Método insere do TAD fila de prioridade corresponde à inserção de um Item na heap
- O algoritmo consiste de 3 passos
 1. Encontrar e criar nó de inserção z (novo último nó depois de w)
 2. Armazenar o Item com chave k em z
 3. Restaurar ordem da heap (discutido a seguir)



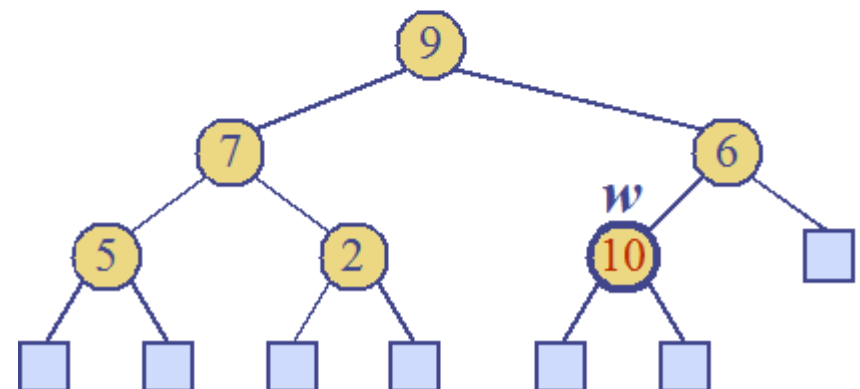
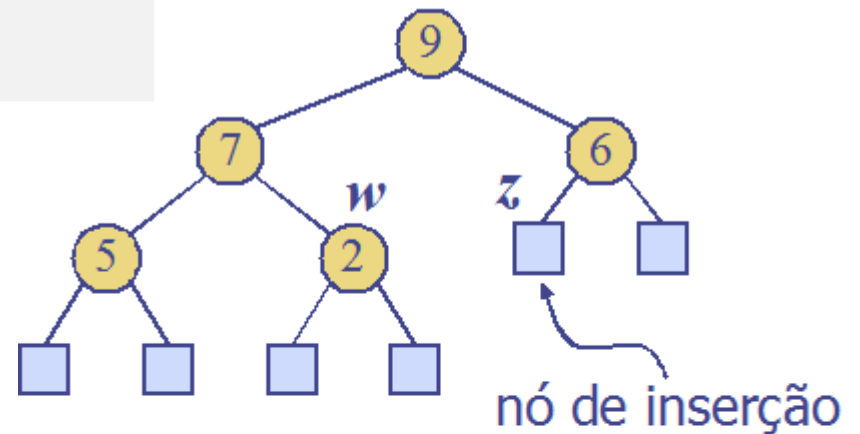
Restauração da Ordem (fix-up)

- Após a inserção de um novo Item, a propriedade de ordem da heap pode ser violada
- A ordem da heap é restaurada trocando os itens caminho acima a partir do nó de inserção
- Termina quando o Item inserido alcança a raiz ou um nó cujo pai possui uma chave maior (ou menor)



Inserção

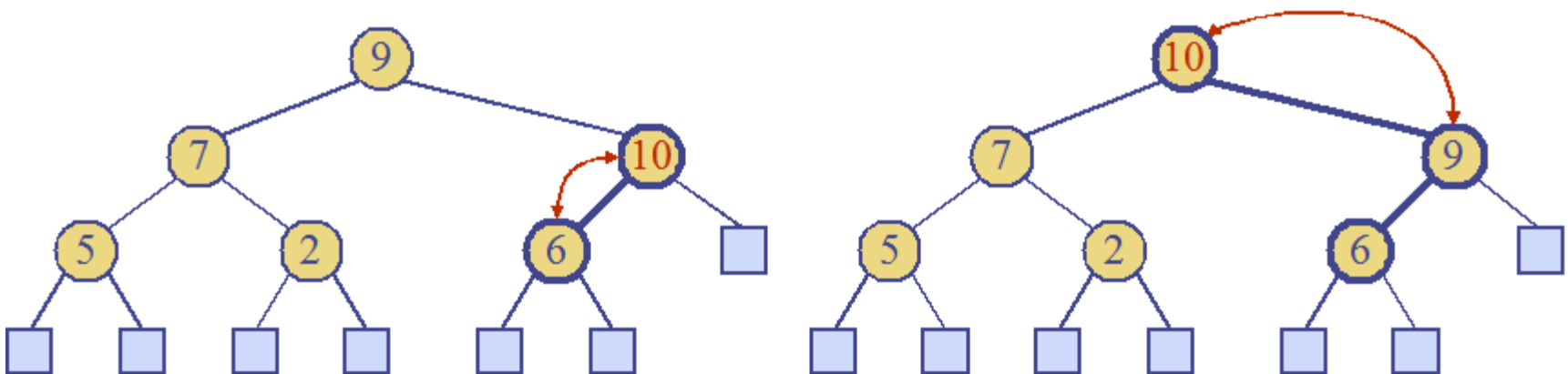
```
Algoritmo Inserir(F,x)
  inserirNoFim(F) //insere na última
  posição
  fix_up(F) //restaura ordem do heap
```



Restauração da Ordem (fix-up)

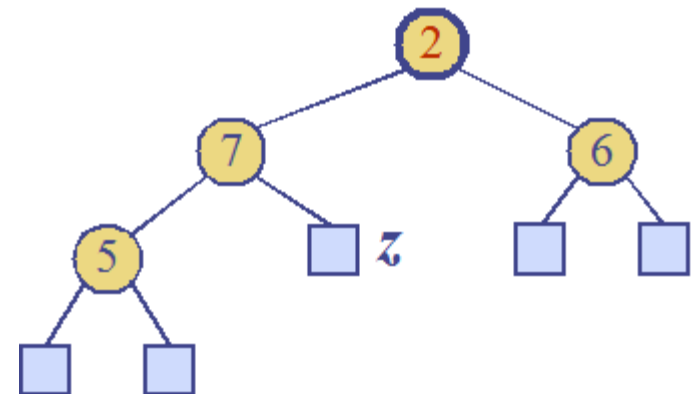
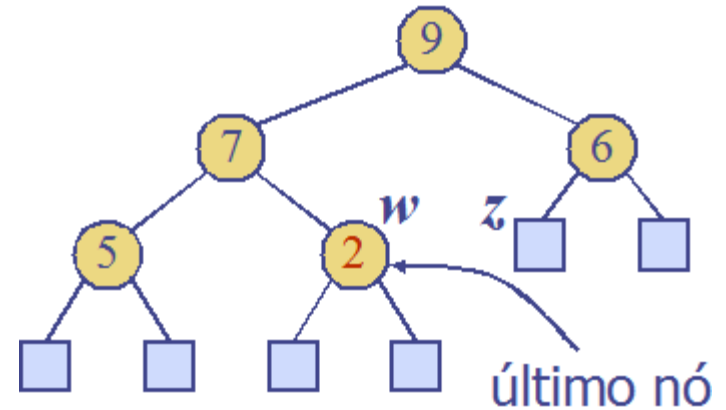
- Para uma heap máxima, temos

```
1 Algoritmo fix_up(F)
2   w = F.ultimo
3   while(!isRoot(F,w)) && (key(F,w) > key(F,parent(F,w)))
4   {
5       swap(F,w,parent(F,w))
6       w = parent(F,w) //sobe
7   }
```



Remoção

- Método remove do TAD fila de prioridade corresponde à remoção do Item da raiz
- O algoritmo de remoção consiste de 3 passos
 1. Armazenar o conteúdo do nó raiz da heap (para retorno)
 2. Copiar o conteúdo de w no nó raiz e remover o nó w
 3. Restaurar ordem da heap (discutido a seguir)

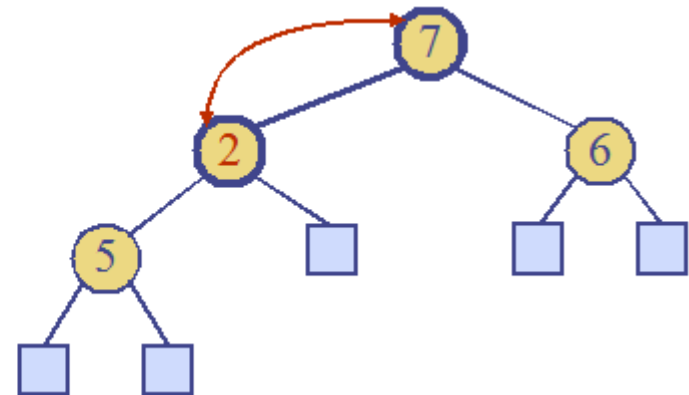
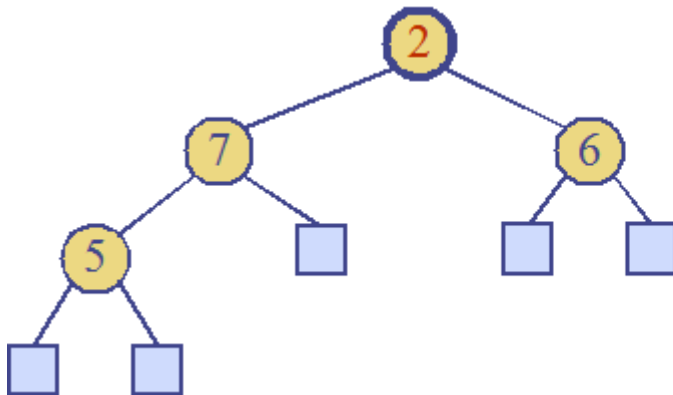


Remoção

- Utilizar o último nó para substituição da raiz na remoção possui várias vantagens, entre elas
 - ▣ Completude garantida (passo 2)
 - ▣ Implementação em tempo constante através de arranjo (discutida posteriormente)

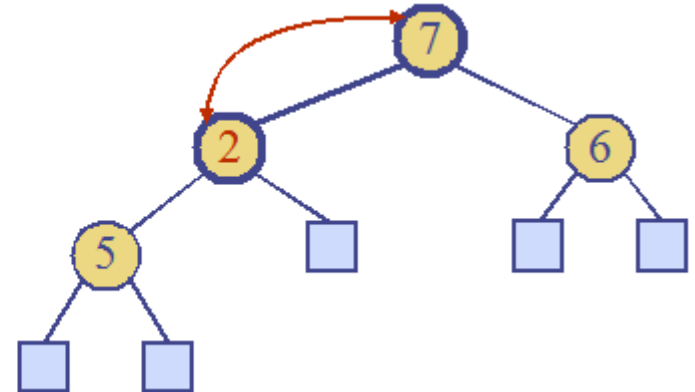
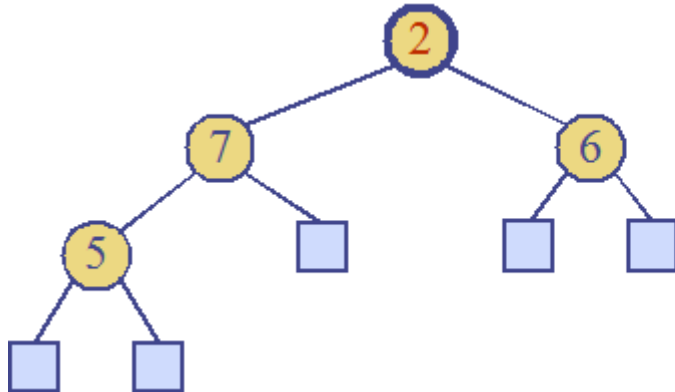
Restauração da Ordem (fix-down)

- Após a remoção, a propriedade de ordem da heap pode ser violada
- A ordem da heap é restaurada trocando os itens caminho abaixo a partir da raiz



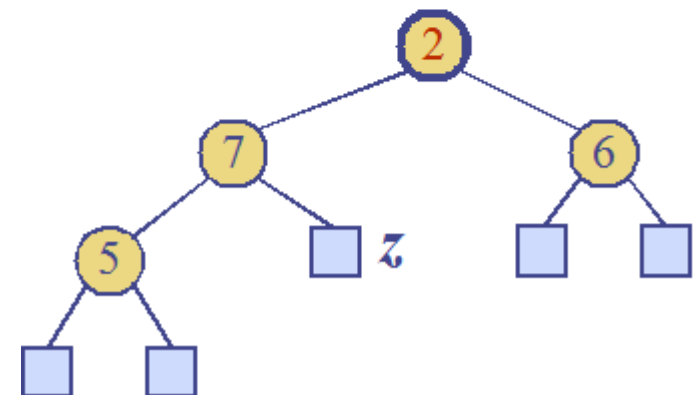
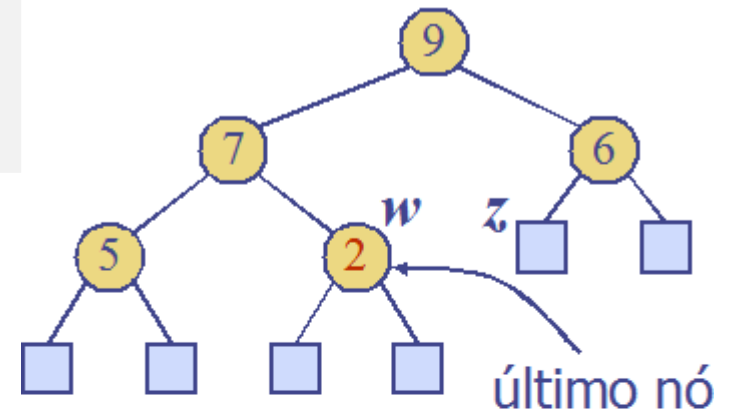
Restauração da Ordem (fix-down)

- O algoritmo fix-down
 - ▣ Termina quando o Item movido para a raiz alcança um nó que não possui filho com chave maior que sua
 - ▣ Quando ambos os filhos possuem chave maior que o Item inserido, a troca é feita com o filho de



Remoção

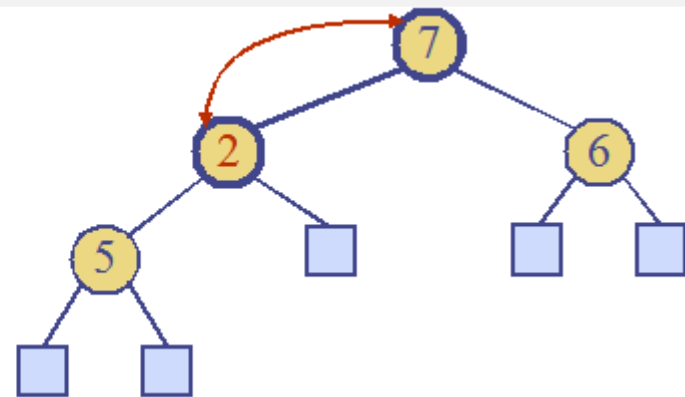
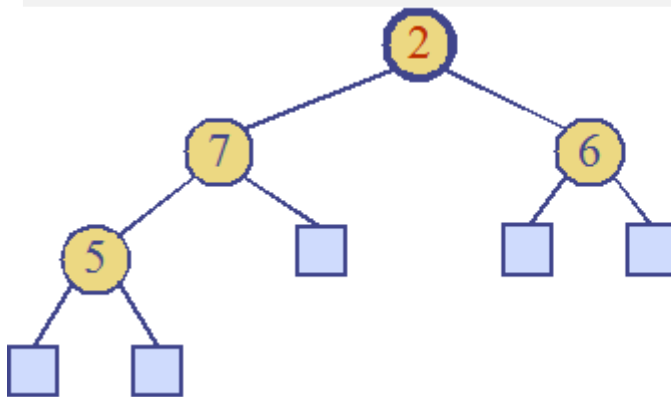
```
Algoritmo Remover(F,x)
  x = inicio(F) //retorna o primeiro nó
  inicio(F) = fim(F) //copia fim no
  início
  fix_down(F) //restaura ordem do heap
```



Restauração da Ordem (fix-down)

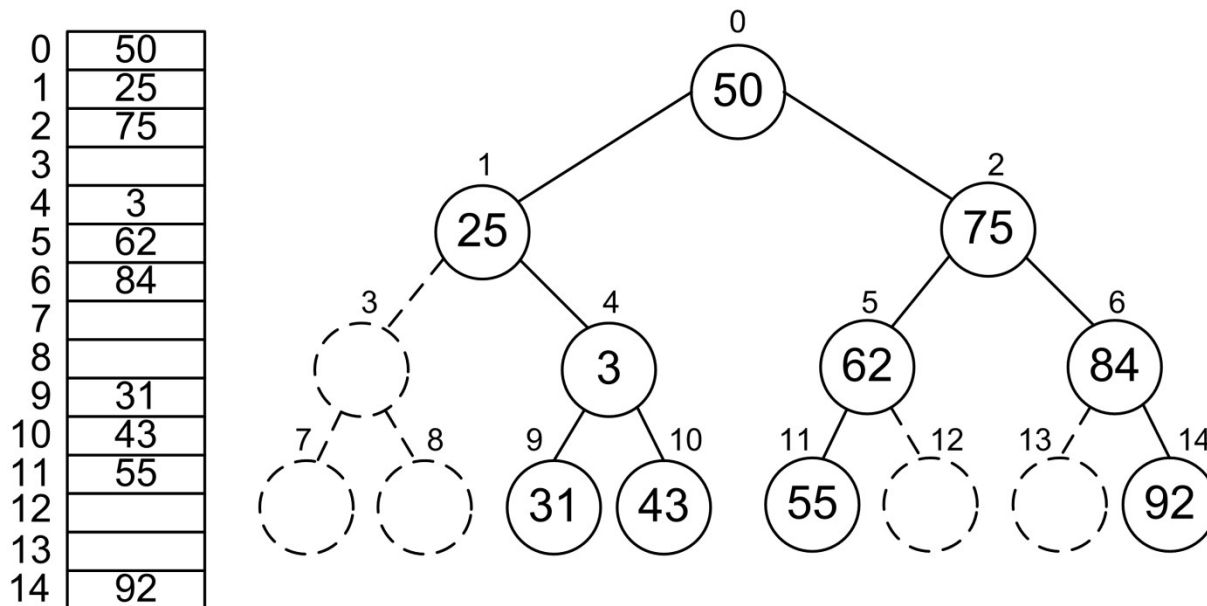
- Para uma heap máxima, temos

```
1 Algoritmo fix_down(F)
2   w = inicio(F)
3   while(tem_filho(w)) {
4     m = maior_filho(w)
5     if(chave(w) >= chave(m)) break
6     swap(F,w,m)
7     w = m //desce
8   }
```



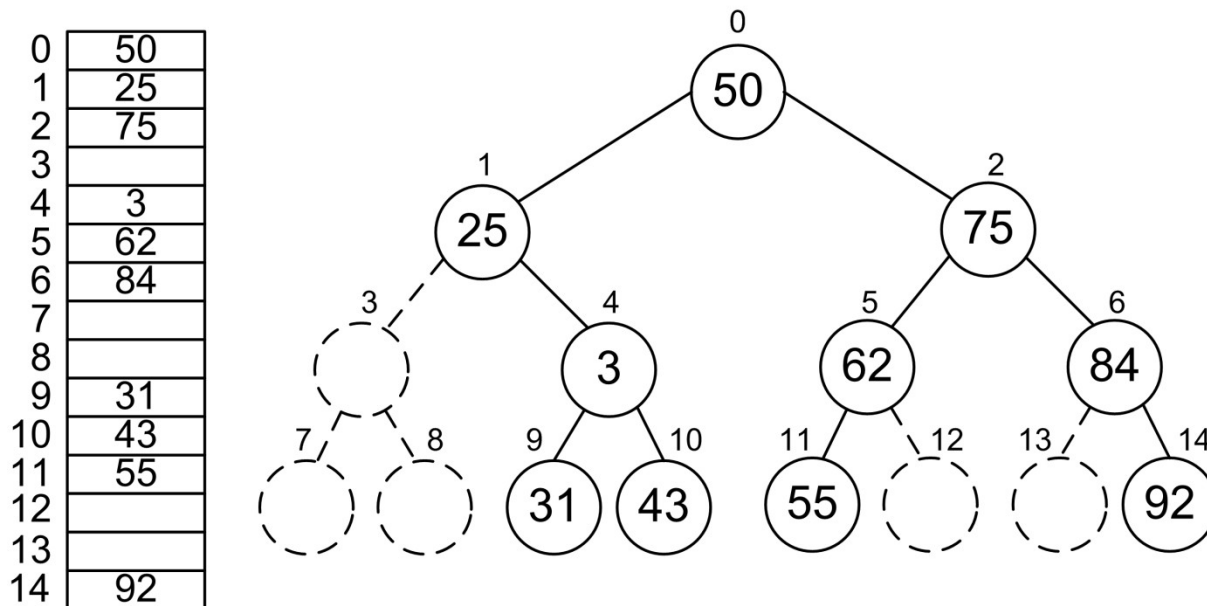
Implementação em Arranjo

- Vetores podem ser empregados para representar árvores binárias
- Caminha pela árvore nível por nível, da esquerda para direita armazenando os nós no vetor
 - ▣ O primeiro nó fica na posição 0 do vetor, seu filho a esquerda fica na posição 1, e assim por diante...



Implementação em Arranjo

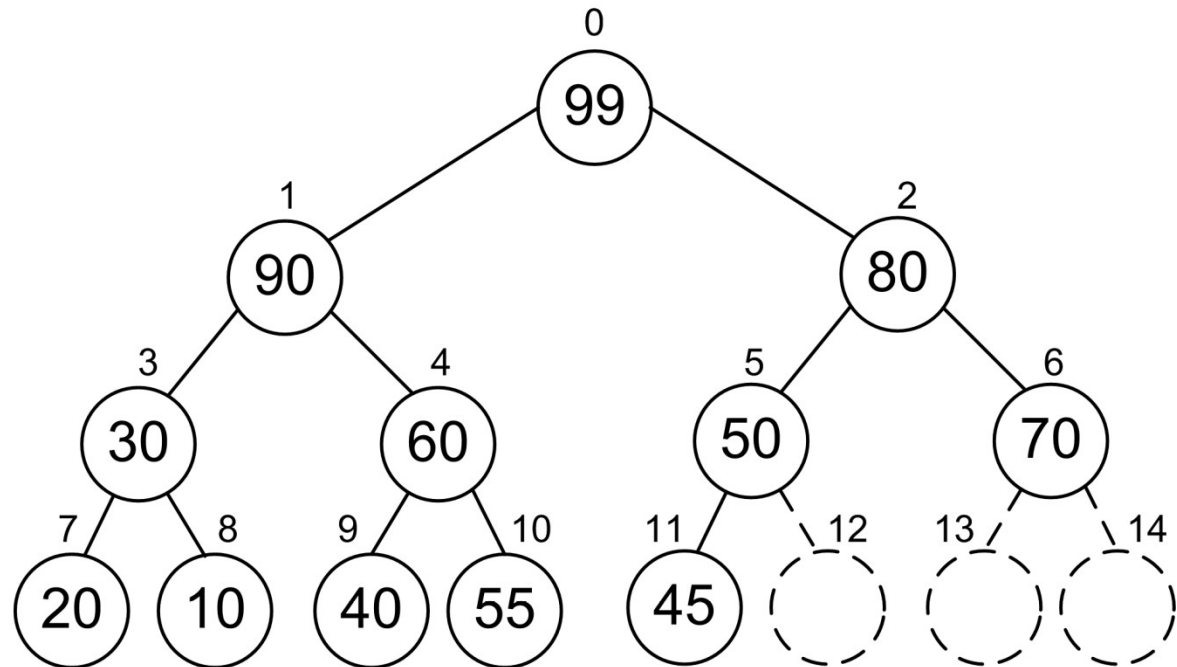
- Nessa definição, dado o índice de um item, podemos encontrar seu
 - ▣ Filho esquerdo : $2 * \text{índice} + 1$
 - ▣ Filho direito : $2 * \text{índice} + 2$
 - ▣ pai: $(\text{índice} - 1)/2$



Implementação em Arranjo

- Como a Heap é uma árvore completa, o vetor não vai ter “buracos” faltando itens
- Os itens que faltam sempre ficam no fim do

V	0	99
	1	90
	2	80
	3	30
	4	60
	5	50
	6	70
	7	20
	8	10
	9	40
	10	55
	11	45
	12	
	13	
	14	



Implementação em Arranjo - Estrutura

```
//.h
typedef struct heap_sequencial HEAP_SEQUENCIAL;
#define TAM 100

//.c
struct heap_sequencial {
    ITEM * vetor[TAM];
    int fim;
};

//main.c
HEAP_SEQUENCIAL *Heap;
```

Implementação em Arranjo - Métodos Básicos

```
1  HEAP_SEQUENCIAL *hep_criar() {
2      HEAP_SEQUENCIAL *heap =
(HEAP_SEQUENCIAL*)malloc(sizeof(HEAP_SEQUENCIAL));
3      if (heap != NULL) {
4          heap->fim = -1;
5      }
6      return heap;
7  }
8
9  int heap_cheia(HEAP_SEQUENCIAL *heap) {
10     return (heap->fim == TAM - 1);
11 }
12
13 int heap_vazia(HEAP_SEQUENCIAL *heap) {
14     return (heap->fim == -1);
15 }
```

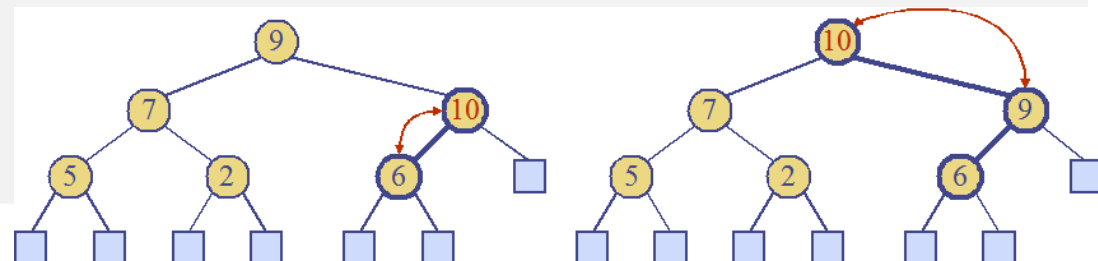
Implementação em Arranjo - Inserção

```
1  int heap_enfileirar(HEAP_SEQUENCIAL *heap, ITEM *item) {
2      if (!heap_cheia(heap)) {
3          heap->fim++;
4          heap->vetor[heap->fim] = item;
5          heap_fix_up(heap);
6          return 1;
7      }
8
9      return 0;
10 }
```

Implementação em Arranjo - Inserção

```
1 void heap_swap(HEAP_SEQUENCIAL *heap, int i, int j) {
2     ITEM *tmp = heap->vetor[i];
3     heap->vetor[i] = heap->vetor[j];
4     heap->vetor[j] = tmp;
5 }
6
7 void heap_fix_up(HEAP_SEQUENCIAL *heap
8     int w = heap->fim;
9     int pai = (w - 1) / 2;
10
11     while (w > 0 && item_get_chave(heap->vetor[w]) >
12         item_get_chave(heap->vetor[pai])) {
13         heap_swap(heap, w, pai);
14         w = pai;
15         pai = (pai - 1) / 2;
16     }
17 }
```

```
1 Algoritmo fix_up(F)
2   w = F.ultimo
3   while(!isRoot(F,w)) && (key(F,w)
4       > key(F,parent(F,w))) {
5       swap(F,w,parent(F,w))
6       w = parent(F,w) //sobe
```



Comparação: Filas de Prioridade


- Via fila ordenada
 - ▣ inserção é $O(n)$
 - ▣ remoção é $O(1)$
 - ▣ próximo é $O(1)$
- Via fila não-ordenada
 - ▣ inserção é $O(1)$
 - ▣ remoção é $O(n)$
 - ▣ próximo é $O(n)$
- Via Heap
 - ▣ inserção é $O(\log n)$
 - ▣ remoção é $O(\log n)$
 - ▣ próximo é $O(1)$

Heap e filas de prioridade

- Heap é utilizado como estrutura de apoio a algoritmos clássicos. Ex:
 - ▣ Ordenação
 - Heapsort
 - Mergesort
 - ▣ Algoritmos em grafos
 - ALGORITMO DE DIJKSTRA → busca de menor caminho em grafo ponderado
 - ALGORITMO DE PRIM → geração de MST (*Minimal Spaning Tree* - Árvore Geradora Mínima)

Exercícios

- Implementar a remoção de um item em uma heap sequencial
- Implementar o TAD fila de prioridades utilizando uma heap encadeada

- 
- Material baseado nos originais produzidos pelos professores:
 - ▣ Gustavo E. de A. P. A. Batista
 - ▣ Fernando V. Paulovich