Filas de Prioridade & Heaps

SCC0202 - Algoritmos e Estruturas de Dados I

Prof. Fernando V. Paulovich
**Baseado no material do Prof. Gustavo Batista*
http://www.icmc.usp.br/~paulovic
paulovic@icmc.usp.br

Instituto de Ciências Matemáticas e de Computação (ICMC)
Universidade de São Paulo (USP)

4 de novembro de 2010

Sumário

Sumário

TAD Fila de Prioridade

- Armazena Itens
- Item: par (chave, informação)
- Operações principais
  - *remove(F)*: remove e retorna o item com maior (menor) prioridade da fila **F**
  - *insere(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
  - *conta(F)*, *vazia(F)*, *cheia(F)*

TAD Fila de Prioridade

- Diferentes Realizações (implementações)

TAD Fila de Prioridade

- Diferentes Realizações (implementações)
  - Estáticas
    - Lista estática (arranjo) ordenada
    - Lista estática (arranjo) não ordenada
    - Heap em arranjo

TAD Fila de Prioridade

- Diferentes Realizações (implementações)
  - Estáticas
    - Lista estática (arranjo) ordenada
    - Lista estática (arranjo) não ordenada
    - Heap em arranjo
  - Dinâmicas
    - Lista dinâmica ordenada
    - Lista dinâmica não ordenada
    - Heap dinâmico

TAD Fila de Prioridade

- Diferentes Realizações (implementações)
  - Estáticas
    - Lista estática (arranjo) ordenada
    - Lista estática (arranjo) não ordenada
    - Heap em arranjo
  - Dinâmicas
    - Lista dinâmica ordenada
    - Lista dinâmica não ordenada
    - Heap dinâmico

- Cada realização possui vantagens e desvantagens

TAD Fila de Prioridade

- Uma das escolhas diretas seria usar uma fila ordenada
  - inserção é $O(n)$
  - remoção é $O(1)$
  - próximo é $O(1)$

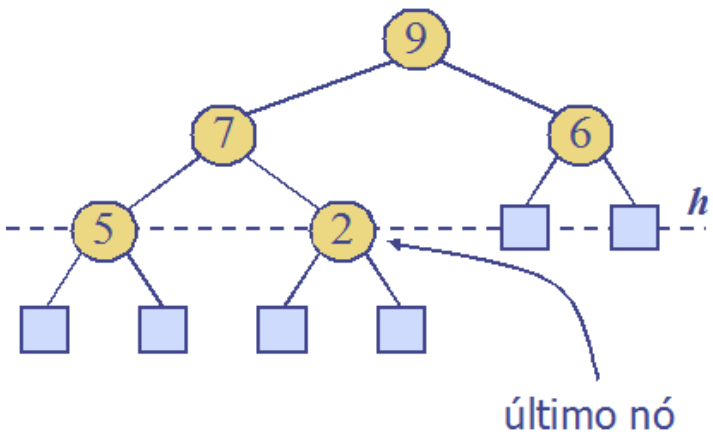
- Outra seria usar uma fila não-ordenada
  - inserção é $O(1)$
  - remoção é $O(n)$
  - próximo é $O(n)$

- Portanto uma abordagem mais rápida precisa ser pensada quando grandes conjuntos de dados são considerados

Sumário

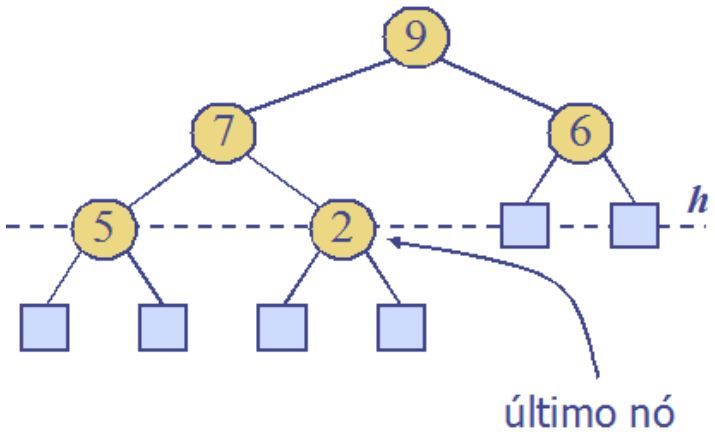
Heaps

- Um heap é uma árvore binária que satisfaz as propriedades
  - **Ordem**: para cada nó v, exceto o nó raiz, tem-se que
    - $chave(v) \leq chave(pai(v))$ - heap máximo
    - $chave(v) \geq chave(pai(v))$ - heap mínimo



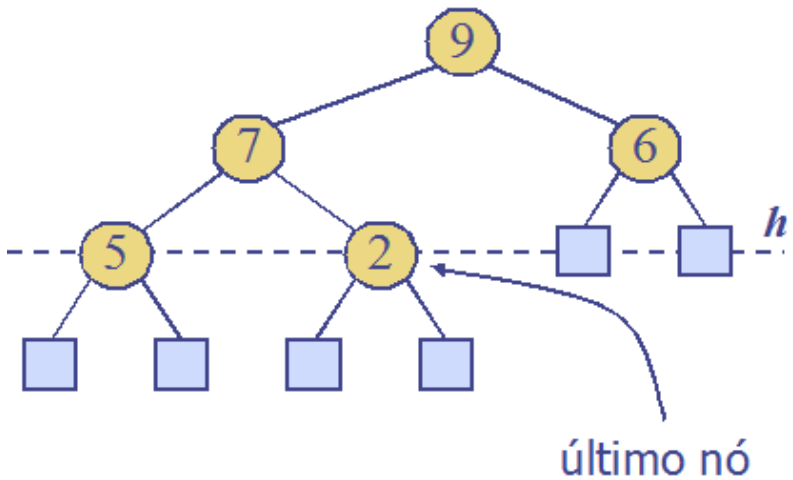
Heaps

- Um heap é uma árvore binária que satisfaz as propriedades
  - **Completude**: 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

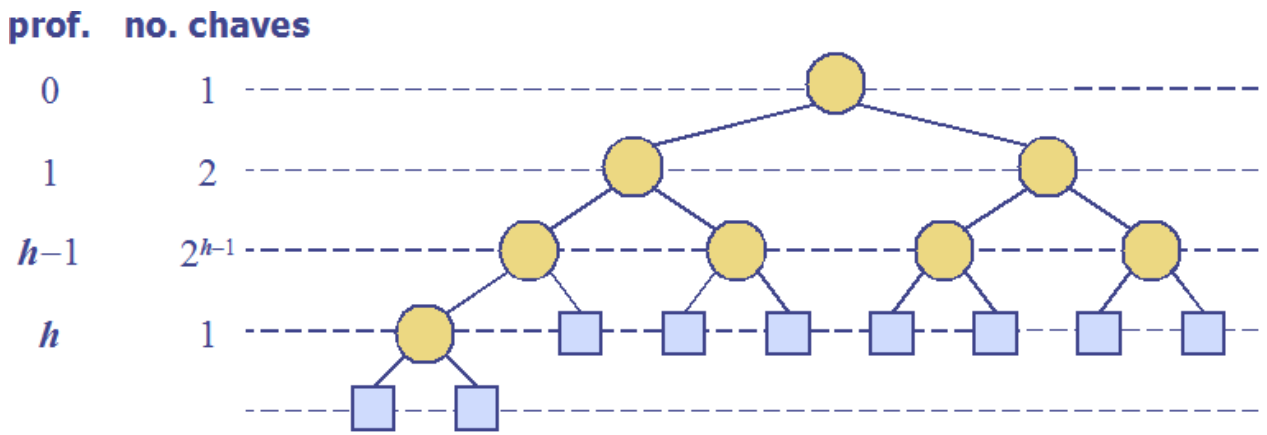
- Convenciona-se aqui
  - Último nó: nó interno mais à direita de profundidade h



Altura de um Heap

Teorema

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



Altura de um Heap

Prova

- Dado que existem 2^i chaves na profundidade $i = 0, \ldots, h-1$ e ao menos 1 chave na profundidade h, tem-se $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$

Altura de um Heap

Prova

- Dado que existem 2^i chaves na profundidade $i = 0, \ldots, h-1$ e ao menos 1 chave na profundidade h, tem-se $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$

- Isso é uma Progressão Geométrica (PG) com razão $q = 2$, dado que a soma de um PG pode ser calculada por $S_k = \frac{a^k \times q - a_1}{q-1}$, temos $n \geq (2^{h-1} \times 2 - 1) + 1 = 2^h$

Altura de um Heap

Prova

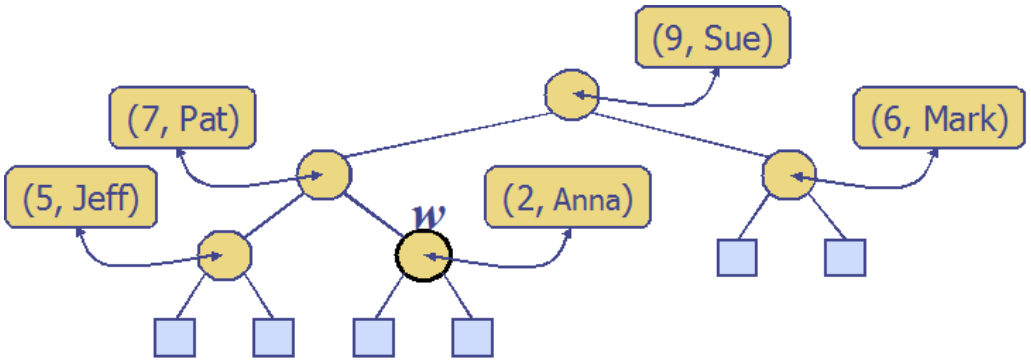
- Dado que existem 2^i chaves na profundidade $i = 0, \ldots, h-1$ e ao menos 1 chave na profundidade h, tem-se $n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$

- Isso é uma Progressão Geométrica (PG) com razão $q = 2$, dado que a soma de um PG pode ser calculada por $S_k = \frac{a^k \times q - a_1}{q-1}$, temos $n \geq (2^{h-1} \times 2 - 1) + 1 = 2^h$

- Logo, $n \geq 2^h$, i.e., $h \leq log_2\ n \Rightarrow h$ é $O(log\ n)$

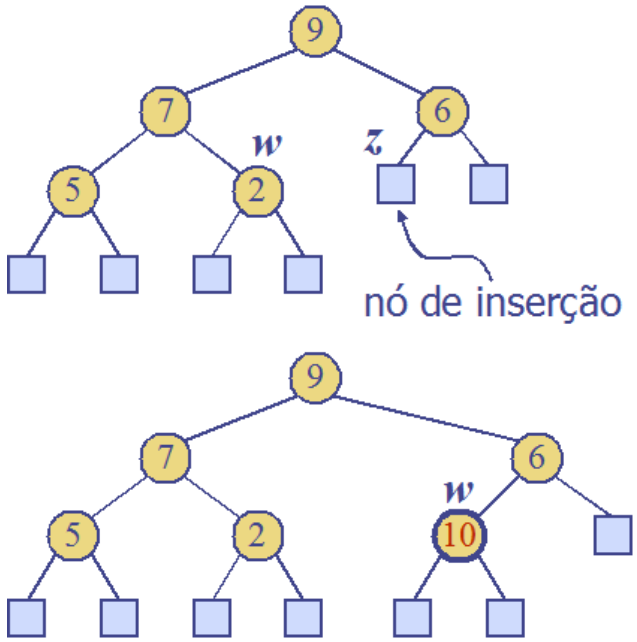
Filas de Prioridade com Heaps

- 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 do heap
  - Heap mínimo: menor chave na raiz do heap
  - Heap máximo: maior chave na raiz do heap



Inserção

- Método insere do TAD fila de prioridade corresponde à inserção de um *Item* no 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 do heap (discutido a seguir)

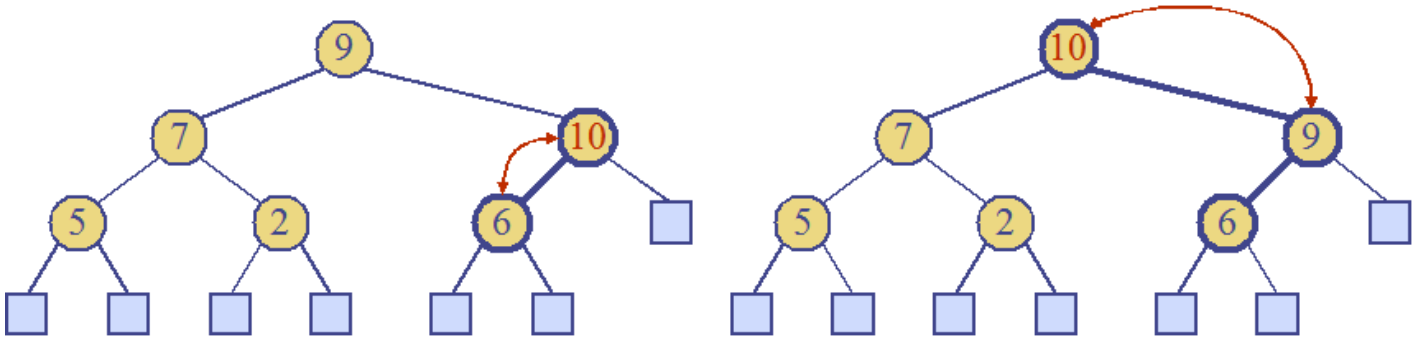


Restauração da Ordem (bubbling-up)

- Após a inserção de um novo *Item*, a propriedade de ordem do heap pode ser violada

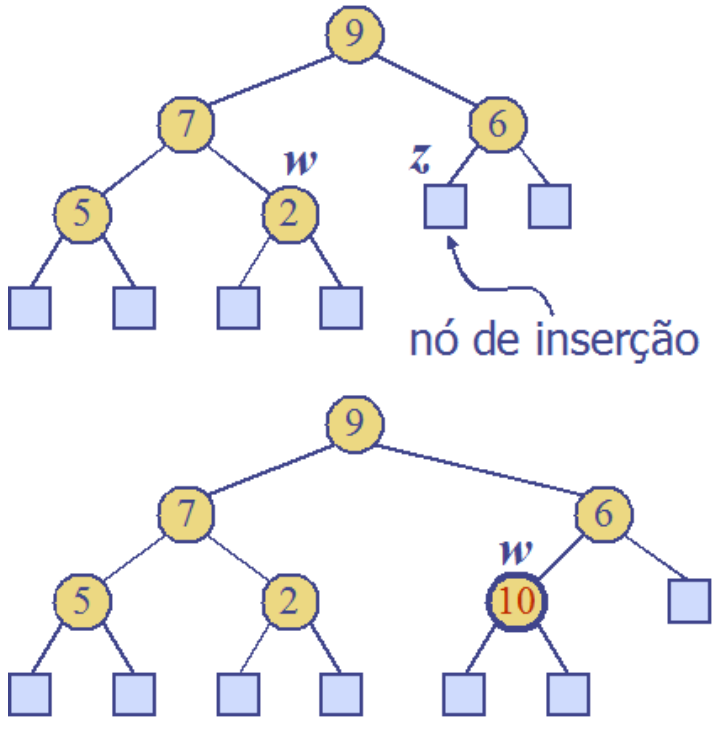
Restauração da Ordem (bubbling-up)

- Após a inserção de um novo *Item*, a propriedade de ordem do heap pode ser violada
- A ordem do 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

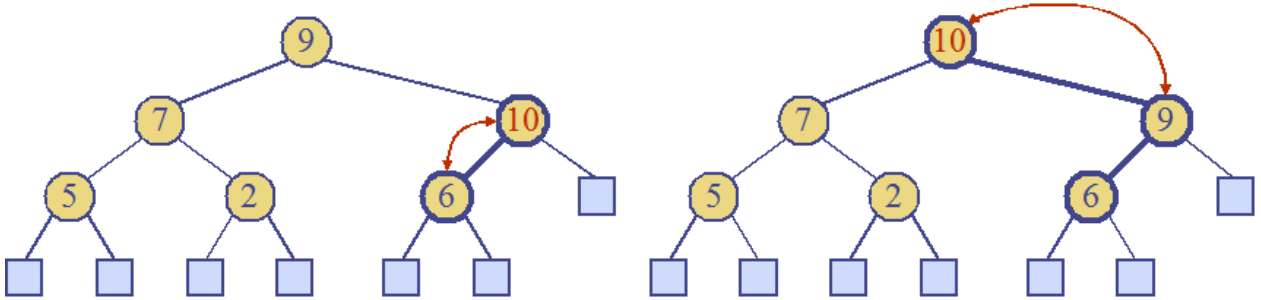
```
Algoritmo Inserir(F,x)
  inserirNoFim(F) //insere na última posição
  bubbling_up(F) //restaura ordem do heap
```



Restauração da Ordem (bubbling-up)

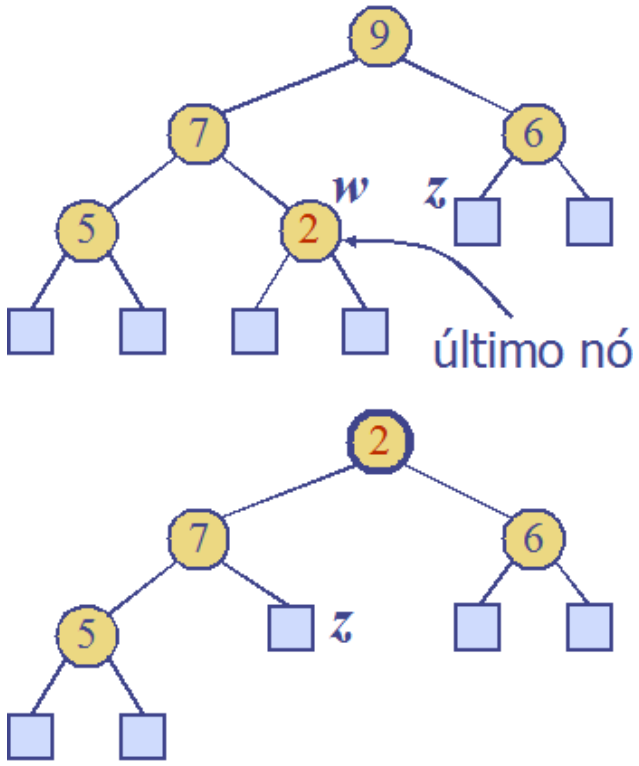
- Para um heap máximo, temos

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



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 do heap (para retorno)
  2. Copiar o conteúdo do w no nó raiz e remover o nó w
  3. Restaurar ordem do heap (discutido a seguir)



Remoção

- Utilizar o último nó para substituição da raiz na remoção possui várias vantagens, entre elas

Remoção

- Utilizar o último nó para substituição da raiz na remoção possui várias vantagens, entre elas
  - Completude garantida (passo 2)

Remoção

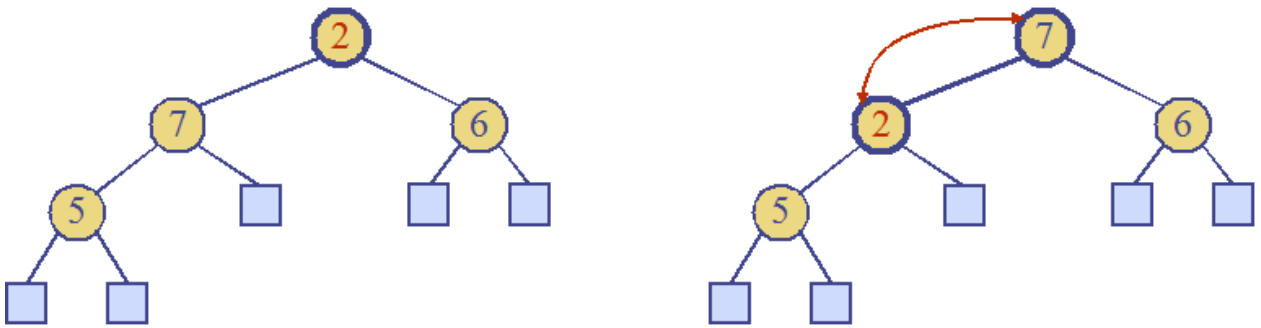
- Utilizar o último nó para substituição da raiz na remoção possui várias vantagens, entre elas
  - Completude garantida (passo 2)
  - Implementação em tempo constante através de arranjo (discutida posteriormente)

Restauração da Ordem (bubbling-down)

- Após a remoção, a propriedade de ordem do heap pode ser violada

Restauração da Ordem (bubbling-down)

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

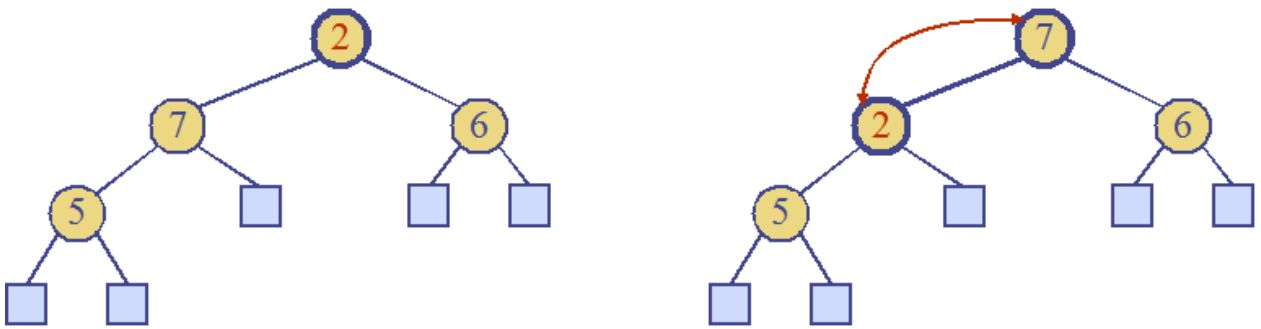


Restauração da Ordem (bubbling-down)

- O algoritmo bubbling-down
  - Termina quando o *Item* movido para a raiz alcança um nó que não possui filho com chave maior que sua

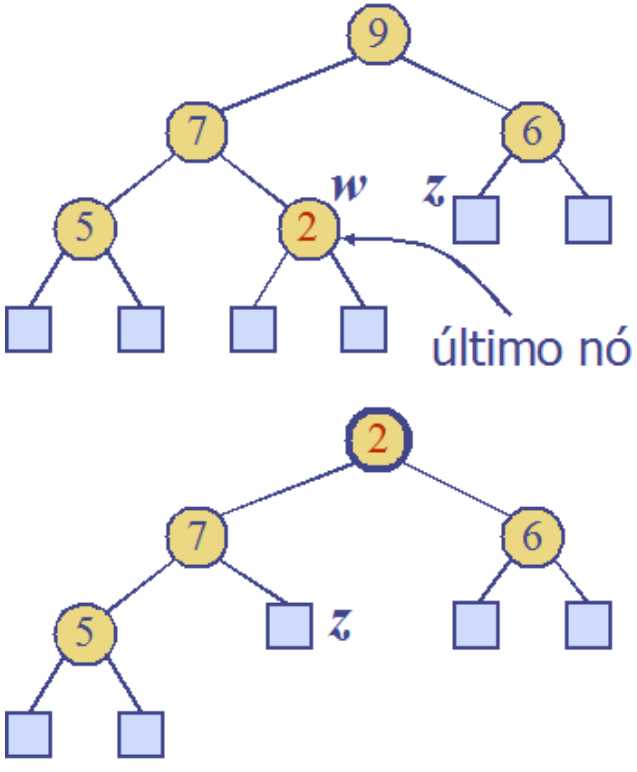
Restauração da Ordem (bubbling-down)

- O algoritmo bubbling-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 maior chave



Remoção

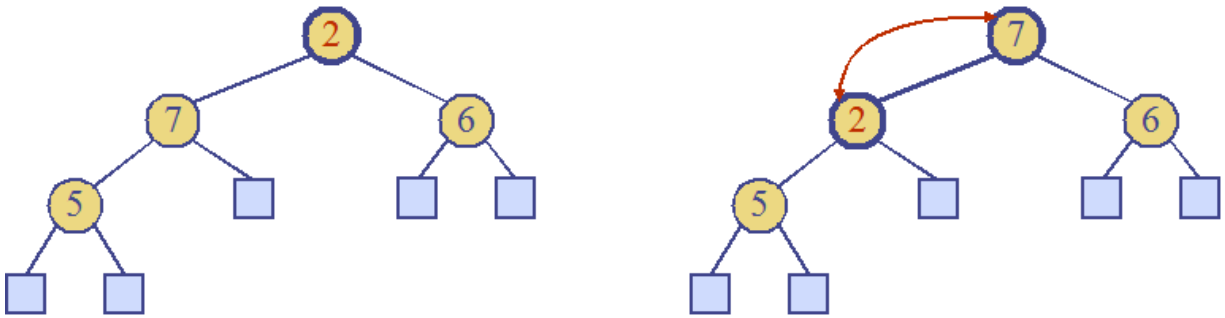
```
Algoritmo Remover(F,x)
  x = inicio(F) //retorna o primeiro nó
  inicio(F) = fim(F) //copia fim no início
  bubbling_down(F) //restaura ordem do heap
```



Restauração da Ordem (bubbling-up)

- Para um heap máximo, temos

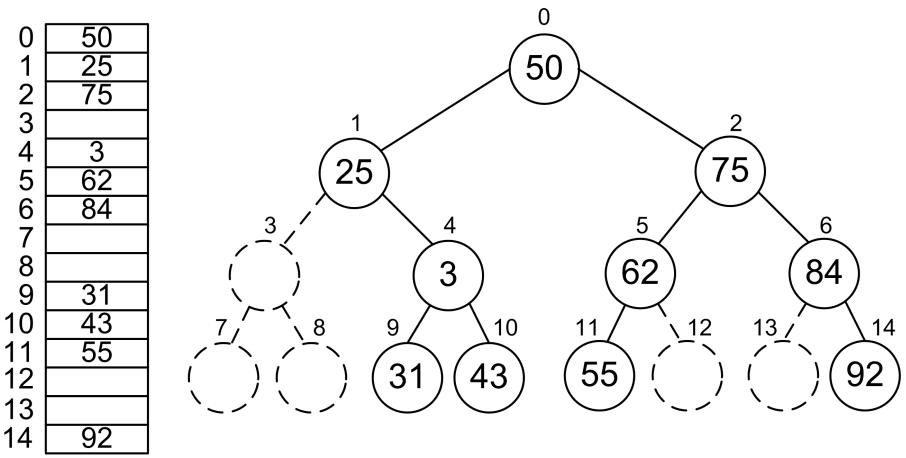
```
1  Algoritmo bubbling_down(F)
2    w = inicio(F)
3    while(tem_filho(w)) {
4      m = maior_filho(w)
5      if(chave(w) >= chave(m)) break
6      swap(F,w,m)
7      w = m //desce
8    }
```



Sumário

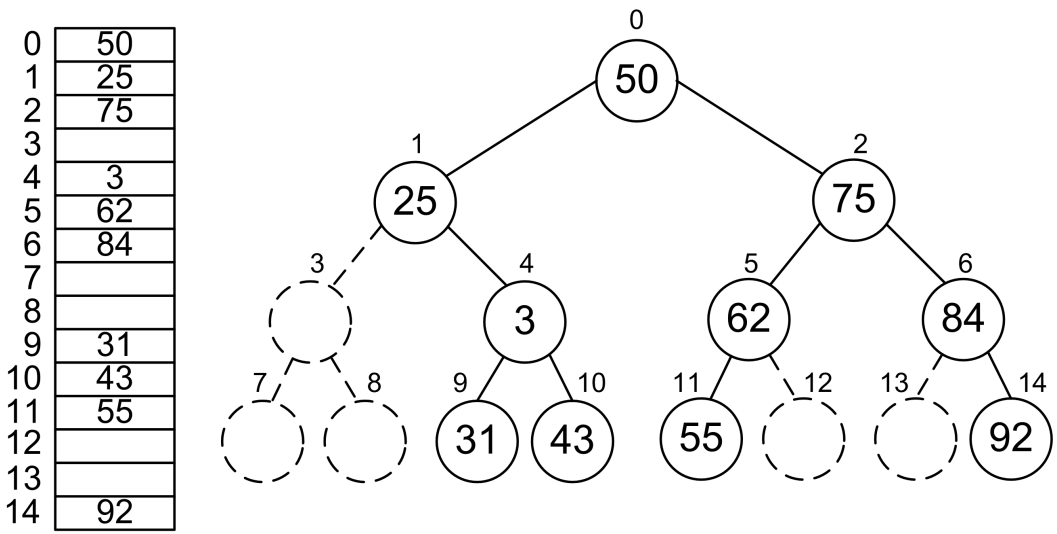
Implementação em Arranjo

- 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...



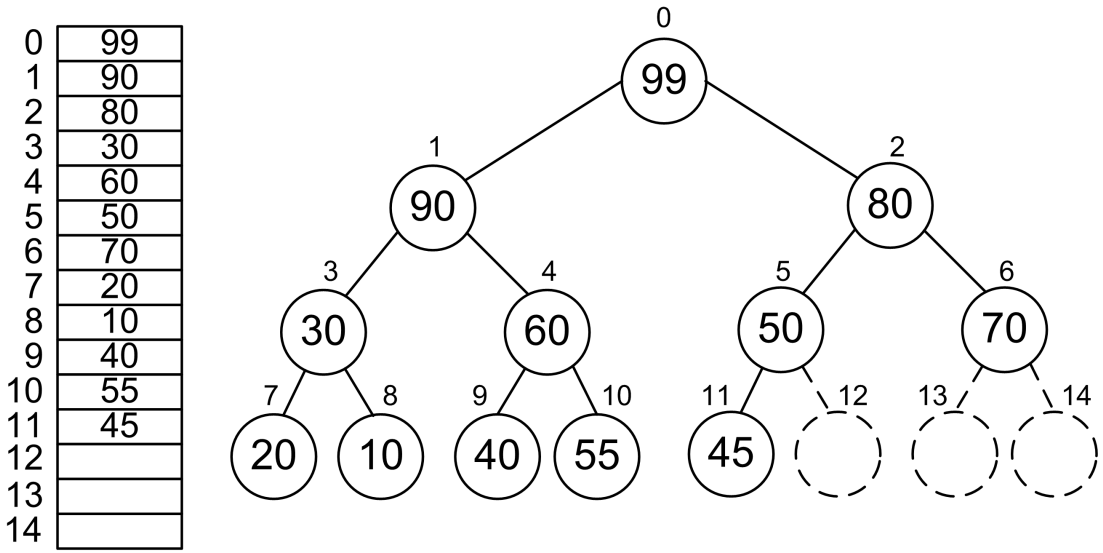
Implementação em Arranjo

- Nessa definição, dado o $indice$ de um item, podemos encontrar seu
  - filho a esquerda : $2 * indice + 1$
  - filho a direita : $2 * indice + 2$
  - pai: $(indice - 1)/2$



Implementação em Arranjo

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



Implementação em Arranjo - Estrutura

```
1  #define TAM 100
2
3  typedef struct {
4    int valor;
5    int chave;
6  } ITEM;
7
8  typedef struct {
9    ITEM itens[TAM];
10   int fim;
11 } FILA_PRIORIDADE;
```

Implementação em Arranjo - Métodos Básicos

```
1  void criar(FILA_PRIORIDADE *fila) {
2    fila->fim = -1;
3  }
4
5  int vazia(FILA_PRIORIDADE *fila) {
6    return (fila->fim == -1);
7  }
8
9  int cheia(FILA_PRIORIDADE *fila) {
10   return (fila->fim == TAM-1);
11 }
```

Implementação em Arranjo - Inserção

```
1  int inserir(FILA_PRIORIDADE *fila, ITEM *item) {
2    if (!cheia(fila)) {
3      fila->fim++; //move o fim da fila
4      fila->itens[fila->fim] = *item; //adiciona novo item
5      bubbling_up(fila); //restaura ordem do heap
6      return 1;
7    }
8
9    return 0;
10 }
```

Implementação em Arranjo - Inserção

```
1  void bubbling_up(FILA_PRIORIDADE *fila) {
2    int indice = fila->fim;
3    int pai = (indice-1)/2;
4
5    while (indice > 0 && //não é a raiz
6           fila->itens[indice].chave > fila->itens[pai].chave) {
7      //troco os nós de posição
8      ITEM tmp = fila->itens[indice];
9      fila->itens[indice] = fila->itens[pai];
10     fila->itens[pai] = tmp;
11
12     indice = pai; //move índice para cima
13     pai = (pai-1)/2; //pai recebe o próprio pai
14   }
15 }
```

Implementação em Arranjo - Remoção

```
1  int remover(FILA_PRIORIDADE *fila, ITEM *item) {
2    if (!vazia(fila)) {
3      *item = fila->itens[0]; //retorna o primeiro item
4      fila->itens[0] = fila->itens[fila->fim]; //copia o último para a ↩
           primeira posição
5      fila->fim--; //decrementa o tamanho da lista
6      bubbling_down(fila); //restaura ordem do heap
7      return 1;
8    }
9
10   return 0;
11 }
```

Implementação em Arranjo - Remoção

```
1  void bubbling_down(FILA_PRIORIDADE *fila) {
2    int indice = 0;
3
4    while (indice < fila->fim / 2) { //enquanto nó tiver ao menos um filho
5      int filhoesq = 2 * indice + 1;
6      int filhodir = 2 * indice + 2;
7
8      int maiorfilho; //encontra maior filho
9      if (filhodir <= fila->fim && //tem filho a direita
10         fila->itens[filhoesq].chave < fila->itens[filhodir].chave) {
11       maiorfilho = filhodir;
12     }else {
13       maiorfilho = filhoesq;
14     }
15
16     //pare caso o item seja igual ou maior ao maior filho
17     if (fila->itens[indice].chave >= fila->itens[maiorfilho].chave) {
18       break;
19     }
20
21     //troco o maior filho com o pai
22     ITEM tmp = fila->itens[indice];
23     fila->itens[indice] = fila->itens[maiorfilho];
24     fila->itens[maiorfilho] = tmp;
25     indice = maiorfilho; //desce
26   }
27 }
```

Comparação de Filas de Prioridade

Via Lista Não-Ordenada

Operação	Tempo
próximo	$O(n)$
insere	$O(1)$
remove	$O(n)$

Via Lista Ordenada

Operação	Tempo
próximo	$O(1)$
insere	$O(n)$
remove	$O(1)$

Via Heaps

Operação	Tempo
proximo	$O(1)$
insere	$O(log\ n)$
remove	$O(log\ n)$

Exercício

- Implemente uma fila de prioridade que use um *Heap* dinâmico