

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)

6 de novembro de 2017



Sumário

- 1 TAD Fila de Prioridade
- 2 Heaps
- 3 Implementação em Array

Sumário

- 1 TAD Fila de Prioridade
- 2 Heaps
- 3 Implementação em Array

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 implementações

TAD Fila de Prioridade

- Diferentes implementações
 - Estáticas
 - Lista estática (array) ordenada
 - Lista estática (array) não ordenada
 - Heap em Array

TAD Fila de Prioridade

- Diferentes implementações
 - Estáticas
 - Lista estática (array) ordenada
 - Lista estática (array) não ordenada
 - Heap em Array
 - Dinâmicas
 - Lista dinâmica ordenada
 - Lista dinâmica não ordenada
 - Heap dinâmico

TAD Fila de Prioridade

- Diferentes implementações
 - Estáticas
 - Lista estática (array) ordenada
 - Lista estática (array) não ordenada
 - Heap em Array
 - Dinâmicas
 - Lista dinâmica ordenada
 - Lista dinâmica não ordenada
 - Heap dinâmico
- Cada implementação possui vantagens e desvantagens

TAD Fila de Prioridade

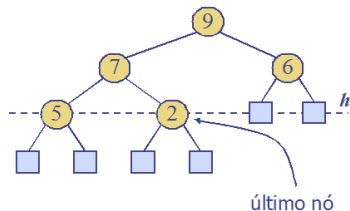
- Uma das escolhas diretas seria usar uma fila ordenada
 - inserção é $O(n)$
 - remoção é $O(1)$
 - próximo é $O(1)$
- Outra seria usar uma fila não-ordenada
 - inserção é $O(1)$
 - remoção é $O(n)$
 - próximo é $O(n)$
- Portanto uma abordagem mais rápida precisa ser pensada quando grandes conjuntos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

Sumário

- 1 TAD Fila de Prioridade
- 2 Heaps
- 3 Implementação em Array

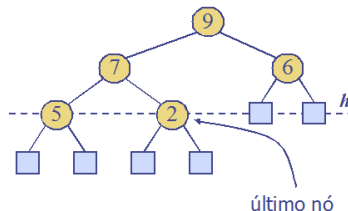
Heaps

- 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



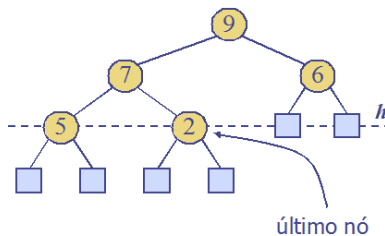
Heaps

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

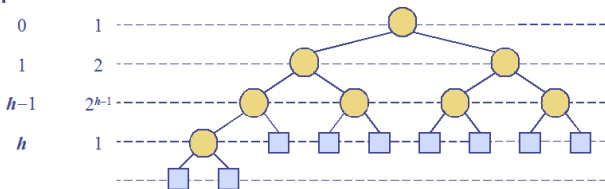


Altura de um Heap

Teorema

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

prof. no. chaves



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

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

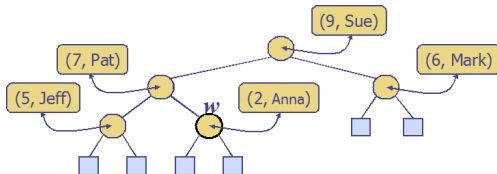
Altura de um Heap

Prova

- Dado que existem 2^i chaves na profundidade $i = 0, \dots, h - 1$ 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 $S_k = \frac{a^k \times q - a_1}{q - 1}$, 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 $O(\log n)$

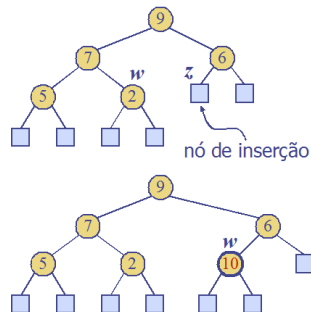
Filas de Prioridade com Heaps

- 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)

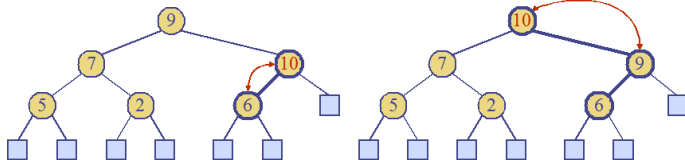


Restauração da Ordem (fix-up)

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

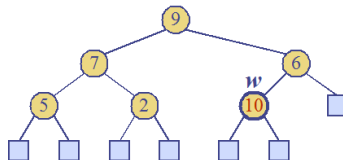
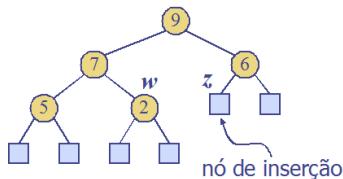
Restauração da Ordem (fix-up)

- 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

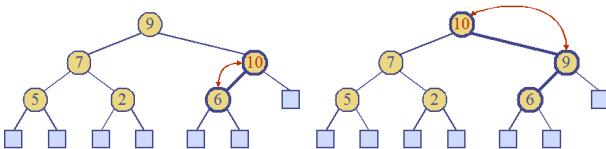
```
Algoritmo Inserir(F,x)
  inserirNoFim(F) //insere na última posição
  fix_up(F) //restaura ordem do heap
```



Restauração da Ordem (fix-up)

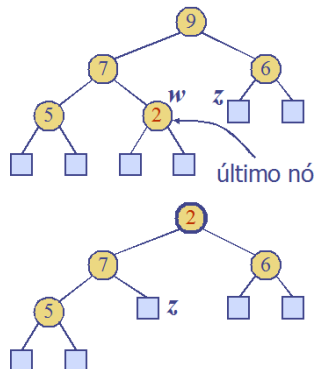
- Para um heap máximo, temos

```
1 Algoritmo fix_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

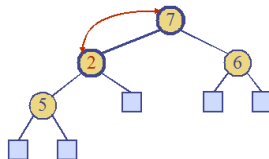
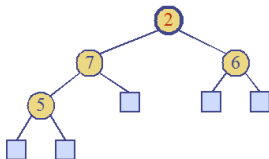
- Utilizar o último nó para substituição da raiz na remoção possui várias vantagens, entre elas
 - Completude garantida (passo 2)
 - Implementação em tempo constante através de array (discutida posteriormente)

Restauração da Ordem (fix-down)

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

Restauração da Ordem (fix-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

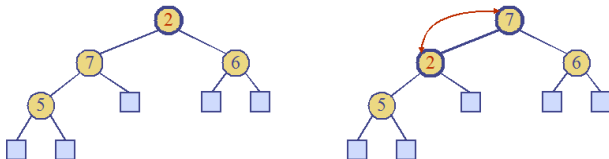


Restauração da Ordem (fix-down)

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

Restauração da Ordem (fix-down)

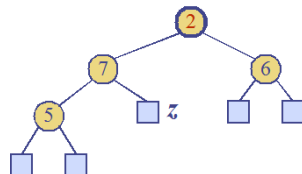
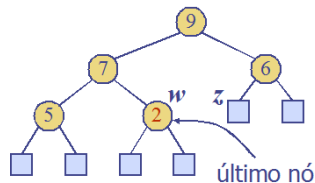
- O algoritmo fix-down
 - Termina quando o *Item* movido para a raiz alcança um nó que não possui filho com chave maior que sua
 - Quando ambos os filhos possuem chave maior que o *Item* inserido, a troca é feita com o filho de maior chave



Remoção

Algoritmo Remove(F,x)

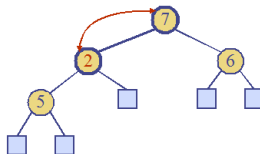
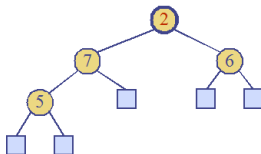
```
x = inicio(F) //retorna o primeiro nó  
inicio(F) = fim(F) //copia fim no início  
fix_down(F) //restaura ordem do heap
```



Restauração da Ordem (fix-up)

- Para um heap máximo, 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   }
```

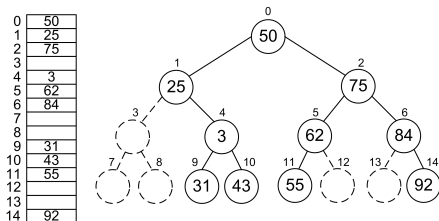


Sumário

- 1 TAD Fila de Prioridade
- 2 Heaps
- 3 Implementação em Array

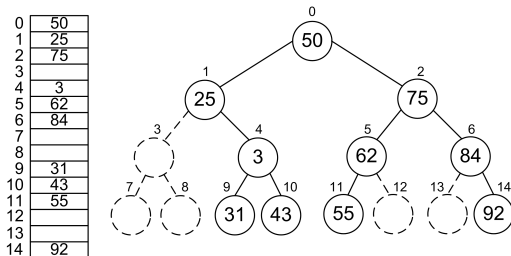
Implementação em Array

- 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...



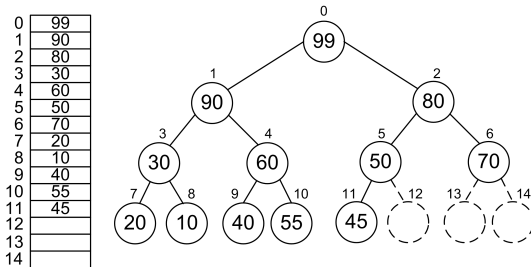
Implementação em Array

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



Implementação em Array

- 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 Array - Estrutura

```
1 typedef struct heap_estatica HEAP_ESTATICA;  
2  
3 #define TAM 100  
4  
5 struct heap_estatica {  
6     ITEM * vetor[TAM];  
7     int fim;  
8 };
```

Implementação em Array - Métodos Básicos

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

Implementação em Array - Inserção

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

Implementação em Array - Inserção

```
1 void swap(HEAP_ESTATICA *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 fix_up(HEAP_ESTATICA *heap) {
8     int pos = heap->fim;
9     int pai = (pos - 1) / 2;
10
11     while (pos > 0 &&
12           heap->vetor[pos]->chave > heap->vetor[pai]->chave) {
13         swap(heap, pos, pai);
14         pos = pai;
15         pai = (pai - 1) / 2;
16     }
17 }
```

Implementação em Array - Remoção

```
1  ITEM *desenfileirar(HEAP_ESTATICA *heap) {  
2      if (!vazia(heap)) {  
3          ITEM *item = heap->vetor[0];  
4          heap->vetor[0] = heap->vetor[heap->fim];  
5          heap->fim--;  
6          fix_down(heap);  
7          return item;  
8      }  
9      return NULL;  
10 }
```

Implementação em Array - Remoção

```
1 void fix_down(HEAP_ESTATICA *heap) {
2     int pos = 0;
3
4     while (pos <= heap->fim / 2) {
5         int filhoesq = 2 * pos + 1;
6         int filhodir = 2 * pos + 2;
7
8         int maiorfilho;
9         if (filhodir <= heap->fim &&
10             heap->vetor[filhoesq]->chave < heap->vetor[filhodir]->chave↵
11             ) {
12             maiorfilho = filhodir;
13         } else {
14             maiorfilho = filhoesq;
15         }
16
17         if (heap->vetor[pos]->chave >= heap->vetor[maiorfilho]->chave) {
18             break;
19         }
20
21         swap(heap, pos, maiorfilho);
22         pos = maiorfilho;
23     }
```

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