

Universidade Estadual de Campinas - UNICAMP
Instituto de Computação - IC

Filas de Prioridade

- 1 Filas de Prioridade
- 2 *Heaps*
- 3 *Heapsort*

- TAD em que cada elemento do conjunto de dados está associado a uma *prioridade*.

– As operações básicas relativas a uma *fila de prioridade* são:

- ① Construção de uma fila de prioridade a partir de um conjunto com n itens.
- ② Seleção do elemento de maior (menor) prioridade.
- ③ Inserção de um novo elemento.
- ④ Remoção do elemento de maior (menor) prioridade.
- ⑤ Alteração do elemento de maior (menor) prioridade.

- TAD em que cada elemento do conjunto de dados está associado a uma *prioridade*.
- As operações básicas relativas a uma *fila de prioridade* são:
 - ① Construção de uma fila de prioridade a partir de um conjunto com n itens.
 - ② Seleção do elemento de maior (menor) prioridade.
 - ③ Inserção de um novo elemento.
 - ④ Remoção do elemento de maior (menor) prioridade.
 - ⑤ Alteração do elemento de maior (menor) prioridade.

- TAD em que cada elemento do conjunto de dados está associado a uma *prioridade*.
- As operações básicas relativas a uma *fila de prioridade* são:
 - ① Construção de uma fila de prioridade a partir de um conjunto com n itens.
 - ② Seleção do elemento de maior (menor) prioridade.
 - ③ Inserção de um novo elemento.
 - ④ Remoção do elemento de maior (menor) prioridade.
 - ⑤ Alteração do elemento de maior (menor) prioridade.

- TAD em que cada elemento do conjunto de dados está associado a uma *prioridade*.
- As operações básicas relativas a uma *fila de prioridade* são:
 - ① Construção de uma fila de prioridade a partir de um conjunto com n itens.
 - ② Seleção do elemento de maior (menor) prioridade.
 - ③ Inserção de um novo elemento.
 - ④ Remoção do elemento de maior (menor) prioridade.
 - ⑤ Alteração do elemento de maior (menor) prioridade.

- TAD em que cada elemento do conjunto de dados está associado a uma *prioridade*.
- As operações básicas relativas a uma *fila de prioridade* são:
 - ① Construção de uma fila de prioridade a partir de um conjunto com n itens.
 - ② Seleção do elemento de maior (menor) prioridade.
 - ③ Inserção de um novo elemento.
 - ④ Remoção do elemento de maior (menor) prioridade.
 - ⑤ Alteração do elemento de maior (menor) prioridade.

- TAD em que cada elemento do conjunto de dados está associado a uma *prioridade*.
- As operações básicas relativas a uma *fila de prioridade* são:
 - ① Construção de uma fila de prioridade a partir de um conjunto com n itens.
 - ② Seleção do elemento de maior (menor) prioridade.
 - ③ Inserção de um novo elemento.
 - ④ Remoção do elemento de maior (menor) prioridade.
 - ⑤ Alteração do elemento de maior (menor) prioridade.

- TAD em que cada elemento do conjunto de dados está associado a uma *prioridade*.
- As operações básicas relativas a uma *fila de prioridade* são:
 - ① Construção de uma fila de prioridade a partir de um conjunto com n itens.
 - ② Seleção do elemento de maior (menor) prioridade.
 - ③ Inserção de um novo elemento.
 - ④ Remoção do elemento de maior (menor) prioridade.
 - ⑤ Alteração do elemento de maior (menor) prioridade.

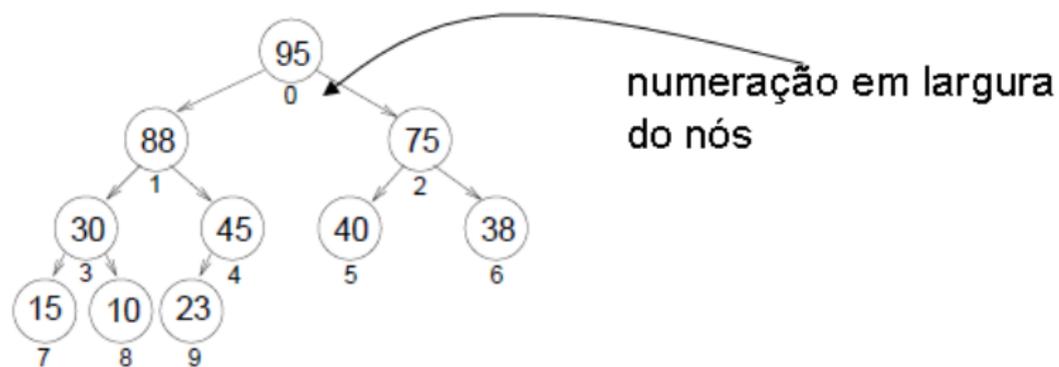
- Estruturalmente, uma *fila de prioridade* pode ser vista como uma árvore binária com as seguintes propriedades:

- 1 É uma árvore binária *completa* ou *quase completa*.
- 2 Em cada nó da árvore, o valor da chave é maior (menor) do que os valores das chaves dos filhos.

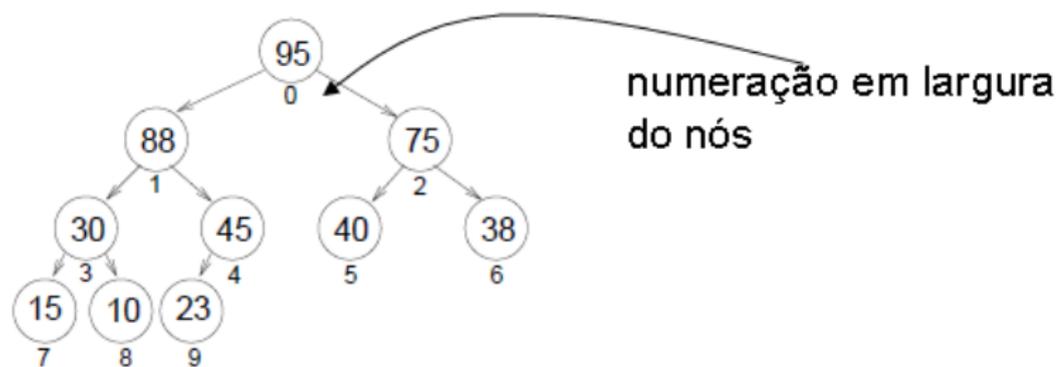
- Estruturalmente, uma *fila de prioridade* pode ser vista como uma árvore binária com as seguintes propriedades:

- 1 É uma árvore binária *completa* ou *quase completa*.
- 2 Em cada nó da árvore, o valor da chave é maior (menor) do que os valores das chaves dos filhos.

- Exemplo:



- Exemplo:



- Heap: Implementação em vetor

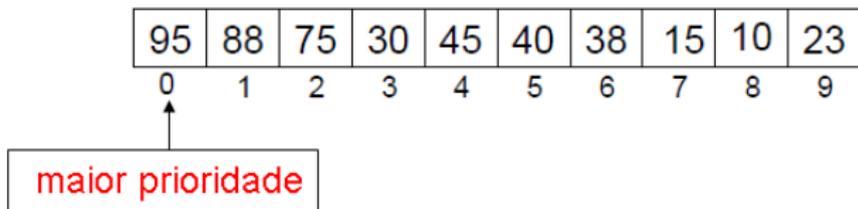
95	88	75	30	45	40	38	15	10	23
0	1	2	3	4	5	6	7	8	9

- Definição: Um *heap* é uma lista linear composta de elementos com chaves s_0, \dots, s_{n-1} , satisfazendo a seguinte propriedade:

$$s_{\lfloor \frac{i-1}{2} \rfloor} \geq s_i, \quad \forall 0 < i \leq n-1 \quad (\text{max-heap})$$

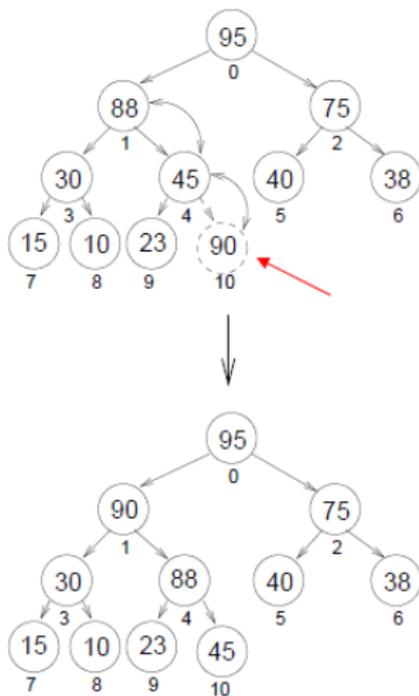
ou

$$s_{\lfloor \frac{i-1}{2} \rfloor} \leq s_i, \quad \forall 0 < i \leq n-1 \quad (\text{min-heap})$$



- Operações básicas sobre um heap:

- Subida ($O(\log n)$):



- Operações básicas sobre um heap:

- Subida em vetor ($O(\log n)$):

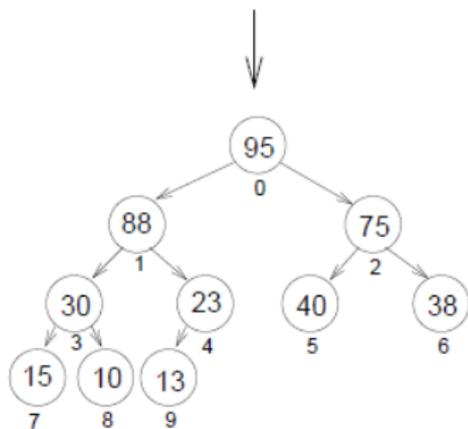
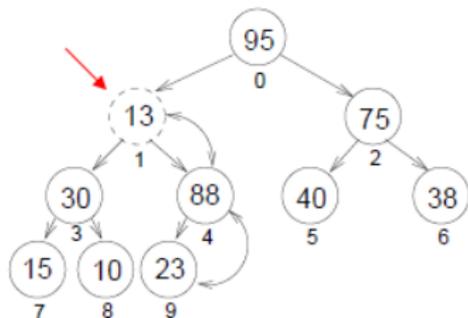
0	1	2	3	4	5	6	7	8	9	10
95	88	75	30	45	40	38	15	10	23	90

0	1	2	3	4	5	6	7	8	9	10
95	88	75	30	90	40	38	15	10	23	45

0	1	2	3	4	5	6	7	8	9	10
95	90	75	30	88	40	38	15	10	23	45

- Operações básicas sobre um heap:

- Descida ($O(\log n)$):



- Operações básicas sobre um heap:

– Descida em vetor ($O(\log n)$):

0	1	2	3	4	5	6	7	8	9
95	13	75	30	88	40	38	15	10	23

0	1	2	3	4	5	6	7	8	9
95	88	75	30	13	40	38	15	10	23

0	1	2	3	4	5	6	7	8	9
95	88	75	30	23	40	38	15	10	13

- Operação *Sobe* em C:

```
#define TAM_MAX 50
```

```
typedef struct {  
    T vetor[TAM_MAX];  
    int tam;  
} Heap;
```

```
void Sobe(Heap *h, int m) {  
    int j = (m-1)/2;  
    T x = (*h).vetor[m];  
  
    while ((m>0) && ((*h).vetor[j]<x)) {  
        (*h).vetor[m] = (*h).vetor[j];  
        m = j;  
        j = (j-1)/2;  
    }  
    (*h).vetor[m] = x;  
} /* Sobe */
```

- Operação *Desce* em C:

```
void Desce(Heap *h, int m) {
    int k = 2*m+1;
    T x = (*h).vetor[m];
    while (k < (*h).tam) {
        if ((k < ((*h).tam) - 1) && ((*h).vetor[k] < (*h).vetor[k+1]))
            k++;
        if (x < (*h).vetor[k]) {
            (*h).vetor[m] = (*h).vetor[k];
            m = k;
            k = 2*k+1;
        } else
            break;
    }
    (*h).vetor[m] = x;
} /* Desce */
```

- Construção de *heaps* (recebe vetor h e constrói heap): $O(n \log n)$

```
void ConstroiHeap1(Heap *h) {  
    int i;  
    for (i=1; i<(*h).tam; i++)  
        Sobe(h,i);  
} /* ConstroiHeap1 */  
|  
void ConstroiHeap2(Heap *h) {  
    int i;  
    for (i=((*h).tam-2)/2; i>=0; i--)  
        Desce(h,i);  
} /* ConstroiHeap2 */
```

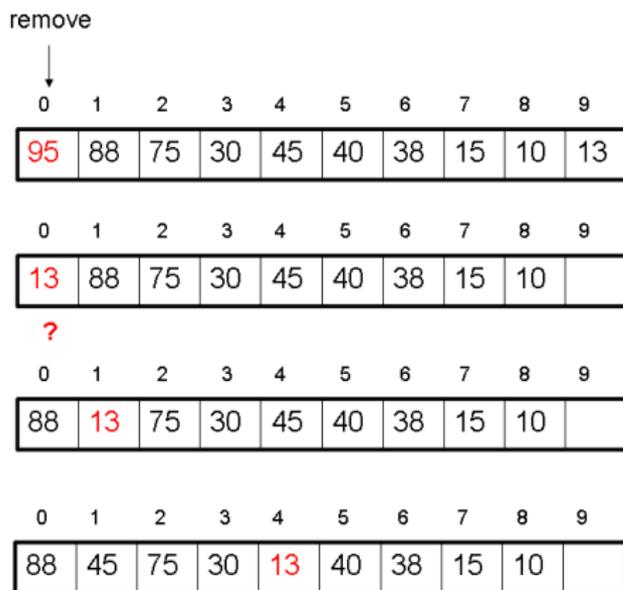
- Inserção em *heaps*: $O(\log n)$

```
void InsereHeap(Heap *h, T x) {  
    vetor[(*h).tam] = x ;    /* insere no final do vetor */  
    ((*h).tam)++;  
    Sobe(h, ((*h).tam-1) ; /* coloca x na posicao correta */  
} /* InsereHeap */
```

- Remoção em *heaps*: $O(\log n)$
 - Sempre feita na raiz, retirando-se o elemento de maior prioridade.
 - A lista passa a ter $n - 1$ elementos.
 - O último elemento do conjunto é colocado no início da lista para forçar a aparição do novo elemento de maior prioridade.

- Remoção em *heaps*: $O(\log n)$

- Sempre feita na raiz, retirando-se o elemento de maior prioridade.
- A lista passa a ter $n - 1$ elementos.
- O último elemento do conjunto é colocado no início da lista para forçar a aparição do novo elemento de maior prioridade.



- Remoção em *heaps* em C: $O(\log n)$

```
void RemoveHeap (Heap *h, T *x) {
    *x = (*h).vetor[0] ;      /* remove a raiz */
    ((*h).tam)-- ;
    (*h).vetor[0] = (*h).vetor[(*h).tam] ;
    Desce(h,0) ;             /* desce chave na posicao 0 */
} /* RemoveHeap */
```

- Implementação por *heap*:
 - Seleção: $O(1)$
 - Inserção: $O(\log n)$
 - Remoção: $O(\log n)$
 - Alteração: $O(\log n)$
 - Construção: $O(n \log n)$

- Implementação por *heap*:
 - Seleção: $O(1)$
 - Inserção: $O(\log n)$
 - Remoção: $O(\log n)$
 - Alteração: $O(\log n)$
 - Construção: $O(n \log n)$

- Implementação por *heap*:
 - Seleção: $O(1)$
 - Inserção: $O(\log n)$
 - Remoção: $O(\log n)$
 - Alteração: $O(\log n)$
 - Construção: $O(n \log n)$

- Implementação por *heap*:
 - Seleção: $O(1)$
 - Inserção: $O(\log n)$
 - Remoção: $O(\log n)$
 - Alteração: $O(\log n)$
 - Construção: $O(n \log n)$

- Implementação por *heap*:
 - Seleção: $O(1)$
 - Inserção: $O(\log n)$
 - Remoção: $O(\log n)$
 - Alteração: $O(\log n)$
 - Construção: $O(n \log n)$

- Implementação por lista não ordenada:
 - Seleção: $O(n)$
 - Inserção: $O(1)$
 - Remoção: $O(n)$
 - Alteração: $O(n)$
 - Construção: $O(n)$

- Implementação por lista não ordenada:
 - Seleção: $O(n)$
 - Inserção: $O(1)$
 - Remoção: $O(n)$
 - Alteração: $O(n)$
 - Construção: $O(n)$

- Implementação por lista não ordenada:
 - Seleção: $O(n)$
 - Inserção: $O(1)$
 - Remoção: $O(n)$
 - Alteração: $O(n)$
 - Construção: $O(n)$

- Implementação por lista não ordenada:
 - Seleção: $O(n)$
 - Inserção: $O(1)$
 - Remoção: $O(n)$
 - Alteração: $O(n)$
 - Construção: $O(n)$

- Implementação por lista não ordenada:
 - Seleção: $O(n)$
 - Inserção: $O(1)$
 - Remoção: $O(n)$
 - Alteração: $O(n)$
 - Construção: $O(n)$

- Implementação por lista não ordenada:
 - Seleção: $O(n)$
 - Inserção: $O(1)$
 - Remoção: $O(n)$
 - Alteração: $O(n)$
 - Construção: $O(n)$

- Implementação por lista ordenada:

- Seleção: $O(1)$
- Inserção: $O(n)$
- Remoção: $O(1)$
- Alteração: $O(n)$
- Construção: $O(n \log n)$

⇒ Estas implementações apresentam operações importantes de complexidade $O(n)$.

- Implementação por lista ordenada:

- Seleção: $O(1)$

- Inserção: $O(n)$

- Remoção: $O(1)$

- Alteração: $O(n)$

- Construção: $O(n \log n)$

⇒ Estas implementações apresentam operações importantes de complexidade $O(n)$.

- Implementação por lista ordenada:
 - Seleção: $O(1)$
 - Inserção: $O(n)$
 - Remoção: $O(1)$
 - Alteração: $O(n)$
 - Construção: $O(n \log n)$

⇒ Estas implementações apresentam operações importantes de complexidade $O(n)$.

- Implementação por lista ordenada:
 - Seleção: $O(1)$
 - Inserção: $O(n)$
 - Remoção: $O(1)$
 - Alteração: $O(n)$
 - Construção: $O(n \log n)$

⇒ Estas implementações apresentam operações importantes de complexidade $O(n)$.

- Implementação por lista ordenada:
 - Seleção: $O(1)$
 - Inserção: $O(n)$
 - Remoção: $O(1)$
 - Alteração: $O(n)$
 - Construção: $O(n \log n)$

⇒ Estas implementações apresentam operações importantes de complexidade $O(n)$.

- Implementação por lista ordenada:
 - Seleção: $O(1)$
 - Inserção: $O(n)$
 - Remoção: $O(1)$
 - Alteração: $O(n)$
 - Construção: $O(n \log n)$

⇒ Estas implementações apresentam operações importantes de complexidade $O(n)$.

- Implementação por lista ordenada:
 - Seleção: $O(1)$
 - Inserção: $O(n)$
 - Remoção: $O(1)$
 - Alteração: $O(n)$
 - Construção: $O(n \log n)$

⇒ Estas implementações apresentam operações importantes de complexidade $O(n)$.

Outra aplicação do *heap*

- O *Heapsort*

– Algoritmo:

- 1 Construir uma fila de prioridade com os elementos a serem ordenados,
- 2 Aplicar remoções sucessivas dispondo o primeiro elemento (maximal ou minimal) na sua posição correta (a última casa do vetor contendo o heap).

- ```
/* O Heapsort */
CrontroiHeap(h)
m = tam_heap-1
enquanto m > 0 faça:
 Troque h(0) com h(m)
 m = m - 1
 Desce(0, m)
```

# Outra aplicação do *heap*

- O *Heapsort*

– Algoritmo:

- 1 Construir uma fila de prioridade com os elementos a serem ordenados,
- 2 Aplicar remoções sucessivas dispondo o primeiro elemento (maximal ou minimal) na sua posição correta (a última casa do vetor contendo o heap).

- ```
/* O Heapsort */  
CrontroiHeap(h)  
m = tam_heap-1  
enquanto m > 0 faça:  
    Troque h(0) com h(m)  
    m = m - 1  
    Desce(0, m)
```

Outra aplicação do *heap*

- O *Heapsort*

– Algoritmo:

- 1 Construir uma fila de prioridade com os elementos a serem ordenados,
- 2 Aplicar remoções sucessivas dispondo o primeiro elemento (maximal ou minimal) na sua posição correta (a última casa do vetor contendo o heap).

- ```
/* O Heapsort */
CrontroiHeap(h)
m = tam_heap-1
enquanto m > 0 faça:
 Troque h(0) com h(m)
 m = m - 1
 Desce(0, m)
```

# Outra aplicação do *heap*

- O *Heapsort*

– Algoritmo:

- 1 Construir uma fila de prioridade com os elementos a serem ordenados,
- 2 Aplicar remoções sucessivas dispondo o primeiro elemento (maximal ou minimal) na sua posição correta (a última casa do vetor contendo o heap).

- ```
/* O Heapsort */  
CrontroiHeap(h)  
m = tam_heap-1  
enquanto m > 0 faça:  
    Troque h(0) com h(m)  
    m = m - 1  
    Desce(0, m)
```

- O Heapsort em C: $O(n \log n)$

```
void HeapSort(Heap *h) {
    int i, n = (*h).tam;
    for (i=(n-2)/2; i>=0; i--) /* constrói heap */
        Desce(h,i);
    for (i=n-1; i>0; i--) { /* ordena */
        T t = (*h).vetor[0];
        (*h).vetor[0] = (*h).vetor[i];
        (*h).vetor[i] = t;
        (*h).tam--;
        Desce(h,0);
    }
    (*h).tam = n;
} /* HeapSort */
```

- Exemplo:

Dados a serem ordenados:

0	1	2	3	4	5
10	8	7	3	20	15

- Exemplo:

Heap:

0	1	2	3	4	5
3	7	8	10	20	15

- Exemplo:

Troca 3 com 15 e Desce até 4:

0	1	2	3	4	5
15	7	8	10	20	3

- Exemplo:

0	1	2	3	4	5
7	15	8	10	20	3

- Exemplo:

0	1	2	3	4	5
7	10	8	15	20	3

- Exemplo:

Troca 7 com 20 e Desce até 3:

0	1	2	3	4	5
20	10	8	15	7	3

- Exemplo:

0	1	2	3	4	5
8	10	20	15	7	3

- Exemplo:

Troca 8 com 15 e Desce até 2:

0	1	2	3	4	5
15	10	20	8	7	3

- Exemplo:

0	1	2	3	4	5
10	15	20	8	7	3

- Exemplo:

Troca 10 com 20 e Desce até 1:

0	1	2	3	4	5
20	15	10	8	7	3

- Exemplo:

0	1	2	3	4	5
15	20	10	8	7	3

- Exemplo:

Troca 15 com 20 e desce até 0 (FIM!!!):

0	1	2	3	4	5
20	15	10	8	7	3

Referências:

- Jayme L. Szwarcfiter e Lilian Markenzon. Estruturas de Dados e seus Algoritmos. Editora LTC, 1994.
- Nívio Ziviani. Projeto de Algoritmos com Implementações em Pascal e C, Thomson Learning, 2004.
- Apostila dos professores Thomasz e Lucchesi.