

# Fila de Prioridades e Heap

Prof<sup>a</sup>. Rose Yuri Shimizu

# Roteiro

1 Fila de Prioridades

2 Heap Sort

3 Intro Sort

# Fila de Prioridades - TAD

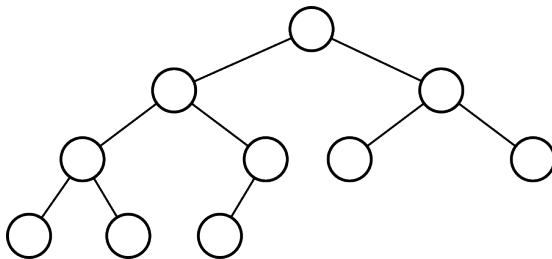
- É um tipo abstrato de dados (TAD)
  - ▶ Representa valores
  - ▶ Acessadas por um conjunto de operações (interface)
- Existem operações que envolvem um grande volume de informações que precisam de alguma ordenação
  - ▶ Não necessariamente precisam estar totalmente ordenados
    - ★ O importante é saber qual tem a maior prioridade
  - ▶ Não necessariamente precisam processar todos os dados
    - ★ Conforme novos dados forem coletados, atualiza-se a fila de prioridades
  - ▶ Muitos dados que são rankeados conforme um critério em que o mais importante é saber quem está no topo
  - ▶ Exemplos: mineração de dados, caminhos em grafos (pesos nas arestas - verificar adjacentes na decisão do caminho)
- A fila pode ser com prioridade máxima (maior chave, maior prioridade) ou mínima (menor valor, maior prioridade)

# Fila de Prioridades - Estrutura de dados

- Vetores e listas encadeadas:
  - ▶ Não ordenado:
    - ★ Processo corresponde a executar um Selection Sort
    - ★ Selecionar o de maior prioridade e colocar no início
  - ▶ Ordenado
    - ★ Processo corresponde a executar um Insertion Sort
    - ★ A cada novo item, posicionar comparando com os seus antecessores
- Heap binária (heap): árvore binária completa com vetores

# Fila de Prioridades - Heap

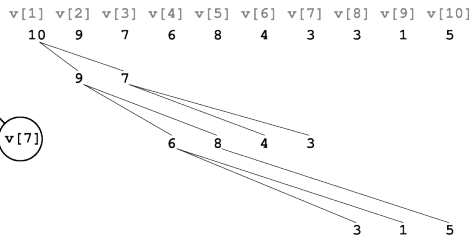
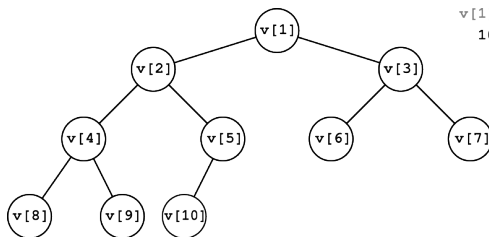
- Forma uma árvore binária completa:
  - ▶ Todos os níveis exceto o último estão cheios
  - ▶ Os nós do último nível estão o mais a esquerda possível



- Raiz: chave de maior prioridade
- Não ordena por completo, só garante-se que:
  - ▶ Quanto mais próximo à raiz, maior a prioridade
  - ▶ Cada nó possui filhos com valores menores ou iguais

# Fila de Prioridades - Heap

- Representada por um vetor:
  - ▶ Eficiente para as operações básicas (logarítmico) da fila de prioridades
  - ▶ Representação sequencial da árvore: facilidade em deixá-la completa
  - ▶ Acesso direto aos nós
  - ▶ Níveis da árvore acessada pelos seus índices
    - ★ Raiz: posição 1
    - ★ Filhos: 2 e 3
    - ★ Netos: 4, 5, 6 e 7
    - ★ E assim por diante.



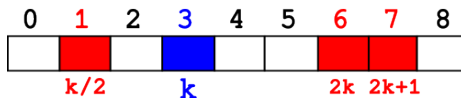
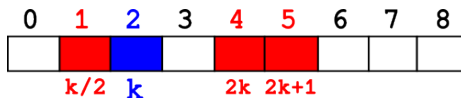
# Fila de Prioridades - Heap

- Navegação trivial para cima e baixo:

- ▶ Simples operação aritmética
- ▶ Sendo um nó na posição  $k$

- ★ pai:  $\lfloor \frac{k}{2} \rfloor$

- ★ filhos:  $2k$  e  $2k + 1$



- Tamanho  $N$  em um vetor  $pq[]$

- ▶  $pq[N+1]$ :  $pq[1..N]$
- ▶ Não utiliza-se a posição  $pq[0]$  (??)
- ▶ E se utilizar?

- ★ pai:  $\lfloor \frac{k-1}{2} \rfloor$

- ★ filhos:  $2k + 1$  e  $2k + 2$

# Fila de Prioridades - Heap

- Interface (manipulação da fila):

- ▶ PQinit(int maxN): criar uma fila de prioridades com capacidade máxima inicial
- ▶ PQempty(): testar se está vazia
- ▶ PQinsert(Item v): inserir uma chave
- ▶ PQdelmax(): retornar e remover (maior prioridade)

```
1 //static: acessível somente no arquivo do código
   fonte
2 static Item *pq;
3 static int N;
4
5 void PQinit(int maxN) {
6     pq = malloc(sizeof(Item)*(maxN+1));
7     N = 0;
8 }
9
10 int PQempty() {
11     return N==0;
12 }
```



# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap

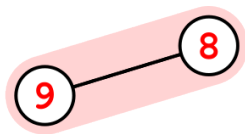


Inserir 8      [8]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap

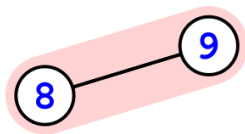


Inserir 9      [8, 9]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



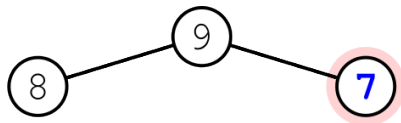
Restaurando 9

[9, 8]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



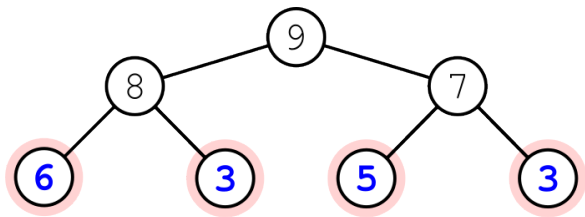
Inserir 7

[9, 8, 7]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



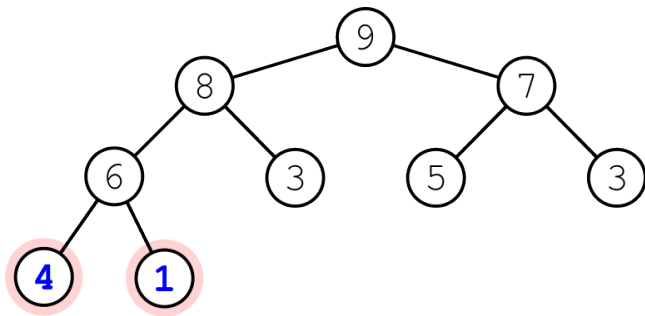
Inserir 6,3,5,3

[9, 8, 7, 6, 3, 5, 3]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



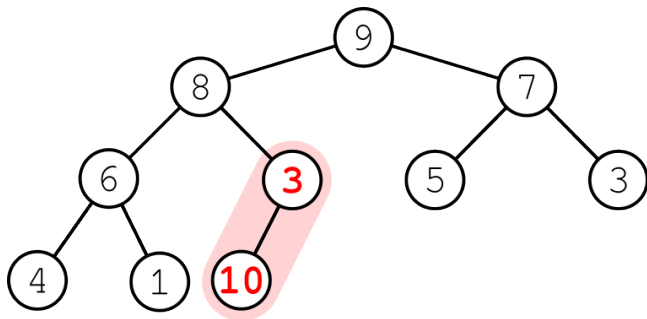
Inserir 4,1

[9, 8, 7, 6, 3, 5, 3, 4, 1]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



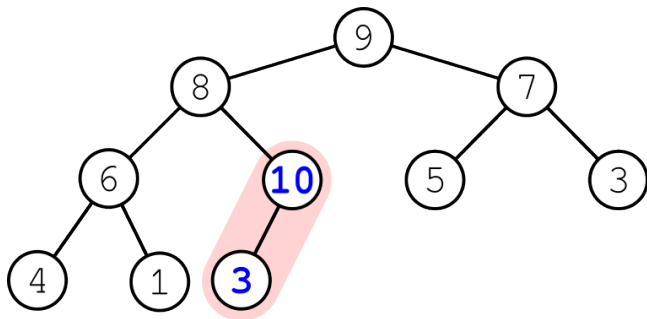
Inserir 10

[9, 8, 7, 6, 3, 5, 3, 4, 1, 10]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



Restaurando 10

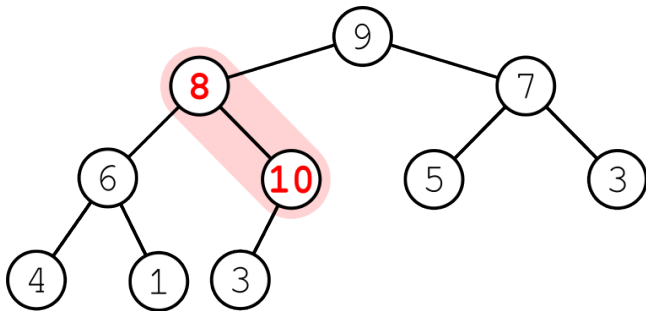
[9, 8, 7, 6, 3, 5, 3, 4, 1, 10]



# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



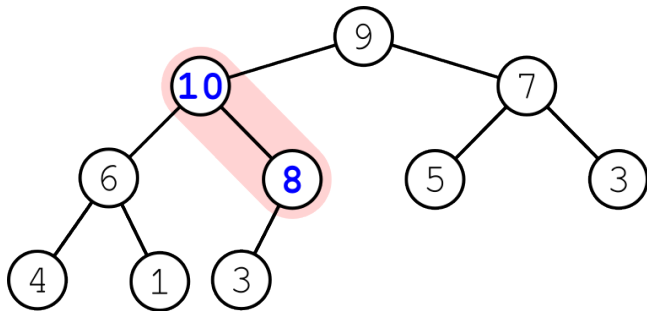
Restaurando 10

[9, 8, 7, 6, 10, 5, 3, 4, 1, 3]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



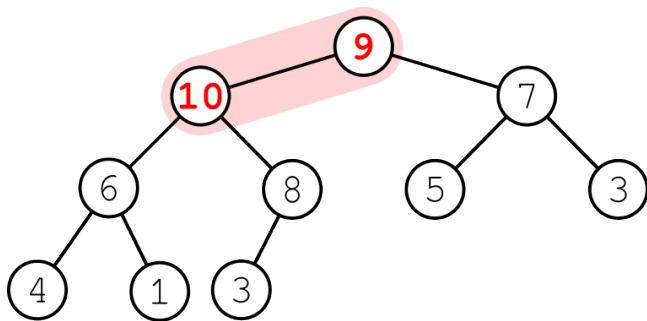
Restaurando 10

[9, 8, 7, 6, 10, 5, 3, 4, 1, 3]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



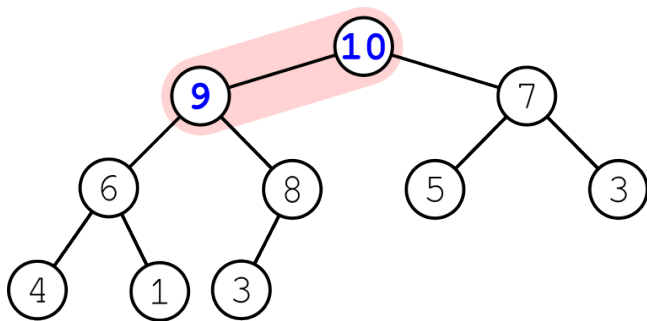
Restaurando 10

[9, 10, 7, 6, 8, 5, 3, 4, 1, 3]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



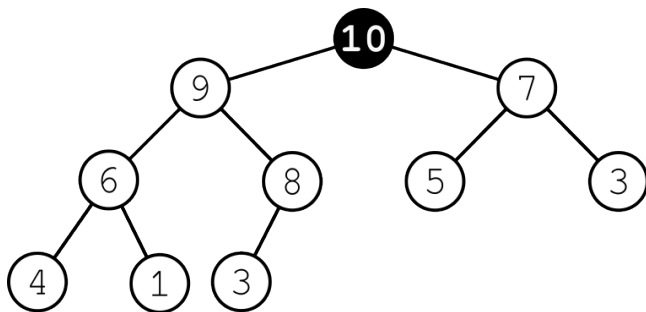
Restaurando 10

[10, 9, 7, 6, 8, 5, 3, 4, 1, 3]

# Fila de Prioridades - Heap - Inserção

- Inserção

- ▶ Inserção nas folhas da heap
- ▶ Restauração/conserto: subindo na heap



# Fila de Prioridades - Heap - Inserção

- Adicionar uma nova chave no fim do vetor
- Restaura a ordenação da heap: bottom-up (swim - fixUp)
  - ▶ Flutue (swap) caso a chave seja maior que seu pai
  - ▶ Recursivamente, flutue até um pai maior ou a raiz

```
1 void fixUp(int k)
2 {
3     //troque com o pai se for maior
4     while(k>1 && less(pq[k/2],pq[k]))
5     {
6         exch(pq[k], pq[k/2]);
7         k = k/2; //verifique o avô
8     }
9 }
10
11 void PQinsert(Item v) {
12     pq[++N] = v;
13     fixUp(N);
14 }
```

# Fila de Prioridades - Heap - Inserção

- Complexidade:  $1 + \log N$  comparações -  $O(\log N)$

```
1 void fixUp(int k)
2 {
3     //k até 1 - reduzindo metade por iteração
4     //altura da árvore ~ log k
5     //quantidade de termos de uma PG
6     while(k>1 && less(pq[k/2],pq[k]))
7     {
8         exch(pq[k], pq[k/2]);
9         k = k/2;
10    }
11 }
```

## Fila de Prioridades - Heap - Inserção

$$a_i = a_1 * q^{(i-1)}$$

$$1 = k * \frac{1}{2}^{(i-1)}$$

$$\log_2 1 = \log_2 k + \log_2 \frac{1}{2}^{(i-1)}$$

$$\log_2 1 = \log_2 k + (i-1)(\log_2 1 - \log_2 2)$$

$$0 = \log_2 k + (i-1)(0-1)$$

$$0 = \log_2 k - i + 1$$

$$i = \log_2 k + 1$$

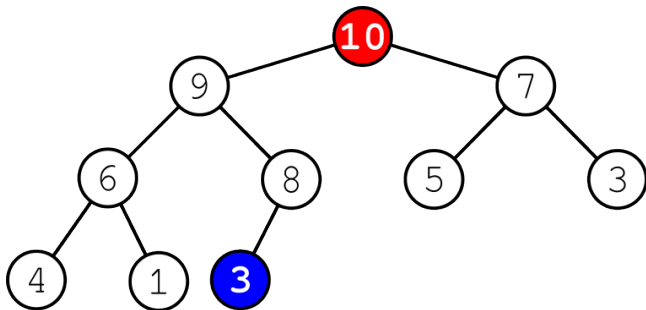
$$F(k) = \log_2 k + 1$$



# Fila de Prioridades - Heap - Remoção

- Remoção

- ▶ Remover qual elemento??
- ▶ Substituir a raiz por uma folha
- ▶ Restauração/conserto: descendo na heap



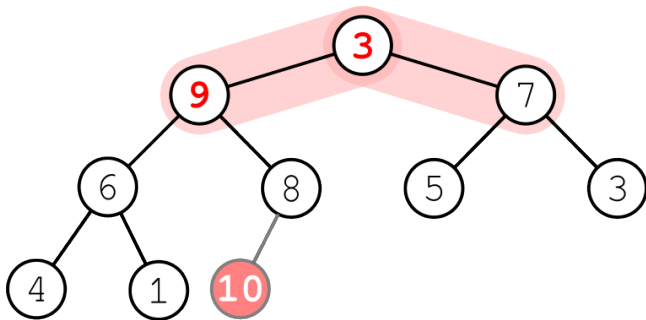
Remover 10

[**10**, 9, 7, 6, 8, 5, 3, 4, 1, **3**]

# Fila de Prioridades - Heap - Remoção

- Remoção

- ▶ Remover qual elemento??
- ▶ Substituir a raiz por uma folha
- ▶ Restauração/conserto: descendo na heap

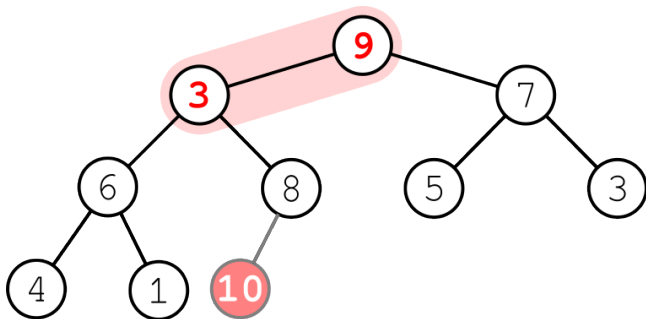


Restaurando 3      [3, 9, 7, 6, 8, 5, 3, 4, 1, 10]

# Fila de Prioridades - Heap - Remoção

- Remoção

- ▶ Remover qual elemento??
- ▶ Substituir a raiz por uma folha
- ▶ Restauração/conserto: descendo na heap



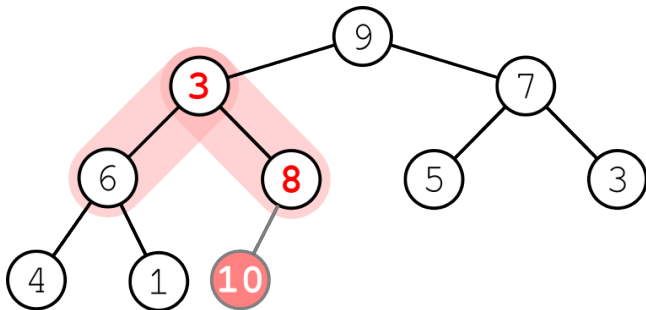
Restaurando 3

[9, 3, 7, 6, 8, 5, 3, 4, 1, 10]

# Fila de Prioridades - Heap - Remoção

- Remoção

- ▶ Remover qual elemento??
- ▶ Substituir a raiz por uma folha
- ▶ Restauração/conserto: descendo na heap



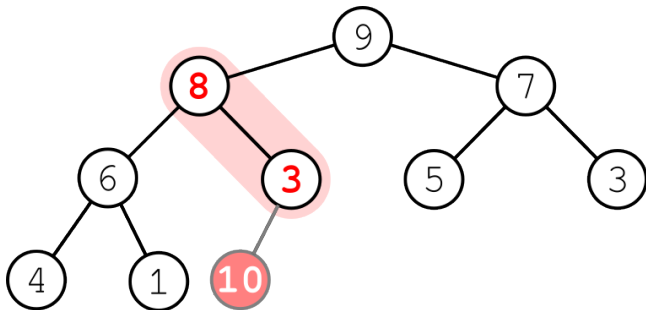
Restaurando 3

[9, 3, 7, 6, 8, 5, 3, 4, 1, 10]

# Fila de Prioridades - Heap - Remoção

- Remoção

- ▶ Remover qual elemento??
- ▶ Substituir a raiz por uma folha
- ▶ Restauração/conserto: descendo na heap



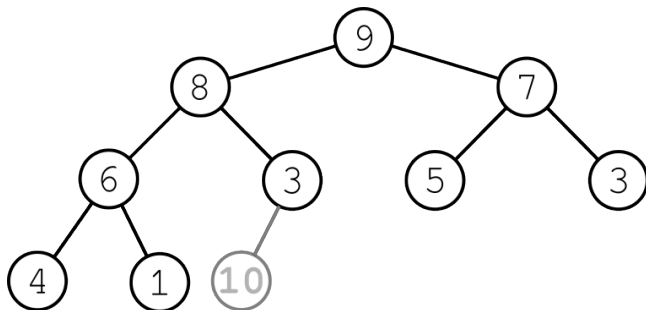
Restaurando 3

[9, **8**, 7, 6, **3**, 5, 3, 4, 1, 10]

# Fila de Prioridades - Heap - Remoção

- Remoção

- ▶ Remover qual elemento??
- ▶ Substituir a raiz por uma folha
- ▶ Restauração/conserto: descendo na heap



Restaurando 3      [9, 8, 7, 6, 3, 5, 3, 4, 1, 10]

# Fila de Prioridades - Heap - Remoção

- Troca a raiz com o último elemento
- Restaura a ordenação da heap: top-down (sink - fixDown)
  - ▶ Afunde caso a chave seja menor que um ou ambos os filhos
    - ★ Swap com o maior filho
  - ▶ Recursivamente, afunde a chave até que ambos os filhos sejam menores (ou iguais) ou atingir a base

```
1 void fixDown(int k, int N)
2 {
3     int j;
4
5     //tem filho?
6     while(2*k <= N) {
7         j = 2*k;
8
9         //qual o maior filho?
10        if(j<N && less(pq[j], pq[j+1])) j++;
11    }
```

## Fila de Prioridades - Heap - Remoção

```
12     //pai maior que o maior filho?
13     if(!less(pq[k], pq[j])) break;
14
15     //senão, afunde (troque com o filho)
16     exch(pq[k], pq[j]);
17     k = j;
18 }
19 }
20
21 Item PQdelmax() {
22     exch(pq[1], pq[N]); //troque topo -> ultimo
23     fixDown(1, N-1);    //reposicione
24     return pq[N--];
25 }
```



# Fila de Prioridades - Heap - Remoção

- Complexidade:  $2 \log N$  comparações

```
1 void fixDown(int k, int N) {  
2     int j;  
3     //2*k até N - dobrando a cada iteração  
4     //altura da árvore ~ log k  
5     //quantidade de termos de uma PG  
6     while(2*k <= N) { //~ log k  
7         j = 2*k;  
8         if(j < N && less(pq[j], pq[j+1])) j++; //1  
9         if(!less(pq[k], pq[j])) break; //1  
10        exch(pq[k], pq[j]);  
11        k = j;  
12    }  
13 }
```

## Fila de Prioridades - Heap - Remoção

$$a_i = a_1 * q^{(i-1)}$$

$$N = 2 * k * 2^{(i-1)} : k : raiz = 1$$

$$N = 2 * 2^{(i-1)}$$

$$\log_2 N = \log_2 2^i$$

$$\log_2 N = i \log_2 2$$

$$i = \log_2 N$$

$$F(N) = (\log_2 N) * (1 + 1) = 2 * \log_2 N$$

# Fila de Prioridades - Heap - Alterar prioridade

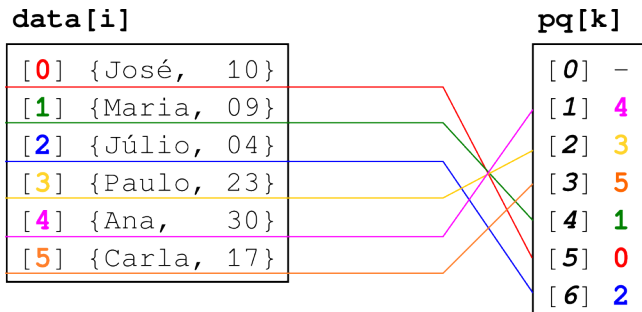
- Se temos o índice na fila de prioridades é trivial

```
1 void PQchange(int k, int valor)
2 {
3     if (v[k] < valor) {
4         v[k] = valor;
5         fixUp(k);
6     } else {
7         v[k] = valor;
8         fixDown(k, N);
9     }
10 }
```

- Se não tem o acesso direto:
  - ▶ Cuidado com as buscas lineares
  - ▶ Solução: Lista de índices

# Fila de Prioridades - Heap - Lista de índices

- Vetor de dados: dados a serem organizados
  - ▶  $\text{data}[i]$
- Fila de prioridades: índices do vetor de dados
  - ▶  $\text{pq}[k]$  : armazenam os índices  $i$ 's dos dados
- Se alterar  $\text{data}[i]$ , com atualizá-lo em  $\text{pq}[k]$ ?
  - ▶ Percorrer  $\text{pq}[]$  à procura de  $i$ ?



# Fila de Prioridades - Heap - Lista de índices

- Vetor de dados:  $data[i]$
- Fila de prioridades:  $pq[k]$
- Se alterar  $data[i]$ , com atualizá-lo em  $pq[k]$ ?
- Lista de índices:
  - ▶  $qp[i]$  :
    - ★ Armazenam os índices  $k$ 's da fila de prioridades
    - ★ Índices  $i$ 's do vetor de dados
  - ▶ Acesso direto à fila de prioridades através do índice de dados
    - ★  $pq[k] = i$
    - ★  $qp[i] = k \leftrightarrow pq[qp[i]] = i \leftrightarrow pq[k] = i$

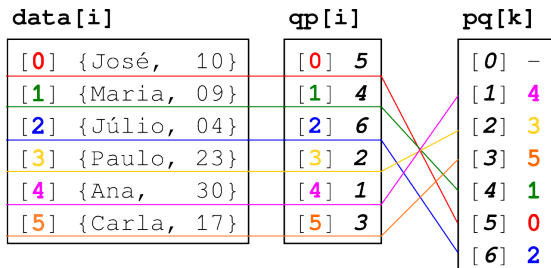
$data[i]$	$qp[i]$	$pq[k]$
[0] {José, 10}	[0] 5	[0] -
[1] {Maria, 09}	[1] 4	[1] 4
[2] {Júlio, 04}	[2] 6	[2] 3
[3] {Paulo, 23}	[3] 2	[3] 5
[4] {Ana, 30}	[4] 1	[4] 1
[5] {Carla, 17}	[5] 3	[5] 0
		[6] 2

# Fila de Prioridades - Heap - Lista de índices

```
1 typedef struct {
2     char nome[20];
3     int chave;
4 } Item;
5
6 static Item *pq;
7 static Item *qp;
8 static int N;
9 Item *data;
10
11 //k índice do elemento na lista de dados
12 //data[k]
13 void PQinsert(int k) {
14     qp[k] = ++N; //data[k] na última posição da
15     fila
16     pq[N] = k;   //inserir na última posição
17     fixUp(N);    //consertar a heap
18                 //pq[N/2].chave < pq[N].chave
19 }
```

# Fila de Prioridades - Heap - Lista de índices

- Prioridade de José mudou para 50
  - ▶  $i = 0$
  - ▶  $data[i].chave = 50$
- Portanto, sua posição na fila deve ser alterada:
  - ▶  $PQchange(0)$ : alteração em  $data[i] \rightarrow i=0$
  - ▶  $data[0]$  está posição 5 na fila de prioridades:
    - ★  $data[0] \rightarrow qp[0] = 5 \rightarrow pq[qp[0]] \rightarrow pq[5]$
  - ▶ Consertar a heap
    - ★  $fixUp(qp[0])$
    - ★  $fixDown(qp[0], N)$



# Fila de Prioridades - Heap - Lista de índices

- Prioridade de José mudou para 50
  - ▶  $i = 0$
  - ▶  $data[i].chave = 50$
- Portanto, sua posição na fila deve ser alterada:
  - ▶  $PQchange(0)$ : alteração em  $data[i] \rightarrow i=0$
  - ▶  $data[0]$  está posição 5 na fila de prioridades:
    - ★  $data[0] \rightarrow qp[0] = 5 \rightarrow pq[qp[0]] \rightarrow pq[5]$
  - ▶ Consertar a heap
    - ★  $fixUp(qp[0])$
    - ★  $fixDown(qp[0], N)$

$data[i]$	$qp[i]$	$pq[k]$
[0] {José, 50}	[0] 5	[0] -
[1] {Maria, 09}	[1] 4	[1] 4
[2] {Júlio, 04}	[2] 6	[2] 3
[3] {Paulo, 23}	[3] 2	[3] 5
[4] {Ana, 30}	[4] 1	[4] 1
[5] {Carla, 17}	[5] 3	[5] 0
		[6] 2



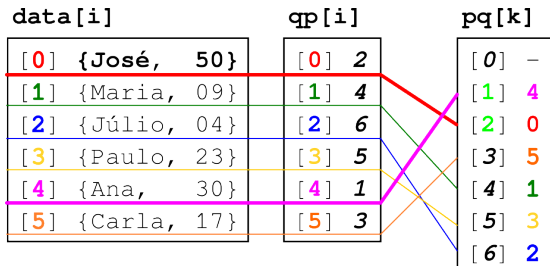
# Fila de Prioridades - Heap - Lista de índices

- Prioridade de José mudou para 50
  - ▶  $i = 0$
  - ▶  $\text{data}[i].\text{chave} = 50$
- Portanto, sua posição na fila deve ser alterada:
  - ▶  $\text{PQchange}(0)$ : alteração em  $\text{data}[i] \rightarrow i=0$
  - ▶  $\text{data}[0]$  está posição 5 na fila de prioridades:
    - ★  $\text{data}[0] \rightarrow \text{qp}[0] = 5 \rightarrow \text{pq}[\text{qp}[0]] \rightarrow \text{pq}[5]$
  - ▶ Consertar a heap
    - ★  $\text{fixUp}(\text{qp}[0])$
    - ★  $\text{fixDown}(\text{qp}[0], N)$

$\text{data}[i]$	$\text{qp}[i]$	$\text{pq}[k]$
[0] {José, 50}	[0] 2	[0] -
[1] {Maria, 09}	[1] 4	[1] 4
[2] {Júlio, 04}	[2] 6	[2] 0
[3] {Paulo, 23}	[3] 5	[3] 5
[4] {Ana, 30}	[4] 1	[4] 1
[5] {Carla, 17}	[5] 3	[5] 3
		[6] 2

# Fila de Prioridades - Heap - Lista de índices

- Prioridade de José mudou para 50
  - ▶  $i = 0$
  - ▶  $data[i].chave = 50$
- Portanto, sua posição na fila deve ser alterada:
  - ▶  $PQchange(0)$ : alteração em  $data[i] \rightarrow i=0$
  - ▶  $data[0]$  está posição 5 na fila de prioridades:
    - ★  $data[0] \rightarrow qp[0] = 5 \rightarrow pq[qp[0]] \rightarrow pq[5]$
  - ▶ Consertar a heap
    - ★  $fixUp(qp[0])$
    - ★  $fixDown(qp[0], N)$



# Fila de Prioridades - Heap - Lista de índices

- Prioridade de José mudou para 50
  - ▶  $i = 0$
  - ▶  $data[i].chave = 50$
- Portanto, sua posição na fila deve ser alterada:
  - ▶  $PQchange(0)$ : alteração em  $data[i] \rightarrow i=0$
  - ▶  $data[0]$  está posição 5 na fila de prioridades:
    - ★  $data[0] \rightarrow qp[0] = 5 \rightarrow pq[qp[0]] \rightarrow pq[5]$
  - ▶ Consertar a heap
    - ★  $fixUp(qp[0])$
    - ★  $fixDown(qp[0], N)$

$data[i]$	$qp[i]$	$pq[k]$
[0] {José, 50}	[0] 1	[0] -
[1] {Maria, 09}	[1] 4	[1] 0
[2] {Júlio, 04}	[2] 6	[2] 4
[3] {Paulo, 23}	[3] 5	[3] 5
[4] {Ana, 30}	[4] 2	[4] 1
[5] {Carla, 17}	[5] 0	[5] 3
		[6] 2

```

1 void PQchange(int i) {
2     //atualizar data[i] na fila
3     //data[i] está na posição qp[i]
4     fixUp(qp[i]);
5     fixDown(qp[i], N);
6 }
7
8 void exch(int a, int b) {
9     //atualizar lista de índices
10    //data[a] trocou de lugar na fila com data[b]
11    int k = qp[a];
12    qp[a] = qp[b];
13    qp[b] = k;
14
15    //atualizar fila de prioridades
16    //data[a] está na posição qp[a] da fila
17    pq[qp[a]] = a;
18
19    //data[b] está na posição qp[b] da fila
20    pq[qp[b]] = b;
21 }

```

# Fila de Prioridades - Heap - Várias filas

```
1  /*****  
2  /* Implementacao com array */  
3  /* Varias filas          */  
4  *****/  
5  typedef int Item;  
6  typedef struct {  
7      Item *qp;  
8      Item *pq;  
9      int N;  
10 }PQueue;  
11  
12 PQueue *PQinit(int);  
13 int PQempty(PQueue*);  
14 void PQinsert(PQueue*, Item);  
15 Item PQdelmax(PQueue*);  
16 void PQchange(PQueue*, int);  
17  
18 void fixUp(PQueue*, int);  
19 void fixDown(PQueue*, int);
```

# Roteiro

1 Fila de Prioridades

2 Heap Sort

3 Intro Sort

# Algoritmos de Ordenação Eficientes - Heap Sort

- Construir e destrui a heap da fila de prioridades
- Usar as filas de prioridades para ordenar elementos
  - ▶ Fase 1: construção da heap-ordenada (fila prioridades)
    - ★ Topo é o de maior prioridade
    - ★ Quanto mais próximo ao topo, maior a prioridade
    - ★ Não há garantia de ordenação de todos os itens
  - ▶ Fase 2: ordenação por remoção

# Algoritmos de Ordenação Eficientes - Heap Sort - versão 1

- Usa-se somente a interface da TAD fila de prioridades
- Criar uma fila de prioridades
  - ▶ Utiliza-se espaço extra
- Fase 1: construção da heap
  - ▶ Construção da heap por inserção
    - ★ Varredura da esquerda para direita
    - ★ fixUp para posicionar na heap
    - ★ Custo proporcional a  $2 * N * \log N$
- Fase 2: ordenação (decrecente)
  - ▶ Ordenação por remoção (maior prioridade)
    - ★ Reorganização da fila de prioridades
    - ★ Cada item removido volta para o vetor original



# Algoritmos de Ordenação Eficientes - Heap Sort

```
1 void PQsort(Item *v, int l, int r) {  
2     PQinit(r-l+1);  
3     for(int k=l; k<=r; k++)  
4     {  
5         PQinsert(v[k]);  
6     }  
7     for(int k=r; k>=l; k--)  
8     {  
9         v[k] = PQdelmax();  
10    }  
11 }
```

# Algoritmos de Ordenação Eficientes - Heap Sort - versão 2

- Usar diretamente as funções exclusivas da TAD fila de prioridades:  
fixUp(swim), fixDown(sink)
- Não há a necessidade de espaços extras
  - ▶ Vetor original é utilizado para construir a heap
- Fase 1: construção da heap
  - ▶ Varredura da direita para esquerda
  - ▶ fixDown para preservar a heap-ordenada
  - ▶ Cada fixDown, constrói uma sub-heap
    - ★ Cada posição no vetor é uma raiz de uma sub-heap
- Fase 2: ordenação (decrecente)
  - ▶ Remover o máximo repetidamente
    - ★ Troca-se o último elemento pela raiz
    - ★ Diminui-se o tamanho da fila
    - ★ fixDown da raiz

# Algoritmos de Ordenação Eficientes - Heap Sort - versão 2

- Fase 1: construção da heap
  - ▶ Inicializa da metade do vetor
  - ▶  $N/2$ : pai dos nós folhas
    - ★ Pular sub-heaps de tamanho 1
  - ▶ Termina na posição 1
  - ▶ Resultado (contra intuitivo):
    - ★ Primeiro elemento sendo o maior elemento do vetor
    - ★ Outros maiores elementos, próximos ao início
  - ▶ Custo proporcional a  $2 * N$  (prove o custo - soma PG):

```
1 for(int k=N/2; k>=1; k--) {  
2     fixDown(k, N);  
3 }
```

# Algoritmos de Ordenação Eficientes - Heap Sort - versão 2

- Fase 2: ordenação (decrecente)
  - ▶ Remover o máximo repetidamente
    - ★ Troca-se o último elemento pela raiz
    - ★ Diminui-se o tamanho da fila
    - ★ fixDown da raiz

```
1 while (N>1) {  
2     exch(pq[1], pq[N]);  
3     fixDown(1, --N);  
4 }
```

# Algoritmos de Ordenação Eficientes - Heap Sort

```
1 void heap_sort(Item *v, int l, int r) {
2     pq = v+l-1; //fila de prioridades construída
   em v
3         //uma posição anterior a v[l]
4         //se l=0 -> pq[1] = v[0]
5     N = r-l+1;
6     for(int k=N/2; k>=1; k--)
7     {
8         fixDown(k, N);
9     }
10    while(N>1)
11    {
12        exch(pq[1], pq[N]);
13        fixDown(1, --N);
14    }
15 }
```

# Algoritmos de Ordenação Eficientes - Heap Sort

- A segunda fase é a mais custosa
  - ▶ Reorganizar o heap a cada remoção
  - ▶ Porém a estrutura da heap (pseudo-ordenada) contribui na tarefa de encontrar o maior elemento
  - ▶ Complexidade: cerca de  $2 * N * \log N + 2 * N$  comparações
    - ★  $2 * N$  na construção da heap
    - ★  $2 * N * \log N$  no conserto da heap (segunda fase)
    - ★  $O(n * \log n)$
    - ★ Façam a recorrência que prove os custos
- In-place: sim
- Estabilidade: não é estável
- Adaptatividade: ordenação x desempenho
  - ▶ Construção da heap: menos chamadas do fixDown
  - ▶ Ordenação final: não contribui no fixDown dos elementos folhas

# Roteiro

1 Fila de Prioridades

2 Heap Sort

3 Intro Sort

# Algoritmos de Ordenação Eficientes - Intro Sort

- É uma importante combinação de algoritmos de ordenação interna, utilizado no C++, C#
  - ▶ Java é quicksort three-way
- Híbrido:
  - ▶ quick + merge(mais espaço) + insertion
  - ▶ quick + heap(maior constante) + insertion
- Solução para utilizar as eficiências e evitar as deficiências de cada método
  - ▶ insertion: pequenos vetores, quase ordenados
  - ▶ quick: bom desempenho na maioria dos casos
  - ▶ quando a profundidade da recursividade atinge um máximo estipulado, aterna-se para outro método de ordenação
- Complexidade no pior caso:  $O(n * \log n)$
- In-place
  - ▶ Merge: espaço extra, proporcional a  $N$
  - ▶ Heap e Quick: sim
- Estabilidade: não estáveis
- Adaptatividade: não aproveita o estado do vetor



# Algoritmos de Ordenação Eficientes - Intro Sort

```
1 void intro(int *v, int l, int r, int maxdepth) {
2     if(r-l<=15){
3         //insertion_sort(v, l, r);
4         return;
5
6     } else if(maxdepth == 0) {
7         //merge_sort(v, l, r);
8         heap_sort(v, l, r);
9
10    } else {
11        compexch(v[l], v[(l+r)/2]);
12        compexch(v[l], v[r]);
13        compexch(v[r], v[(l+r)/2]);
14
15        int p = partition(v, l, r);
16        intro(v, l, p-1, maxdepth-1);
17        intro(v, p+1, r, maxdepth-1);
18    }
19 }
```

# Algoritmos de Ordenação Eficientes - Intro Sort

```
1 void intro_sort(int *v, int l, int r)
2 {
3     //duas vezes a altura da árvore
4     int maxdepth = 2*((int)log2((double)(r-l+1)))
5     ;
6     intro(v, l, r, maxdepth);
7     insertion_sort(v, l, r);
8 }
```