

# Filas de prioridade

## Conceitos e implementação

Prof. Marcelo de Souza

45EST – Algoritmos e Estruturas de Dados  
Universidade do Estado de Santa Catarina



## Leitura principal:

- ▶ Capítulo 9 de [Goodrich et al. \(2014\)](#)<sup>1</sup> – Filas de prioridade.

## Leitura complementar:

- ▶ Capítulo 6 de [Szwarcfiter e Markenzon \(2009\)](#)<sup>2</sup> – Listas de prioridades.
- ▶ Capítulo 4 de [Lafore e Machado \(2004\)](#)<sup>3</sup> – Pilhas e filas.

---

<sup>1</sup>Michael T Goodrich et al. (2014). *Data structures and algorithms in Java*. 6ª ed. John Wiley & Sons.

<sup>2</sup>Jayme Luiz Szwarcfiter e Lilian Markenzon (2009). *Estruturas de Dados e seus Algoritmos*. Vol. 2. Livros Técnicos e Científicos.

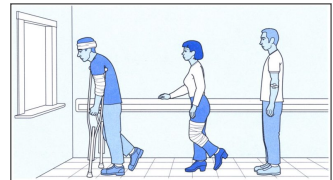
<sup>3</sup>Robert Lafore e Eveline Vieira Machado (2004). *Estruturas de dados & Algoritmos em Java*. Ciência Moderna.

# Filas de prioridade

## Ideia geral

**Fila de prioridade:** cada elemento tem uma prioridade, que determina a ordem de remoção.

- ▶ O elemento prioritário é o próximo a ser removido.
- ▶ **Atendimento médico:** a gravidade do paciente define sua prioridade de atendimento.



# Filas de prioridade

## Ideia geral

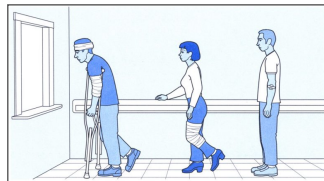


**Fila de prioridade:** cada elemento tem uma prioridade, que determina a ordem de remoção.

- ▶ O elemento prioritário é o próximo a ser removido.
- ▶ **Atendimento médico:** a gravidade do paciente define sua prioridade de atendimento.

### Operações:

- ▶ `insert`: insere um elemento na fila com sua prioridade.
- ▶ `min`: retorna o elemento prioritário da fila.
- ▶ `removeMin`: remove (e retorna) o elemento prioritário da fila.



# Filas de prioridade

## Ideia geral

**Fila de prioridade:** cada elemento tem uma prioridade, que determina a ordem de remoção.

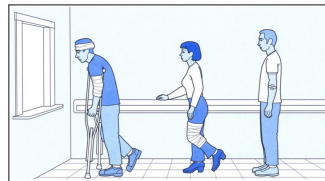
- ▶ O elemento prioritário é o próximo a ser removido.
- ▶ **Atendimento médico:** a gravidade do paciente define sua prioridade de atendimento.

## Operações:

- ▶ `insert`: insere um elemento na fila com sua prioridade.
- ▶ `min`: retorna o elemento prioritário da fila.
- ▶ `removeMin`: remove (e retorna) o elemento prioritário da fila.

## Aplicações:

- ▶ Busca em grafos (e.g. algoritmo de Dijkstra).
- ▶ Otimização combinatória (e.g. bin packing).
- ▶ Inteligência artificial (e.g. algoritmo A\*).



# Filas de prioridade

## Ideia geral



### Funcionamento

- ▶ Junto com o elemento, precisamos armazenar sua prioridade.
- ▶ Portanto, a fila armazenará uma **entrada** contendo uma **chave** e um **valor**.
  - ▶ Chave: prioridade do elemento.
  - ▶ Valor: elemento armazenado.
- ▶ O elemento de menor chave possui prioridade.



# Filas de prioridade

## Ideia geral

### Funcionamento

- ▶ Junto com o elemento, precisamos armazenar sua prioridade.
- ▶ Portanto, a fila armazenará uma **entrada** contendo uma **chave** e um **valor**.
  - ▶ Chave: prioridade do elemento.
  - ▶ Valor: elemento armazenado.
- ▶ O elemento de menor chave possui prioridade.

### Detalhes

- ▶ Várias entradas com mesma chave (mesma prioridade): escolhe aleatoriamente.
- ▶ A chave não precisa ser numérica (i.e., pode ser um tipo estruturado).
  - ▶ Fila de um banco: a prioridade é definida por vários atributos (idoso, gestante, cliente preferencial, tempo de chegada, ...).



# Filas de prioridade

## Exemplo de funcionamento

Seja uma fila de prioridade implementada usando uma lista sequencial, inicialmente vazia.

Método	Retorno	Conteúdo (não ordenado)	Conteúdo (ordenado)
insert(5, A)	—	{ (5, A) }	{ (5, A) }
insert(9, C)	—	{ (5, A), (9, C) }	{ (9, C), (5, A) }
insert(3, B)	—	{ (5, A), (9, C), (3, B) }	{ (9, C), (5, A), (3, B) }
min()	(3, B)	{ (5, A), (9, C), (3, B) }	{ (9, C), (5, A), (3, B) }
removeMin()	(3, B)	{ (5, A), (9, C) }	{ (9, C), (5, A) }
insert(7, D)	—	{ (5, A), (9, C), (7, D) }	{ (9, C), (7, D), (5, A) }
removeMin()	(5, A)	{ (9, C), (7, D) }	{ (9, C), (7, D) }
removeMin()	(7, D)	{ (9, C) }	{ (9, C) }
removeMin()	(9, C)	{ }	{ }
removeMin()	null	{ }	{ }
isEmpty()	true	{ }	{ }





# Filas de prioridade

Análise de complexidade (diferentes implementações)

Podemos implementar uma fila de prioridade usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.



# Filas de prioridade

Análise de complexidade (diferentes implementações)

Podemos implementar uma fila de prioridade usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.

- Qual a melhor escolha? ... vamos **analisar primeiro**, e escolher uma opção!



# Filas de prioridade

Análise de complexidade (diferentes implementações)

Podemos implementar uma fila de prioridade usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.

- Qual a melhor escolha? ... vamos **analisar primeiro**, e escolher uma opção!

Operação	Não ordenado		Ordenado	
	Arranjo	Lista encadeada	Arranjo	Lista encadeada
insert	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
min	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
removeMin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

— não estamos considerando o tempo gasto com *resize* no arranjo.



# Filas de prioridade

Análise de complexidade (diferentes implementações)

Podemos implementar uma fila de prioridade usando **arranjos** ou **encadeamento**. E ainda podemos manter a estrutura **não ordenada** ou **ordenada**.

- Qual a melhor escolha? ... vamos **analisar primeiro**, e escolher uma opção!

Operação	Não ordenado		Ordenado	
	Arranjo	Lista encadeada	Arranjo	Lista encadeada
insert	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
min	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
removeMin	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$

— não estamos considerando o tempo gasto com *resize* no arranjo.

**Conclusão:** usaremos uma implementação usando **arranjos ordenados**.

- Há uma estrutura mais eficiente chamada *heap*, que estudaremos mais adiante.



# Filas de prioridade

## Implementação das entradas

Classe Entry:

```
1  public class Entry<K,V> {  
2      private K k;  
3      private V v;  
4  
5      public Entry(K key, V value) {  
6          k = key;  
7          v = value;  
8      }  
9  
10     public K getKey() { return k; }  
11     public V getValue() { return v; }  
12  
13     public String toString() {  
14         return "[" + k + "; " + v + "];"  
15     }  
16 }
```



# Filas de prioridade

## Implementação das entradas

### Classe Entry:

```
1  public class Entry<K,V> {
2      private K k;
3      private V v;
4
5      public Entry(K key, V value) {
6          k = key;
7          v = value;
8      }
9
10     public K getKey() { return k; }
11     public V getValue() { return v; }
12
13     public String toString() {
14         return "[" + k + "; " + v + "];"
15     }
16 }
```

- ▶ A entrada possui tipos genéricos para a chave (K) e o valor (V).
- ▶ É fornecido somente um método construtor parametrizado e somente métodos get. Ou seja, não é possível criar uma entrada vazia, nem substituir sua chave ou seu valor.
- ▶ A classe Entry não é aninhada na fila de prioridade, uma vez que desejamos ter acesso à estrutura de uma entrada fora da estrutura de dados.



# Filas de prioridade

## Implementação (interface)

Interface PriorityQueue:

```
1 public interface PriorityQueue<K, V> {  
2     int size();  
3     boolean isEmpty();  
4     Entry<K, V> insert(K key, V value);  
5     Entry<K, V> min();  
6     Entry<K, V> removeMin();  
7 }
```



# Filas de prioridade

## Implementação (interface)

### Interface PriorityQueue:

```
1 public interface PriorityQueue<K, V> {  
2     int size();  
3     boolean isEmpty();  
4     Entry<K, V> insert(K key, V value);  
5     Entry<K, V> min();  
6     Entry<K, V> removeMin();  
7 }
```

- ▶ A fila de prioridade é uma coleção de entradas, que armazenam a chave e o valor (com tipos K e V, respectivamente).
- ▶ As operações sempre retornam a entrada sendo consultada/alterada.





# Filas de prioridade

## Comparação de chaves

Quando usamos uma **chave numérica** (e.g., um valor inteiro) para definir a prioridade, a comparação de chaves é natural e já fornecida pelo tipo de dados usado (`Integer`).

- ▶ O elemento prioritário é o que possui **menor valor** de chave.



# Filas de prioridade

## Comparação de chaves

Quando usamos uma **chave numérica** (e.g., um valor inteiro) para definir a prioridade, a comparação de chaves é natural e já fornecida pelo tipo de dados usado (`Integer`).

- ▶ O elemento prioritário é o que possui **menor valor** de chave.

Quando usamos uma **chave estruturada** (e.g., uma classe e seus atributos), precisamos definir uma forma de comparar duas chaves.

- ▶ Os clientes do banco têm dois atributos: sua categoria ( $A$  ou  $B$ ), e seu horário de chegada. Clientes da categoria  $A$  são especiais, e têm prioridade no atendimento. Quando dois clientes são da mesma categoria, o que chegou primeiro tem prioridade.



# Filas de prioridade

## Comparação de chaves

Quando usamos uma **chave numérica** (e.g., um valor inteiro) para definir a prioridade, a comparação de chaves é natural e já fornecida pelo tipo de dados usado (`Integer`).

- ▶ O elemento prioritário é o que possui **menor valor** de chave.

Quando usamos uma **chave estruturada** (e.g., uma classe e seus atributos), precisamos definir uma forma de comparar duas chaves.

- ▶ Os clientes do banco têm dois atributos: sua categoria (*A* ou *B*), e seu horário de chegada. Clientes da categoria *A* são especiais, e têm prioridade no atendimento. Quando dois clientes são da mesma categoria, o que chegou primeiro tem prioridade.

## Como comparar?

- ▶ **Opção 1:** a classe da chave implementa `Comparable` (e o método `compareTo`) + um comparador genérico.
- ▶ **Opção 2:** implementar um comparador específico para a classe da chave.



# Filas de prioridade

## Comparação de chaves

**Opção 1:** a classe da chave implementa Comparable + um comparador genérico.

A chave implementa Comparable:

```
1  private class MyKey implements Comparable<MyKey> {  
2      char category; // 'A' or 'B'  
3      int value;  
4  
5      public int compareTo(MyKey o) {  
6          if(this.category == o.category) {  
7              if(this.value == o.value) return 0;  
8              else if(this.value < o.value) return -1;  
9              else return 1;  
10         } else if(this.category == 'A')  
11             return -1;  
12         else  
13             return 1;  
14     }  
15 }
```



# Filas de prioridade

## Comparação de chaves

**Opção 1:** a classe da chave implementa Comparable + um comparador genérico.

A chave implementa Comparable:

```
1 private class MyKey implements Comparable<MyKey> {  
2     char category; // 'A' or 'B'  
3     int value;  
4  
5     public int compareTo(MyKey o) {  
6         if(this.category == o.category) {  
7             if(this.value == o.value) return 0;  
8             else if(this.value < o.value) return -1;  
9             else return 1;  
10        } else if(this.category == 'A')  
11            return -1;  
12        else  
13            return 1;  
14    }  
15 }
```

- ▶ A classe MyKey implementa a chave para priorizar o atendimento bancário.
- ▶ O método compareTo usa a categoria (category) e o tempo de chegada (value) para comparar o objeto atual (this) com outro objeto (o).
- ▶ Se ambos têm a mesma prioridade, o método retorna 0; se o objeto atual tem prioridade, retorna -1; se o outro objeto tem prioridade, retorna 1.



# Filas de prioridade

## Comparação de chaves

**Opção 1:** a classe da chave implementa Comparable + um comparador genérico.

A classe DefaultComparator implementa o comparador genérico:

```
1 public class DefaultComparator<E> implements Comparator<E> {  
2     public int compare(E a, E b) {  
3         return ((Comparable<E>) a).compareTo(b);  
4     }  
5 }
```



# Filas de prioridade

## Comparação de chaves

**Opção 1:** a classe da chave implementa Comparable + um comparador genérico.

A classe DefaultComparator implementa o comparador genérico:

```
1 public class DefaultComparator<E> implements Comparator<E> {  
2     public int compare(E a, E b) {  
3         return ((Comparable<E>) a).compareTo(b);  
4     }  
5 }
```

- ▶ O comparador assume que o tipo genérico E implementa Comparable e, consequentemente, fornece o método compareTo (linha 3).
- ▶ Sua tarefa é invocar esse método, comparando dois objetos recebidos por parâmetro e retornando o resultado da comparação.
- ▶ Note que a ordem dos parâmetros altera o resultado: -1 indica que a tem prioridade, enquanto 1 indica que b tem prioridade.



# Filas de prioridade

## Comparação de chaves

**Opção 2:** implementar um comparador específico para a classe da chave.

Comparador específico para MyKey:

```
1  private class MyKeyComparator implements Comparator<MyKey> {
2      public int compare(MyKey a, MyKey b) {
3          if(a.category == b.category) {
4              if(a.value == b.value)
5                  return 0;
6              else if(a.value < b.value)
7                  return -1;
8              else
9                  return 1;
10         } else if(a.category == 'A')
11             return -1;
12         else
13             return 1;
14     }
15 }
```





# Filas de prioridade

## Comparação de chaves

**Opção 2:** implementar um comparador específico para a classe da chave.

Comparador específico para MyKey:

```
1  private class MyKeyComparator implements Comparator<MyKey> {
2      public int compare(MyKey a, MyKey b) {
3          if(a.category == b.category) {
4              if(a.value == b.value)
5                  return 0;
6              else if(a.value < b.value)
7                  return -1;
8              else
9                  return 1;
10         } else if(a.category == 'A')
11             return -1;
12         else
13             return 1;
14     }
15 }
```

- ▶ Neste caso, MyKey não precisa implementar a interface Comparable (nem o método compareTo), pois a comparação fica a cargo da classe MyKeyComparator.
- ▶ O método compare implementa a comparação de dois objetos da classe MyKey.



# Filas de prioridade

Implementação usando um arranjo ordenado

```
1  public class ArrayPriorityQueue<K,V> implements PriorityQueue<K, V> {
2      private static class PQEntry<K,V> implements Entry<K,V> {
3          private K k;
4          private V v;
5
6          public PQEntry(K key, V value) { k = key; v = value; }
7
8          public K getKey() { return k; }
9          public V getValue() { return v; }
10         public String toString() { return "[" + k + "; " + v + "]"; }
11     }
12
13     private List<Entry<K,V>> list = new ArrayList<>();
14     private Comparator<K> comp;
15
16     //...
17 }
```

- ▶ A classe define modela suas entradas (classe aninhada PQEntry), usa um ArrayList como estrutura fundamental (linha 13) e define um comparador (linha 14).



# Filas de prioridade

Implementação usando um arranjo ordenado

Definição e uso do comparador:

```
1 public ArrayPriorityQueue() {  
2     this(new DefaultComparator<K>());  
3 }  
4  
5 public ArrayPriorityQueue(Comparator<K> c) {  
6     comp = c;  
7 }  
8  
9 private int compare(Entry<K,V> a, Entry<K,V> b) {  
10     return comp.compare(a.getKey(), b.getKey());  
11 }
```



# Filas de prioridade

Implementação usando um arranjo ordenado

Definição e uso do comparador:

```
1 public ArrayPriorityQueue() {  
2     this(new DefaultComparator<K>());  
3 }  
4  
5 public ArrayPriorityQueue(Comparator<K> c) {  
6     comp = c;  
7 }  
8  
9 private int compare(Entry<K,V> a, Entry<K,V> b) {  
10     return comp.compare(a.getKey(), b.getKey());  
11 }
```

- ▶ O comparador específico da chave usada pode ser informada no construtor parametrizado (linha 5). Caso ele não seja informado, é usado o comparador genérico (linha 2), assumindo que a chave implementa `Comparable`.
- ▶ Na comparação (método `compare`), duas entradas são comparadas invocando o comparador e informando suas chaves.



# Filas de prioridade

Implementação usando um arranjo ordenado

Método insert:

```
1  public Entry<K, V> insert(K key, V value) {
2      Entry<K,V> newest = new PQEntry<>(key, value);
3      if(isEmpty())
4          list.add(0, newest);
5      else {
6          int j = 0;
7          while(j < size() && compare(newest, list.get(j)) < 0)
8              j++;
9          list.add(j, newest);
10     }
11     return newest;
12 }
```



# Filas de prioridade

Implementação usando um arranjo ordenado

Método insert:

```
1 public Entry<K, V> insert(K key, V value) {
2     Entry<K,V> newest = new PQEntry<>(key, value);
3     if(isEmpty())
4         list.add(0, newest);
5     else {
6         int j = 0;
7         while(j < size() && compare(newest, list.get(j)) < 0)
8             j++;
9         list.add(j, newest);
10    }
11    return newest;
12 }
```

- ▶ A nova entrada é criada com a chave e valor recebidos (linha 2).
- ▶ Caso a fila esteja vazia, a entrada é inserida na primeira posição (linhas 3 e 4).
- ▶ Caso contrário, as linhas 6 a 9 buscam a posição e inserem a entrada mantendo a ordenação.



# Filas de prioridade

Implementação usando um arranjo ordenado

Método min:

```
1 public Entry<K,V> min() {  
2     if (isEmpty()) return null;  
3     return list.get(size() - 1);  
4 }
```

Método removeMin:

```
1 public Entry<K,V> removeMin() {  
2     if (isEmpty()) return null;  
3     return list.remove(size() - 1);  
4 }
```

