Filas de prioridade

Conceitos e implementação

Prof. Marcelo de Souza

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

UDESC

Material de apoio

Leitura principal:

- Capítulo 9 de Goodrich et al. (2014)[1] – Filas de prioridade.

Leitura complementar:

- Capítulo 6 de Szwarcfiter e Markenzon (2009)[2] – Listas de prioridades.
- Capítulo 4 de Lafore e Machado (2004)[3] – Pilhas e filas.

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

[2] Jayme Luiz Szwarcfiter e Lilian Markenzon (2009). *Estruturas de Dados e seus Algoritmos*. Vol. 2. Livros Técnicos e Científicos.

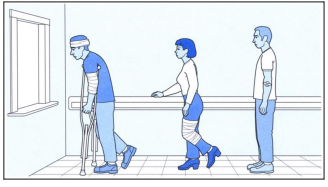
[3] Robert Lafore e Eveline Vieira Machado (2004). *Estruturas de dados & Algoritmos em Java*. Ciência Moderna.

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

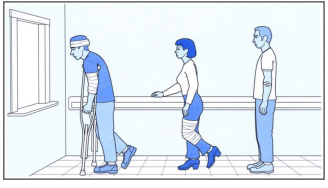
Ideia geral

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.

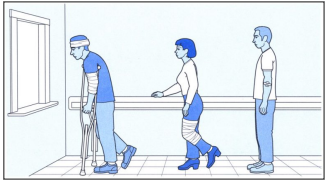
- O elemento prioritário é o próximo a ser removido.
- **Atendimento médico:** a gravidade do paciente define sua prioridade de atendimento.

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 (entradas e tipo abstrato de dados)

Interface `Entry`:

```
1 public interface Entry<K, V> {
2   K getKey();
3   V getValue();
4 }
```

Filas de prioridade

Implementação (entradas e tipo abstrato de dados)

Interface `Entry`:

```
1 public interface Entry<K, V> {
2   K getKey();
3   V getValue();
4 }
```

- A entrada possui tipos genéricos de chave (`K`) e valor (`V`).

Filas de prioridade

Implementação (entradas e tipo abstrato de dados)

Interface `Entry`:

```
1 public interface Entry<K, V> {
2   K getKey();
3   V getValue();
4 }
```

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 entrada possui tipos genéricos de chave (`K`) e valor (`V`).

Filas de prioridade

Implementação (entradas e tipo abstrato de dados)

Interface `Entry`:

```
1 public interface Entry<K, V> {
2   K getKey();
3   V getValue();
4 }
```

- A entrada possui tipos genéricos de chave (`K`) e valor (`V`).

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 arranjos

```
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).

45EST – Algoritmos e Estruturas de Dados
Prof. Marcelo de Souza