45EST – Algoritmos e Estruturas de Dados

Filas de prioridade

Prof. Marcelo de Souza

UDESC Ibirama
Bacharelado em Engenharia de Software

marcelo.desouza@udesc.br
Versão compilada em 1 de setembro de 2018

Leitura obrigatória:

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

Leitura complementar:

- Capítulo 6 de Szwarcfiter and Markenzon [2009] – Listas de prioridades.
- Capítulo 11 de Preiss [2001] – Heaps e filas de prioridade.

Filas de prioridade

Ideia geral:

- Cada elemento da fila tem um valor prioridade.
- A remoção é feita pela ordem de prioridade.
- Exemplos:
  - Controle de tráfego aéreo.
  - Fila de um banco com clientes preferenciais.

Funcionamento:

- A fila armazena uma **entrada** com dois campos: chave e valor.
- Chave: prioridade do elemento.
- Valor: elemento armazenado.

- Elemento de menor chave possui prioridade.
- Métodos:
  - `insert(k, v)`, `min()`, `removeMin()`.

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min()	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin()	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin()	(5,A)	{ (7,D), (9,C) }
removeMin()	(7,D)	{ (9,C) }
removeMin()	(9,C)	{ }
removeMin()	null	{ }
isEmpty()	true	{ }

Detalhes:

- Várias entradas com mesma chave → escolhe aleatoriamente.
- Chave não precisa ser numérica (ex: pode ser um tipo estruturado).

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 k, V v);
```

```
    Entry<K,V> min();
    Entry<K,V> removeMin();
}
```

Comentários:

- Uma fila de prioridades armazena uma coleção de entradas e fornece métodos de acesso a elas.
- Para isso, usa internamente alguma estrutura de dados básica.

Comparação de chaves:

- As chaves precisam ser comparáveis, formando uma ordem de elementos.
- Opção 1:
  1. Definir a classe da chave como comparável (`Comparable`) e implementar o método `compareTo`.
     - Muitas classes do Java já são comparáveis, como `Integer`.
  2. Utilizar uma classe `Comparator` genérica para qualquer chave comparável.

Classe comparadora genérica (`DefaultComparator`):

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

- Opção 2:
  1. Criar uma classe `Comparator` específica para a chave usada.

Exemplo de comparador para tamanho de String:

```
1  public class StringLengthComparator implements Comparator<String> {
2     public int compare(String a, String b) {
3        if(a.length() < b.length())
4           return -1;
5        else if(a.length() == b.length())
6           return 0;
7        else
8           return 1;
9     }
10 }
```

Comentários:

- Ao comparar dois elementos a e b:
  - Retorno -1 se $a < b$.
  - Retorno 1 se $b < a$.
  - Retorno 0 se $b = a$.

Implementação de uma fila de prioridade *não ordenada*:

```
1  public class UnsortedPriorityQueue<K, V>
2     implements PriorityQueue<K, V> {
3
4     protected static class PQEntry<K,V> implements Entry<K,V> {
5        private K k;
6        private V v;
7
8        public PQEntry(K key, V value) {
9           k = key;
10          v = value;
11       }
12
13       public K getKey() { return k; }
14       public V getValue() { return v; }
```

```
15          protected void setKey(K key) { k = key; }
16          protected void setValue(V value) { v = value; }
17      }
18
19      private PositionalList<Entry<K,V>> list =
20          new LinkedPositionalList<>();
21      private Comparator<K> comp;
22
23      public UnsortedPriorityQueue() {
24          this(new DefaultComparator<K>());
25      }
26
27      public UnsortedPriorityQueue(Comparator<K> c) {
28          comp = c;
29      }
30
31      protected int compare(Entry<K,V> a, Entry<K,V> b) {
32          return comp.compare(a.getKey(), b.getKey());
33      }
34
35      protected boolean checkKey(K k) {
36          try {
37              return (comp.compare(k, k)) == 0;
38          } catch(ClassCastException e) {
39              throw new IllegalArgumentException("Incompatible key");
40          }
41      }
42
43      public boolean isEmpty() {
44          return size() == 0;
45      }
46
47      private Position<Entry<K, V>> findMin() {
48          Position<Entry<K,V>> small = list.first();
49          Position<Entry<K,V>> walk = small;
50          while(walk != null) {
51              if(compare(walk.getElement(), small.getElement()) < 0)
52                  small = walk;
```

```
53            walk = list.after(walk);
54         }
55         return small;
56      }
57
58      public Entry<K, V> insert(K k, V v) {
59
60         checkKey(key);
61         Entry<K,V> newest = new PQEntry<>(key, value);
62         list.addLast(newest);
63         return newest;
64      }
65
66      public Entry<K, V> min() {
67         if(list.isEmpty()) return null;
68         return findMin().getElement();
69      }
70
71      public Entry<K, V> removeMin() {
72         if(list.isEmpty()) return null;
73         return list.remove(findMin());
74      }
75
76      public int size() { return list.size(); }
77
78      public String toString() {
79         StringBuilder sb = new StringBuilder("(");
80         Position<Entry<K,V>> walk = list.first();
81         while(walk != null) {
82            sb.append(walk.getElement().getValue() + " [" +
               ↪  walk.getElement().getKey() + "]");
83            walk = list.after(walk);
84            if (walk != null)
85            sb.append(", ");
86         }
87         sb.append(")");
88         return sb.toString();
89      }
```

```
90  }
```

Comentários:

- A classe `PQEntry` modela entradas `<chave, valor>`.
- As entradas são armazenadas em uma lista posicional.
- A fila possui um comparador `comp`. Ele é recebido como argumento na construção da fila, ou é atribuído o construtor *default* predefinido.
- O método `checkKey` verifica se a chave é comparável.
- O método `findMin` busca elemento mínimo, a ser retornado/removido nos métodos `min` e `removeMin`.

Implementação de uma fila de prioridade *ordenada*:

```
1  public class SortedPriorityQueue<K,V>
2     implements PriorityQueue<K, V> {
3
4     protected static class PQEntry<K,V> implements Entry<K,V> {
5        private K k;
6        private V v;
7
8        public PQEntry(K key, V value) {
9           k = key;
10          v = value;
11       }
12
13       public K getKey() { return k; }
14       public V getValue() { return v; }
15       protected void setKey(K key) { k = key; }
16       protected void setValue(V value) { v = value; }
17    }
18
19    private PositionalList<Entry<K,V>> list =
20       new LinkedPositionalList<>();
```

```
21      private Comparator<K> comp;
22
23
24      public SortedPriorityQueue() {
25         this(new DefaultComparator<K>());
26      }
27
28      public SortedPriorityQueue(Comparator<K> c) {
29         comp = c;
30      }
31
32      protected int compare(Entry<K,V> a, Entry<K,V> b) {
33         return comp.compare(a.getKey(), b.getKey());
34      }
35
36      protected boolean checkKey(K k) {
37         try {
38            return (comp.compare(k, k)) == 0;
39         } catch(ClassCastException e) {
40            throw new IllegalArgumentException("Incompatible key");
41         }
42      }
43
44      public boolean isEmpty() {
45         return size() == 0;
46      }
47
48      public Entry<K,V> insert(K key, V value) {
49
50         checkKey(key);
51         Entry<K,V> newest = new PQEntry<>(key, value);
52         Position<Entry<K,V>> walk = list.last();
53
54         while (walk != null && compare(newest, walk.getElement()) < 0)
55            walk = list.before(walk);
56         if (walk == null)
57            list.addFirst(newest);
58         else
```

```
59            list.addAfter(walk, newest);
60         return newest;
61      }
62
63      public Entry<K,V> min() {
64         if (list.isEmpty()) return null;
65         return list.first().getElement();
66      }
67
68      public Entry<K,V> removeMin() {
69         if (list.isEmpty()) return null;
70         return list.remove(list.first());
71      }
72
73      public int size() { return list.size(); }
74
75      public String toString() {
76         StringBuilder sb = new StringBuilder("(");
77         Position<Entry<K,V>> walk = list.first();
78         while(walk != null) {
79            sb.append(walk.getElement().getValue() + " [" +
               ↪  walk.getElement().getKey() + "]");
80            walk = list.after(walk);
81            if (walk != null)
82            sb.append(", ");
83         }
84         sb.append(")");
85         return sb.toString();
86      }
87   }
```

Comentários:

- Diferente da fila não ordenada, o método `insert` adiciona a entrada de forma ordenada, do menor ao maior valor de chave.
- O elemento mínimo é o armazenado na primeira posição da lista.

Comparação de performance:

Método	`UnsortedPriorityQueue`	`SortedPriorityQueue`
`size`	$O(1)$	$O(1)$
`isEmpty`	$O(1)$	$O(1)$
`insert`	$O(1)$	$O(n)$
`min`	$O(n)$	$O(1)$
`removeMin`	$O(n)$	$O(1)$

Atividades

1. Implemente uma fila de prioridade para armazenar os passageiros de uma companhia aérea. Clientes do plano *premium* possuem prioridade sobre clientes do plano *basic*. Além disso, clientes *prioritários* (idosos, gestantes, etc.) possuem ainda mais prioridade que clientes *premium*. Crie um sistema para chamar os clientes para embarque pelo seu nome considerando a prioridade de cada um. Como critério de desempate dentro de um mesmo tipo, considere a idade do cliente. Para modelar a prioridade (chave) utilize uma classe. Implemente usando filas ordenadas e não ordenadas.

2. Resolva os seguintes exercícios de Goodrich et al. [2014]:

R-9.3: O que cada uma das chamadas `removeMin` retorna, dentre a seguinte sequências de operações em uma fila de prioridade: `insert(5,A)`, `insert(4,B)`, `insert(7,F)`, `insert(1,D)`, `removeMin()`, `insert(3,J)`, `insert(6,L)`, `removeMin()`, `removeMin()`, `insert(8,G)`, `removeMin()`, `insert(2,H)`, `removeMin()`, `removeMin()`.

R-9.4: Um aeroporto está desenvolvendo uma simulação computacional de controle de tráfego aéreo para lidar com eventos como aterrissagens e decolagens. Cada evento tem um *timestamp* que simboliza o tempo em que o evento ocorrerá. A simulação necessita realizar eficientemente duas operações fundamentais:

- Inserir um evento com um *timestamp* (isto é, adicionar um evento futuro).
- Retornar o evento com o *timestamp* mais próximo (isto é, determinar o próximo evento para processar).
- Qual estrutura de dados deverá ser usada para realizar essas operações? Por quê?

R-9.5: O método `min` da classe `UnsortedPriorityQueue` executa em tempo $O(n)$. Faça uma pequena modificação na classe para que o método `min` execute em tempo $O(1)$. Explique qualquer modificação necessária em outros métodos da classe.

R-9.6: Você pode adaptar sua solução do exercício anterior para fazer o método `removeMin` da classe `UnsortedPriorityQueue` executar em tempo $O(1)$? Justifique sua resposta.

R-9.12: Considere uma situação onde um usuário possui chaves numéricas e deseja ter uma fila de prioridade *orientada para o máximo*. Como pode uma fila de prioridade padrão (orientada ao mínimo) ser usada para tal propósito?

C-9.25: Mostre como implementar uma pilha usando apenas uma fila de prioridade e uma variável inteira adicional.

C-9.26: Mostre como implementar uma fila FIFO usando apenas uma fila de prioridade e uma variável inteira adicional.

C-9.27: O Professor Idle sugere a seguinte solução para o exercício anterior. Sempre que uma entrada é inserida na fila, é dada uma chave que

é igual ao tamanho atual da fila. Esta estratégia resulta em uma semântica FIFO? Prove que é verdadeiro ou dê um contra exemplo.

C-9.28: Reimplemente a classe `SortedPriorityQueue` usando um array. Certifique-se de manter o desempenho do `removeMin` em $O(1)$.

Referências

Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2014). *Data structures and algorithms in Java*. John Wiley & Sons, 6th edition.

Preiss, B. R. (2001). *Estruturas de dados e algoritmos: padrões de projetos orientados a objetos com Java*. Campus.

Szwarcfiter, J. L. and Markenzon, L. (2009). *Estruturas de Dados e seus Algoritmos*, volume 2. Livros Técnicos e Científicos.