Capítulo 8

Fila de prioridade e heap

Um sistema de mensageria é implementado. Esse sistema gerencia a comunicação entre diversos sistemas, criando uma certa integração entre eles, por exemplo, autoatendimento e mainframe, em instituições financeiras. Contudo, com o aumento do volume de mensagens, informações importantes são perdidas ao longo do processo. Esse é o típico caso para o uso de uma fila de prioridade.

Em linhas gerais, uma fila de prioridade tem o comportamento parecido com o de uma fila tradicional, a FIFO (*first-in first-out*, em inglês). Contudo, a fila tradicional, assim como os demais tipos de estrutura de dados, tem uma sequência arbitrária de entrada e saída, geralmente organizadas de maneira linear entre seus elementos armazenados (NECAISE, 2010; GOODRICH; TAMASSIA, 2013).

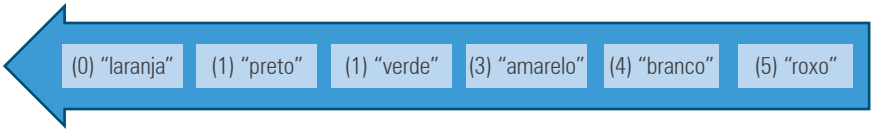
Diferentemente das estruturas tradicionais (apesar da entrada de uma fila de prioridade ser arbitrária), sua saída é determinada pela prioridade do elemento. Quanto maior a prioridade dele, mais rápido ele deve ser removido da fila (CORMEN *et al*., 2009; GOODRICH; TAMASSIA, 2013).

Sendo assim, pode-se entender que a fila de prioridade é uma extensão da fila tradicional, na qual é atribuído um valor de prioridade para o elemento armazenado. Um elemento, ou classe, deve ser criado para que seja atribuída a prioridade de cada item da fila. Ele deve armazenar o objeto do tipo da fila e um valor inteiro que represente a sua prioridade. A prioridade mais alta de uma fila é dada pelo seu menor inteiro, em outras palavras, o valor 0 é o elemento de maior prioridade dentro da fila. Ele deve ser removido primeiro. No caso da existência de diversos itens com a mesma prioridade, é importante ressaltar que os itens devem se comportar como uma fila tradicional. Isso vale apenas para os itens com o mesmo grau de prioridade (NECAISE, 2010; GOODRICH; TAMASSIA, 2013). Veja uma sequência de pseudocódigo a seguir:

```
1.     enfileirar("roxo", 5)
2.     enfileirar("preto", 1)
3.     enfileirar("laranja", 0)
4.     enfileirar("branco", 4)
5.     enfileirar("verde", 1)
6.     enfileirar("amarelo", 3)
```

Cada linha do código acima representa uma chamada para enfileirar uma cor dentro da fila de prioridade. Como segundo parâmetro do método *enfileirar*, é informado o valor da prioridade da cor dentro da fila. Com base na sequência apresentada, a figura 1 apresenta a posição dos itens na fila de prioridade de acordo com as regras dessa estrutura de dados.

Figura 1 – Fila de prioridade entre cores apresentando a sequência de saída de acordo com a prioridade de cada item



Perceba que indiferentemente do momento em que cada item foi adicionado na fila, ela manteve a ordem de prioridade começando do inteiro 0 e, no exemplo, indo até o 5. Veja que cada item está representado pela prioridade entre parênteses e o valor posicionado logo na sequência da linha.

Quando ocorreu a situação de existirem dois itens com a mesma prioridade na figura 1, respeitou-se a ordem de inserção na fila. Esse é o caso que ocorreu entre os itens "preto" e "verde" na fila de prioridade apresentada.

Entre as filas de prioridade podem existir dois tipos, as restritas (*bounded*, em inglês) e as sem restrições (*unbounded*). Uma fila de prioridade restrita determina o intervalo de inteiros que representará os valores de prioridade. Nesse caso, não pode ser determinado nenhuma prioridade fora desse intervalo. Já a fila de prioridade sem restrições não tem um intervalo de prioridades predeterminado. Sendo assim, a única regra referente às prioridades é que o menor inteiro tem prioridade mais alta que o maior inteiro (NECAISE, 2010).

Na definição de uma estrutura de dados do tipo fila de prioridade, devem ser considerados os métodos básicos para determinar as operações que podem ser realizadas por ela (FORBELLONE; EBERSPACHER, 2014; NECAISE, 2010; GOODRICH; TAMASSIA, 2013). Os métodos que devem ser considerados são:

- ***FilaPrioridade()*:** cria uma fila de prioridade sem restrições.

- ***FilaPrioridade*(limite):** cria uma fila de prioridade restrita, em que o intervalo é iniciado em 0 e vai até o limite informado como parâmetro.
- ***ehVazia* (*isEmpty*, em inglês):** informa um resultado booleano dizendo se a pilha possui algum elemento armazenado ou não.
- ***tamanho* (*length*, em inglês):** retorna à quantidade de elementos/itens armazenados no momento da consulta.
- ***enfileira* (item, prioridade) (*enqueue*, em inglês):** insere um elemento/item na fila de acordo com a prioridade informada. A prioridade deve estar dentro do intervalo limitante, nos casos de filas restritas.
- ***desenfileira* (*dequeue*, em inglês):** remove e retorna o elemento que está no início da fila de prioridade. Esse elemento possui o maior grau de prioridade. É importante também verificar se a fila não está vazia antes de realizar essa operação.

Com os métodos básicos definidos e apresentados, é hora de implementar a classe FilaPrioridade. A base para armazenar os dados é uma lista, com as operações determinadas pela regra de prioridade. Sendo assim, é possível realizar a implementação utilizando um vetor estático, um vetor dinâmico ou uma estrutura chamada *heap*. As devidas implementações serão apresentadas nos próximos tópicos.

1 Implementando a fila de prioridades com vetores

Em Java, existem dois tipos de vetores que auxiliam a implementação das filas de prioridades. O primeiro tipo utilizado é o vetor estático, ou vetor primitivo. Ele possui algumas limitações durante a implementação, o que, em alguns casos, pode ser algo determinante na decisão sobre seu uso. A segunda possibilidade de implementação com o uso

de vetores é pela API Collections do Java. A API fornece recursos de maneira a otimizar o trabalho na implementação da fila de prioridade, além de trabalhar com alocação dinâmica de memória, facilitando o controle da quantidade de itens armazenados no ciclo do sistema.

Para melhor entendimento dos métodos básicos utilizados na estrutura de uma fila de prioridade, a primeira demonstração de implementação será feita com vetores primitivos. A implementação será realizada com a fila de prioridade sem restrições, assim são evitadas algumas verificações de limites que podem gerar confusão na compreensão das operação realizadas pelo código.

Como são necessárias duas informações base para compor um item que será armazenado na fila de prioridade, o objeto ou valor e sua prioridade, é importante criar uma classe que o represente. Essa classe é implementada de forma genérica para que, no momento do uso, seja determinado o tipo do valor que será armazenado no item.

PARA SABER MAIS

Tipos genéricos são muito utilizados em diversos sistemas e implementações de algoritmos em que os dados são dinâmicos. Para mais informações, consulte: https://docs.oracle.com/javase/tutorial/java/generics/types.html (acesso em: 20 abr. 2020).

Por se tratar de uma classe que representa o objeto que modela os itens armazenados na fila de prioridade, sua implementação é simples. Ela apenas define os atributos; um construtor que requisita os dois atributos, tornando-os obrigatórios; os métodos *getters;* e um método para formatar a exibição do item como texto. A implementação da classe Item é apresentada a seguir utilizando como premissa a técnica para tipos genéricos do Java.

```
1.  public class Item<T> {
2.     private T valor;
3.     private int prioridade;
4.
5.     public Item(T valor, int prioridade) {
6.            this.valor = valor;
7.            this.prioridade = prioridade;
8.     }
9.
10.    public T getValor() {
11.           return valor;
12.    }
13.
14.    public int getPrioridade() {
15.           return prioridade;
16.    }
17.
18.    @Override
19.    public String toString() {
20.           return String.format("(%d - %s)",
21.                         this.prioridade,
22.                         this.valor.toString());
23.    }
24. }
```

Com a classe Item definida, é possível construir a classe FilaPrioridade para compreender o funcionamento da estrutura de dados. Ela é apresentada a seguir, e na sequência a explicação linha a linha sobre o código.

```
1.  public class FilaPrioridade {
2.     private Item<?>[] itens;
3.     private int tamanho;
4.     private int capacidade;
5.
6.     public FilaPrioridade(int capacidade) {
7.            this.tamanho = 0;
8.            this.capacidade = capacidade;
9.            this.itens = new Item[this.capacidade];
10.    }
11.
12.    public FilaPrioridade() {
13.           this(10);
14.    }
15.
16.    public boolean ehVazia() {
17.           return this.tamanho == 0;
18.    }
19.
20.    public int tamanho() {
21.           return this.tamanho;
22.    }
23.
24.    public void enfileira(Item<?> i) {
25.           if(this.capacidade == this.tamanho) {
26.                  throw new RuntimeException("A fila
está cheia.");
27.           }
28.
29.           this.itens[this.tamanho++] = i;
30.    }
31.
32.    public Item<?> desenfileira() {
33.           if(this.ehVazia()) {
34.                  throw new RuntimeException("A fila
está vazia.");
35.           }
36.
37.           int indice = this.buscaMaiorPrioridade();
38.           Item<?> altaPrioridade = this.itens[indice];
```

```
39.           for(int i = indice; i < this.tamanho() - 1;
i++) {
40.                   this.itens[i] = this.itens[i + 1];
41.           }
42.
43.           this.tamanho--;
44.           return altaPrioridade;
45.    }
46.
47.    private int buscaMaiorPrioridade() {
48.           int p = 0;
49.           for (int i = 0; i < this.tamanho(); i++) {
50.                  if (this.itens[p].getPrioridade() >
51.                         this.itens[i].getPrioridade())
{
52.                         p = i;
53.                  }
54.           }
55.           return p;
56.    }
57.
58.    @Override
59.    public String toString() {
60.           return Arrays.toString(this.itens);
61.    }
62. }
```

A classe é iniciada ao declarar os atributos necessários para a manipulação e armazenamento das informações. Na linha 2, é declarado um vetor de itens, seguido por dois inteiros, o *tamanho* (linha 3), que representa a quantidade de elementos armazenados dentro fila, e a *capacidade* (linha 4), representando a quantidade máxima comportada pela fila. Logo a seguir, os dois construtores são declarados (entre as linhas 6 e 14), com a diferença de aceitar a capacidade máxima igual a 10 (segundo construtor) ou determinar a capacidade máxima no momento da instância do objeto. Da linha 16 à linha 18, o método que informa se a fila está vazia ou não é apresentado.

A implementação do método que consulta o tamanho da lista no momento corrente é realizada também (linhas 20 a 22). Ambos os métodos de suporte auxiliando as validações básicas sobre a regra para uma fila de prioridade. O próximo método implementado é o *enfileira* (linhas 24 a 30), que recebe como parâmetro um Item. Existe a validação para identificar se a fila não atingiu sua capacidade máxima (linhas 25 a 27), e, na sequência, o item é adicionado ao final da fila (linha 29). O método *desenfileira* é apresentado na sequência entre as linhas 32 e 45. O método deve retornar o item que está saindo da fila no momento de sua chamada. O primeiro passo importante do método é validar que existe um item para sair, caso contrário, deve-se lançar uma exceção informando que a fila está vazia (linhas 33 a 35). Validado que existem itens na fila, deve-se buscar pelo item de maior prioridade (lembrando que a maior prioridade é o 0 ou o menor inteiro entre as prioridades na fila). A busca do item de maior prioridade é realizada através do método privado *buscaMaiorPrioridade*, implementado entre as linhas 47 e 56. Ele foi implementado separado por uma questão de organização do código e para facilitar a leitura. O método *buscaMaiorPrioridade* retorna a posição da primeira ocorrência de maior prioridade na fila. Essa posição é recebida pelo método *desenfileirar* (linha 37), que imediatamente separa o item correspondente (linha 38) para retornar ao final. Entre as linhas 39 e 41 são reposicionados os itens dentro do vetor, e depois o tamanho é decrescido em um ponto (linha 43). O último método implementado para auxiliar na exibição de lista como texto é o *toString* (linhas 58 a 61).

A seguir é apresentada a classe App, demonstrando o uso da fila de prioridade em um simples teste. O exemplo utilizado é o mesmo do código demonstrado na introdução deste capítulo, o problema das cores.

```
1.  public class App {
2.     public static void main(String[] args) {
3.
4.            FilaPrioridade f = new FilaPrioridade(6);
5.            try {
6.                   f.enfileira(new Item<String>(“roxo”,
5));
7.                   f.enfileira(new Item<String>(“preto”,
1));
8.                   f.enfileira(new
Item<String>(“laranja”, 0));
9.                   f.enfileira(new
Item<String>(“branco”, 4));
10.                  f.enfileira(new Item<String>(“verde”,
1));
11.                  f.enfileira(new
Item<String>(“amarelo”, 3));
12.           } catch(RuntimeException e) {
13.                  System.err.println(e.getMessage());
14.           }
15.           System.out.printf(“%s\n\n”, f);
16.
17.           try {
18.                  while(!f.ehVazia()) {
19.                         System.out.println(f.
desenfileira());
20.                  }
21.           }catch(RuntimeException e) {
22.                  System.err.println(e.getMessage());
23.           }
24.    }
25. }
```

Na linha 4, uma fila de prioridade é declarada determinando sua capacidade igual a 6 itens. Entre as linhas 5 e 14 os itens são adicionados, informando primeiro o texto representando o valor do item na fila e na sequência a prioridade do item. A fila de prioridade completa é exibida na linha 15. Depois, entre as linhas 17 e 23, é realizado um laço de repetição do tipo *while* para desenfileirar os itens enquanto a fila não estiver vazia. O resultado desse código é apresentado a seguir.

```
[(5 - roxo), (1 - preto), (0 - laranja), (4 - branco), (1
- verde), (3 - amarelo)]

(0 - laranja)
(1 - preto)
(1 - verde)
(3 - amarelo)
(4 - branco)
(5 - roxo)
```

1.1 Trabalhando com o vetor dinâmico

A fila de prioridade implementada possui alguns pontos que dificultam o controle dos objetos. Além de seu tamanho ser limitado, muitas operações executadas são grandes e podem gerar problemas de compreensão e leitura do código-fonte. Para auxiliar uma implementação mais enxuta e com mais recursos de melhor desempenho, é utilizada a classe ArrayList da API Collections do Java. Ela é mais dinâmica e mais confiável para implementar uma fila de prioridade utilizando vetores. A implementação é apresentada a seguir.

```
1.  public class FilaPrioridade {
2.      private List<Item<?>> itens = new
ArrayList<Item<?>>();
3.
4.      public boolean ehVazia() {
5.              return this.itens.isEmpty();
6.      }
7.
8.      public int tamanho() {
```

```
9.            return this.itens.size();
10.    }
11.
12.    public void enfileira(Item<?> i) {
13.           this.itens.add(i);
14.    }
15.
16.    public Item<?> desenfileira() {
17.           if(this.ehVazia()) {
18.                  throw new RuntimeException("A fila
está vazia.");
19.           }
20.
21.           Item<?> altaPrioridade = this.itens.get(0);
22.           for(Item<?> i : this.itens) {
23.                  if(i.getPrioridade() <
altaPrioridade.getPrioridade()) {
24.                         altaPrioridade = i;
25.                  }
26.           }
27.
28.           this.itens.remove(altaPrioridade);
29.           return altaPrioridade;
30.    }
31.
32.    @Override
33.    public String toString() {
34.           return this.itens.toString();
35.    }
36. }
```

Utilizar a API Collections facilita a implementação em muitos pontos. A quantidade de linhas de código entre ambas as classes apresentadas diminui em aproximadamente um terço. Apenas um atributo é necessário ser declarado, o vetor de itens que é um ArrayList. Os métodos *ehVazia* (linhas 4 a 6) e *tamanho* (linhas 8 a 10) foram simplificados em seu corpo. Eles utilizam métodos implementados através da classe ArrayList. O método *enfileira* ficou ainda mais simples, uma vez que não existe a necessidade de se validar se capacidade máxima foi ou não atingida

(linhas 12 a 14). Desenfileirar (linhas 16 a 30) ficou mais fácil, uma vez que é possível passar a referência do objeto que deve ser removido do vetor. Então é necessário apenas percorrer o vetor em busca do item de maior prioridade. Caso existam dois com a mesma prioridade, a ordem de inserção prevalece como maior prioridade. Por fim, o método *toString* também foi simplificado para exibir melhor o vetor como um texto.

Perceba que o método *desenfileira* ficou com o maior trabalho (principalmente na implementação com vetor primitivo), pois ele deve identificar qual é o item de maior prioridade para removê-lo. Mas essa é a estratégia que facilita na hora de armazenar e ela apresenta um desempenho menor na remoção do item. Uma outra estratégia é determinar a posição de enfileiramento no momento da inclusão do item na fila. Assim, o gasto de tempo será menor quando ele for removido. Entretanto, a melhor opção para implementar uma fila de prioridade é utilizando uma estrutura de dados do tipo heap. Essa estrutura será apresentada no tópico a seguir.

2 Heap

As implementações feitas até o momento na fila de prioridade não possuem um desempenho ótimo, o que significa que podem ser melhoradas. O principal ganho que pode existir nas operações de uma fila de prioridade são a inserção ou remoção de um item, que têm em seu pior caso uma complexidade O(n). A ideia para melhorar essas operações básicas é criar um balanceamento entre a inserção e a remoção das informações. Assim, não existe um pior caso apenas em uma ou em outra, ambas conseguem manter um bom desempenho (GOODRICH; TAMASSIA, 2013).

Para balancear a implementação de uma fila de prioridade, utiliza-se uma estrutura de dados chamada heap. O tempo das operações de inserção e remoção de dados em um heap é de O(log(n)). Esse é um excelente resultado quando comparado com as demais implementações baseadas em sequência. Uma estrutura do tipo heap tem como premissa

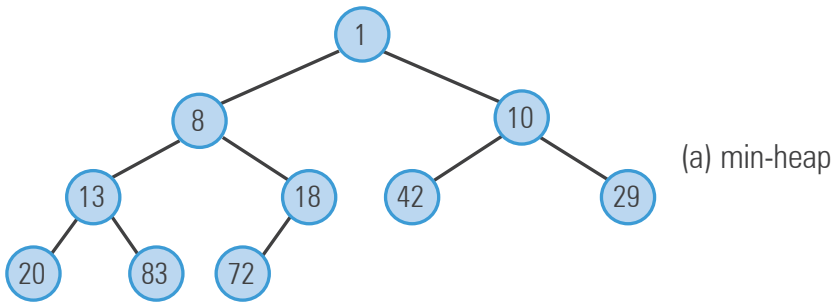
o abandono do uso de uma lista sequencial na maneira de armazenar e buscar uma informação, e como base uma árvore binária (CORMEN *et al*., 2009; GOODRICH; TAMASSIA, 2013).

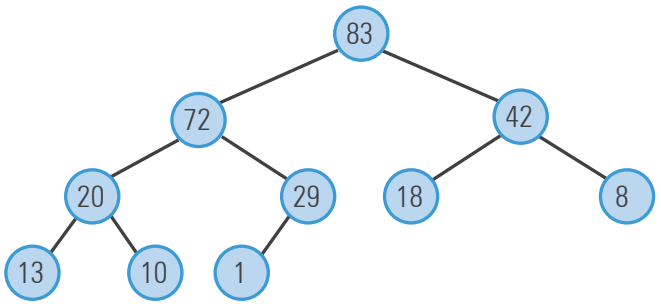
A árvore binária que representa o heap (A) armazena uma coleção de nós internos. Os nós podem ser compostos por chave e valor ou apenas valor, o importante é que exista uma maneira de estabelecer uma ordem de comparação entre eles. Além disso, o heap A deve manter as seguintes propriedades (GOODRICH; TAMASSIA, 2013):

- **relacional:** que define a forma como os nós serão armazenados; e
- **estrutural:** que define a estrutura com base nos próprios nós armazenados.

A ordem dos nós na estrutura heap é dada de duas maneiras: max-heap e min-heap. O max-heap tem como propriedade para ordenação que os nós filhos são sempre menores que o seu nó pai. É importante ressaltar que, pela regra de uma árvore binária (que compõe o heap), um nó só pode gerar dois filhos. O min-heap tem a propriedade de ordenação que os nós pais são sempre menores que seus filhos (NECAISE, 2010). A figura 2 apresenta um exemplo de um heap aplicando as duas propriedades, max-heap e min-heap.

Figura 2 – Duas possíveis ordenações do heap apresentadas: (a) min-heap priorizando o menor elemento; (b) max-heap priorizando o maior elemento





(b) max-heap

A figura 2 (a) apresenta a formação de um heap, no formato de árvore binária, optando pela ordenação do min-heap, na qual os filhos são maiores que seus pais. Enquanto isso, na figura 2 (b) é apresentada a ordenação pela propriedade max-heap, na qual os filhos são menores que seus pais. É importante ressaltar alguns termos que são oriundos da árvore binária e utilizados também no formato heap. O preenchimento de uma árvore binária sempre é realizado da esquerda para a direita. Os nós 1 (min-heap) e 83 (max-heap) são denominados raízes do heap. E os nós que não possuem filhos até o momento são chamados de nós folhas (TENENBAUM; LANGSAM; AUGENSTEIN, 2004; CORMEN *et al*., 2009).

O heap é uma estrutura de dados especializada que possui operações limitadas. A inserção de um novo nó dentro de um heap deixa-o posicionado de acordo com a regra de ordenação adotada. A remoção só é realizada através da raiz do heap. Para continuar os estudos dessa estrutura de dados, os exemplos serão guiados pela propriedade do max-heap. O min-heap pode ser obtido por meio da inversão dos condicionais apresentados no algoritmo.

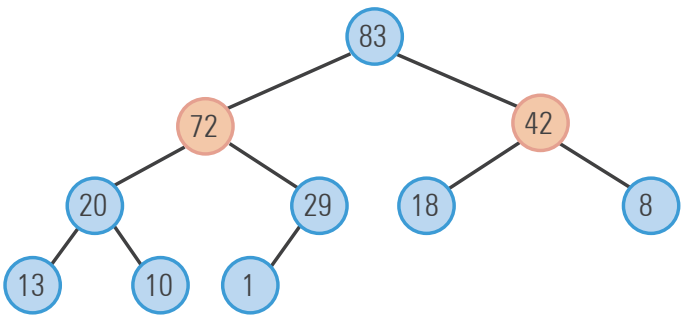
2.1 Inserindo um novo nó

Quando qualquer novo nó for inserido dentro do heap, as propriedades de ordem e formato devem ser mantidas. Sendo assim, identificar

onde o novo nó será inserido é uma pequena parte do problema. A maior questão é como fazer a inserção se a posição em que esse nó deve permanecer está ocupada por um outro nó. Sendo assim, a probabilidade de ocorrer uma movimentação entre os nós é extremamente alta. Dessa forma, o nó deve ser inserido sempre pelas folhas e não pela raiz (que é o sentido de leitura padrão). Antes de inserir por uma folha, deve-se procurar pelo pai que possui apenas um filho. Ele deve ser priorizado (NECAISE, 2010).

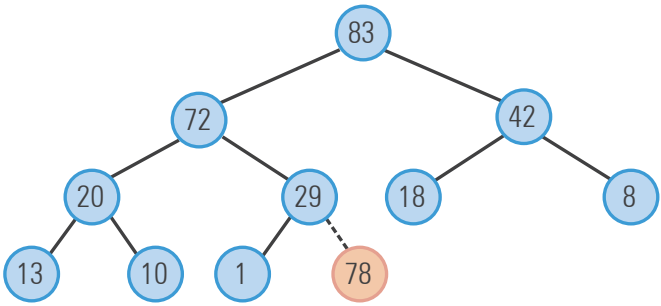
O primeiro passo do algoritmo é identificar as posições que podem ser ocupadas pelo novo nó. Para exemplificar melhor, o max-heap apresentado na figura 2 será utilizado até o final do capítulo. O nó de chave 78 será inserido no heap. A figura apresenta em laranja as duas possíveis posições que o nó 78 pode ocupar após o processo de inserção.

Figura 3 – Em laranja são apresentadas as possíveis posições que o nó 78 pode ocupar depois do processo de inserção



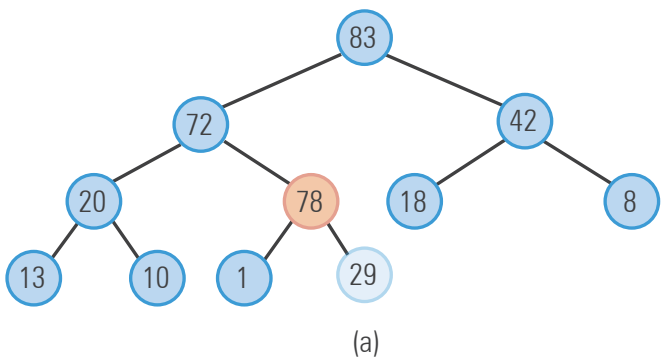
Na sequência, o nó é criado e anexado com uma folha, no exemplo, ele é uma folha, filha do nó 29, que possui apenas um filho. Veja na figura 4 o nó 78 sendo anexado como uma folha (elemento na cor laranja).

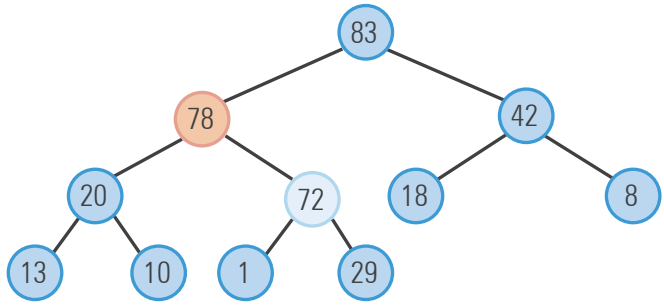
Figura 4 – Nó 78 (em laranja) é criado e anexado como uma folha, filha de 29



Nesse momento, a propriedade max-heap de ordenação está comprometida. Para restaurar a propriedade de ordenação do heap, o nó inserido deve percorrer o caminho reverso do heap, em direção à raiz, até encontrar a posição correta. Essa operação é chamada de *sift-up*, que também é conhecida como *up-heap, heapify-up, bubble-up,* entre outros nomes (NECAISE, 2010). A figura 5 (a) mostra o primeiro passo da operação *sift-up*, na qual existe uma troca entre os nós 29 (cor mais clara) e 78 (em laranja). A figura 5 (b) exibe o segundo e último passos, para o exemplo, nos quais a troca do nó 78 (em laranja) é feita com o nó 72 (cor mais clara), reestabelecendo a ordem do max-heap.

Figura 5 – Apresentação do processo de *sift-up* reestabelecendo a ordem max-heap





(b)

Com o balanceamento realizado, a operação para a inserção do nó de chave 78 foi concluída. Na próxima seção, é apresentado o processo da operação de remoção do nó raiz da estrutura de dados heap.

2.2 Removendo um nó

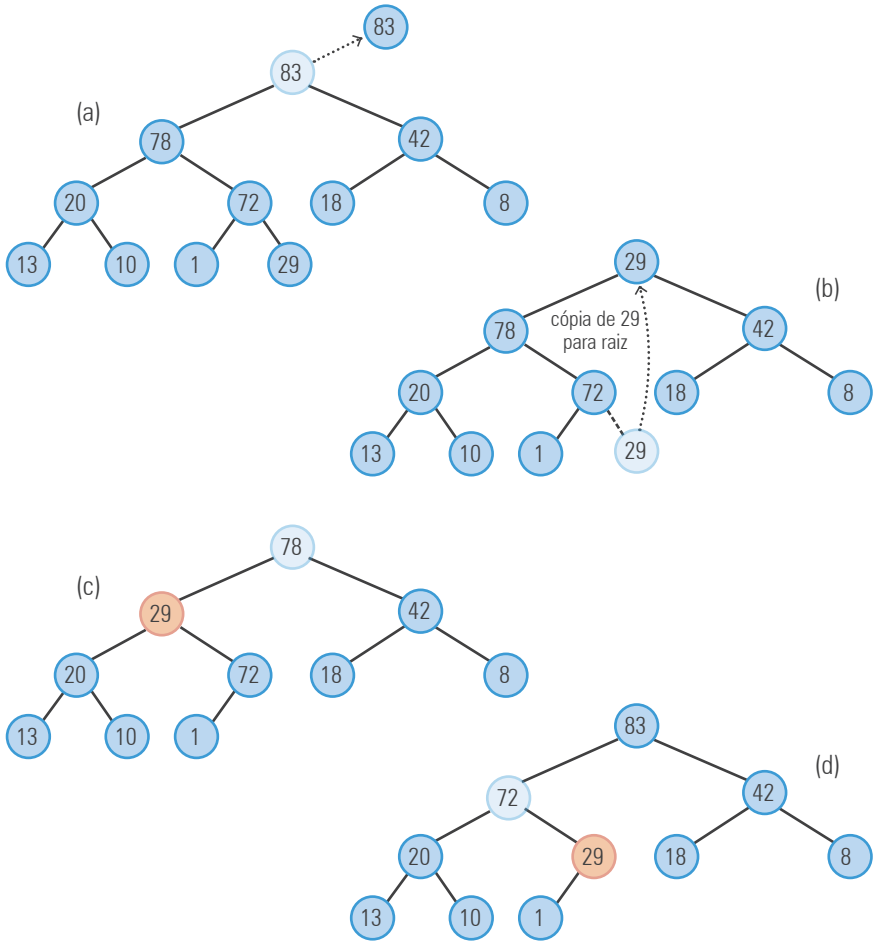
A extração de um nó dentro de uma estrutura de heap só pode ocorrer através do nó raiz. Sendo assim, se a ordenação for realizada através da regra max-heap, sempre serão removidos os nós maiores primeiro. No caso de se optar por uma implementação de min-heap, os nós menores terão prioridade na remoção (NECAISE, 2010; GOODRICH; TAMASSIA, 2013).

Para que a ordem seja mantida após a remoção de um nó, um nó folha deve assumir o lugar da posição do nó raiz. Apesar de esse processo de substituição ser claro e de fácil implementação, ainda não garante por completo todas as propriedades necessárias para manter a ordem do heap (NECAISE, 2010). A folha escolhida, pela regra da árvore binária, é a mais à direita da árvore.

Após esse processo de cópia, o nó que foi copiado para a raiz deverá percorrer o caminho da árvore até que ela se torne novamente uma folha. Quando o processo for finalizado, a estrutura heap novamente

estará organizada e ordenada. O processo é conhecido como *sift-down* e é apresentado na figura 6.

Figura 6 – Processo para remoção do nó raiz 83 junto com o *sift-down* aplicado ao nó 29, após seu reposicionamento no heap



No processo de remoção apresentado pela figura 6, é possível identificar os quatro passos executados para manter as propriedades de um heap. No passo (a), o nó 83, que era a raiz, é removido do heap. Para reestabelecer a ordem e o formato do heap, a última folha à direita é

reposicionada como a nova raiz do heap, conforme o passo (b). Nesse momento, o processo *sift-down* é inicializado e o nó raiz corrente (chave 29, em laranja) é trocado de posição com o 78 (cor mais clara). O primeiro passo (c) do *sift-down* é realizado e, em sequência, o segundo passo (d). Neste último, existe a troca entre os nós de chave 72 (cor mais clara) e 29 (em laranja). O heap, agora, está balanceado novamente.

2.3 Implementando um heap

Com as operações definidas, é hora de implementar a estrutura de heap apresentada com a propriedade de max-heap. Confira a seguir a implementação da classe Heap em Java.

```
1.  public class Heap {
2.      private List<Item<?>> itens = new
ArrayList<Item<?>>();
3.
4.      public int tamanho() {
5.              return this.itens.size();
6.      }
7.
8.      public boolean ehVazia() {
9.              return this.itens.isEmpty();
10.     }
11.
12.     public void adiciona(Item<?> i) {
13.             this.itens.add(i);
14.             this.siftUp(this.itens.size() - 1);
15.     }
16.
17.     public Item<?> remove() {
18.             Item<?> removido = this.itens.get(0);
19.             this.itens.set(0, this.itens.get(this.itens.
size() - 1));
20.             this.itens.remove(this.itens.size() - 1);
21.             this.siftDown(0);
22.             return removido;
```

```
23.     }
24.
25.     private void siftUp(int n) {
26.             if (n > 0) {
27.                     int pai = Math.floorDiv(n, 2);
28.                     if(this.itens.get(n).getPrioridade()
>
29.                             this.itens.get(pai).
getPrioridade()) {
30.                             this.swap(n, pai);
31.                             this.siftUp(pai);
32.                     }
33.             }
34.     }
35.
36.     private void siftDown(int n) {
37.             int esquerda = 2 * n + 1;
38.             int direita = 2 * n + 2;
39.             int maior = n;
40.
41.             if((esquerda < this.itens.size()) &&
42.                             this.itens.get(esquerda).
getPrioridade() >=
43.                             this.itens.get(maior).
getPrioridade()) {
44.                     maior = esquerda;
45.             } else if((direita < this.itens.size()) &&
46.                             this.itens.get(direita).
getPrioridade() >=
47.                             this.itens.get(maior).
getPrioridade()) {
48.                     maior = direita;
49.             }
50.
51.             if(maior != n) {
52.                     this.swap(n, maior);
53.                     this.siftDown(maior);
54.             }
55.
56.     }
57.
```

```
58.     private void swap(int a, int b) {
59.             Item<?> temp = this.itens.get(a);
60.             this.itens.set(a, this.itens.get(b));
61              this.itens.set(b, temp);
62.     }
63.
64.     @Override
65.     public String toString() {
66.             return this.itens.toString();
67.     }
68. }
```

No código apresentado, é utilizado um ArrayList para armazenar os itens da estrutura heap. Apesar de ser utilizada uma lista, em sua essência, a forma de percorrer é diferente, ou seja, não é mais linear. A forma de percorrer a lista caracteriza essa estrutura de dados como uma árvore binária. Os métodos de suporte *tamanho* (linhas 4 a 6) e *ehVazia* (linhas 8 a 10) são os primeiros implementados na classe Heap. Na sequência, é apresentada a implementação do método *adiciona* (linhas 12 a 15), que recebe como parâmetro um Item, o mesmo tipo definido no início do capítulo para manter o contexto da fila de prioridade. Após adicionar o item no ArrayList (linha 13), o método *siftUp* é invocado, recebendo o último índice da lista como parâmetro. O método *siftUp* (linhas 25 a 34) verifica se o índice recebido não se refere ao índice da raiz (linha 26), depois identifica o índice do nó pai, que é o número inteiro da divisão do índice recebido por 2 (linha 27). Com o pai determinado, verifica-se se o nó tem prioridade maior que a do nó pai. Em caso positivo, é feita uma troca de posições com o auxílio do método *swap* (linhas 58 a 62). O índice do nó pai é o alvo agora, então ele é enviado em uma chamada recursiva do método *siftUp* até todos os nós terem sido percorridos.

O próximo método é o método *remove* (linhas 17 a 23), que faz uma cópia do item a ser removido (linha 18), copiando a última folha para a posição da raiz (linha 19). Na sequência, o método *siftDown* é invocado, recebendo 0 (o índice da raiz) como parâmetro. No método *siftDown*

(linhas 36 a 56), são selecionados os filhos da esquerda (linha 37) e da direita (linha 38) e o nó recebido é identificado como o maior (linha 39). Na sequência, os nós da esquerda e da direita são comparados e verificados se são maiores que os nó pai (linhas 41 a 49). Em caso positivo, eles passam a ser o índice do maior elemento e é feita uma troca com o pai (linha 52). Depois, o método *siftDown* é chamado de maneira recursiva para continuar a ordenação do heap.

Apesar de a implementação seguir a propriedade de max-heap, para utilizar o heap como uma fila de prioridade basta implementá-lo com a propriedade min-heap.

3 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Implemente o heap para uma fila de prioridade.
2. Execute o teste para verificar o desempenho de cada uma das estruturas. Faça uma tabela comparativa.
3. Implemente um sistema de emissão de passagens aéreas em um terminal com regras de prioridades para idosos, gestantes e adultos acompanhados de crianças. Ao todo, devem ser 4 níveis de prioridade, pelo menos.

Considerações finais

Apresentamos ao longo deste capítulo os conceitos de fila de prioridade e heap e de como eles se relacionam. Eles podem ser utilizados em diversas aplicações que envolvam classificações de prioridade, como o caso de sistemas de mensageria e até em filas de bancos, supermercados e aeroportos, onde existam regras de prioridades ou preferenciais. Algumas implementações não são tão triviais e até o resultado prático

pode divergir um pouco do teórico, já que a divisão visual é mais intuitiva do que a divisão da árvore através do código. São muitos os arredondamentos que podem divergir um pouco do visual. Contudo, isso não quer dizer que haja algum erro, as propriedades são e sempre devem ser mantidas durante todo o processo. Alinhando o conhecimento da estrutura de dados com os algoritmos, muitas possibilidades devem ser exploradas para otimizações e soluções de novos problemas.

Referências

CORMEN, Thomas H. *et al*. **Introduction to algorithms**. Cambridge: MIT Press, 2009.

DEITEL, Paul J.; DEITEL, Harvey M. **Java**: como programar. São Paulo: Pearson, 2008.

FORBELLONE, André Luiz Villar; EBERSPACHER, Henri Frederico. **Lógica de programação**: a construção de algoritmos e estruturas de dados. São Paulo: Prentice Hall, 2005.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

NECAISE, Rance D. **Data structures and algorithms using Python**. Hoboken: John Wiley & Sons, Inc., 2010.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: MAKRON Books, 2004.

Sobre o autor

Andrey Araujo Masiero é doutor em engenharia elétrica na área de inteligência artificial pelo Centro Universitário da Fundação Educacional Inaciana Padre Sabóia de Medeiros (FEI) e bolsista PROSUP/CAPES. Mestre em engenharia elétrica na área de inteligência artificial pelo Centro Universitário FEI e bolsista em tempo integral do projeto FINEP Pesquisa e Estatística baseada em Acervo Digital de Prontuário Médico do Paciente em Telemedicina Centrada no Usuário. Graduado em ciência da computação pelo Centro Universitário FEI em 2009. Premiado com o primeiro lugar na Expocom com o trabalho Sistema de Gerenciamento de Patterns, Anti-Patterns e Personas (SIGEPAPP). Profissional com 16 anos de experiência no mercado, participou de projetos com o governo do Estado de São Paulo e em projetos da sociedade privada, como a integração entre os bancos Santander e Real. Áreas de interesse: engenharia de usabilidade, interação homem-computador, interação humano-robô, engenharia de software, inteligência artificial, aprendizado de máquina, data mining e algoritmos computacionais.