Apontamentos de Estruturas de Dados

Queues

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Outubro de 2022

Índice

7. Queues

7.1. Introdução

As *Queues* são o que costumamos chamar de filas, como "por favor, entre na fila para um almoço grátis". As características essenciais de uma fila são que ela é ordenada e que o acesso à fila é restrito às suas extremidades: as coisas podem entrar na fila apenas por trás e sair da fila apenas pela frente.

> **Queue:** Dispensador que contém uma sequência de elementos que permite inserções apenas numa extremidade, chamada de traseira, e remoções e acesso a elementos na outra extremidade, chamada de frente.

As filas também são chamadas de listas *first-in-first-out* (primeiro a entrar primeiro a sair), ou listas FIFO. As filas são importantes na computação devido aos muitos casos em que os recursos fornecem serviços por ordem de chegada, como trabalhos enviados a uma impressora ou processos que aguardam pelo CPU num sistema operativo.

Uma fila é normalmente representada na horizontal. Uma extremidade da fila é a parte traseira (ou cauda) - `rear`, onde os elementos são adicionados, o outro lado é a frente (ou cabeça) - `front`, a partir da qual os elementos são removidos. Ao contrário de uma pilha (*Stack*), que opera no final da coleção, uma fila opera em ambas as extremidades.

7.2. Aplicação de Filas

As filas são estruturas muito importantes em simulações de computador, processamento de dados, gestão de informação assim como no âmbito dos sistemas operativos. Nas simulações, as estruturas de filas são usadas para representar eventos da vida real, como filas de carros em cruzamentos de semáforos e postos de combustível, filas de pessoas no ponto de *check-out* em supermercados, filas de clientes de bancos, etc.

Nos sistemas operativos as estruturas de filas são usadas para representar diferentes programas na memória do computador na ordem em que são executados. Por exemplo, se um programa `J` é enviado antes do programa `K`, então o programa `J` é adicionado antes do programa `K` na memória do computador e o programa `J` é executado antes do programa `K`.

7.3. Queue ADT

As filas são *containers* e contêm valores de algum tipo. Devemos portanto falar da `Queue ADT` de `T`, onde `T` é o tipo dos elementos mantidos na fila. O conjunto de valores deste tipo é o conjunto de todas as filas que contêm elementos do tipo `T`. O conjunto de valores inclui assim a fila vazia, as filas com um elemento do tipo `T`, as filas com dois elementos do tipo `T` e assim por diante. As operações essenciais deste tipo são as seguintes:

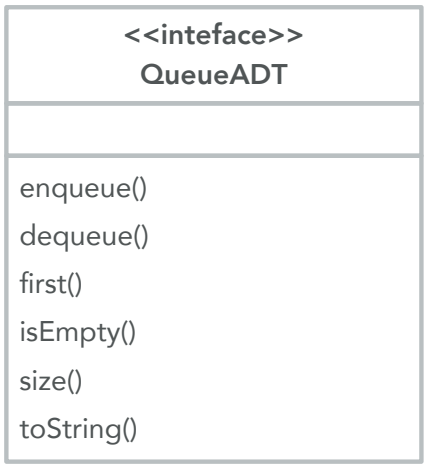
Operação	**Descrição**
`enqueue`	Adiciona um elemento à traseira da fila
`dequeue`	Remove o elemento da frente da fila
`first`	Examina o elemento na frente da fila
`isEmpty`	Determina se a fila está vazia
`size`	Determina o número de elementos da fila
`toString`	Representação da fila em string

O termo `enqueue` é usado para se referir ao processo de adicionar um elemento na fila (*Queue*). Da mesma forma, retirar da fila (`dequeue`) é o processo de remoção do elemento no nó que está na frente da fila. Como uma pilha, uma fila não permite aos utilizadores aceder

aos elementos no meio da fila. O método `toString` é considerado apenas por conveniência, não devem ser criadas operações que dependem desse método.

7.4. Interface Queue

Tal como na pilha, a fila é uma estrutura semelhante a uma lista que fornece acesso restrito aos seus elementos. Os elementos da fila só podem ser inseridos na parte de trás (operação `enqueue`) e removidos da frente (operação `dequeue`). As filas funcionam tal como quando estamos na fila de um balcão de cinema por exemplo. Se ninguém fizer batota, as novas pessoas vão para o final da fila. A pessoa na frente da fila é a próxima a ser atendida. Assim, as filas libertam os seus elementos por ordem de chegada. A figura seguinte mostra um exemplo de interface `Queue ADT` tendo por base as operações descritas no ponto anterior.




```
1. public interface QueueADT<T> {
2.     /**
3.      * Adds one element to the rear of this queue.
4.      * @param element  the element to be added to
5.      * the rear of this queue
6.      */
7.     void enqueue(T element);
8.     /**
9.      * Removes and returns the element at the front of
10.     * this queue.
11.     *
```

```
12.     * @return  the element at the front of this queue
13.     */
14.    T dequeue();
15.    /**
16.     * Returns without removing the element at the front of
17.     * this queue.
18.     * @return  the first element in this queue
19.     */
20.    T first();
21.    /**
22.     * Returns true if this queue contains no elements.
23.     *
24.     * @return  true if this queue is empty
25.     */
26.    boolean isEmpty();
27.    /**
28.     * Returns the number of elements in this queue.
29.     *
30.     * @return  the integer representation of the size
31.     * of this queue
32.     */
33.    int size();
34.    /**
35.     * Returns a string representation of this queue.
36.     * @return  the string representation of this queue
37.     */
38.    String toString();
39.}
```

7.5. Usar Filas – Um Exemplo

A técnica da cifra de César é a primeira e mais simples técnica de criptografia. É simplesmente um tipo de cifra de substituição, ou seja, cada letra de um determinado texto é substituída por uma letra num número fixo de posições no alfabeto. Por exemplo, com um deslocamento de 1, A seria substituído por B, B tornaria-se C e assim por diante. O método supostamente recebeu o nome de Júlio César que aparentemente o usou para se comunicar com os seus

exércitos. Assim, para cifrar um determinado texto precisamos de um valor inteiro conhecido como deslocamento que indica o número de posições que cada letra do texto foi movida para baixo.

A encriptação pode ser representada usando aritmética modular, primeiro transformando as letras em números de acordo com o esquema, `A = 0, B = 1,…, Z = 25`. A encriptação de uma letra por um deslocamento `n` pode ser descrita matematicamente como:

$$E_n(x) = (x+n) \bmod 26$$

E da desencriptação:

$$D_n(x) = (x-n) \bmod 26$$

Uma melhoria pode ser feita alterando o quanto uma letra é mudada de posição dependendo do local onde a letra está na mensagem – através de uma chave de repetição. A chave de repetição é uma série de números inteiros que determina quanto é que cada caracter será movido. Por exemplo, considere a seguinte chave de repetição:

3 1 7 4 2 5

O primeiro caracter da mensagem é deslocado 3 posições, o seguinte 7, e assim por diante. Quando a chave está esgotada é recomeçado do início da chave.

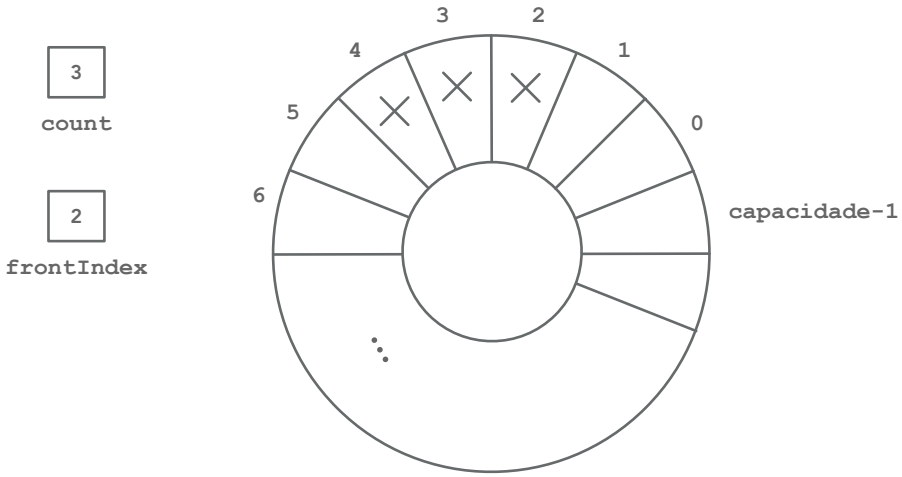
Mensagem Codificada	n	o	v	a	n	j	g	h	l		m	u		u	r	x	l	v
Chave	3	1	7	4	2	5	3	1	7		4	2		5	3	1	7	4
Mensagem Descodificada	k	n	o	w	l	e	d	g	e		i	s		p	o	w	e	r

7.6. Implementar uma Queue ADT com recurso a um array

Existem duas abordagens para implementar uma `QueueADT`: uma implementação contígua através de *arrays* e uma implementação através de listas simplesmente ligadas.

A implementação de filas de elementos do tipo `T` através de *arrays* requer um *array* `T` para armazenar o conteúdo da fila e de alguma forma acompanhar a frente e a cauda da fila. Podemos, por exemplo, decidir que o elemento da frente da fila (se existir) será sempre armazenado na posição 0 e guardar o tamanho da fila, o que implica que o elemento na cauda estaria no local `tamanho-1`. Esta abordagem exige que os dados sejam movidos para frente no *array* sempre que um elemento sai o que não é muito eficiente.

Uma solução inteligente para este problema é permitir que os dados no *array* "flutuem" para cima à medida que os elementos entram e saem e dessa forma voltam para o início do *array* quando necessário. É como se as posições no *array* estivessem num formato circular em vez de linear. A figura seguinte ilustra esta solução. Os elementos da fila são mantidos no *array* de armazenamento. A variável `frontIndex` mantém o controle da posição do *array* que contém o elemento na frente da fila e `count` contém o número de elementos na fila. A capacidade é o tamanho do *array* e, portanto, o número de elementos que podem ser armazenados na fila.



Como podemos ver na figura, os dados ocupam as regiões com um `X`: existem três elementos na fila, com o elemento da cabeça na posição `[2]` (como indicado pela variável `frontIndex`) e o elemento da cauda na posição `[4]` (`frontIndex+count-1` mod capacidade é 4). O próximo elemento a entrar na fila será colocado na posição `[5]`; De uma forma geral, os elementos são colocado na fila segundo a seguinte fórmula:

(frontIndex+count) mod capacity

A divisão modular é o que faz os valores da fila formarem uma distribuição circular do final do *array* até o início. Este truque de usar um *array* circular é a abordagem padrão para implementar filas em posições contíguas.

Se for usado um *array* fixo a fila poderá ficar cheia; se for usado um *array* dinâmico a fila será essencialmente ilimitada. Normalmente, redimensionar um *array* é uma operação computacionalmente pesada porque um novo *array* deve ser alocado, o conteúdo do *array* copiado e o antigo libertado, então essa flexibilidade é adquirida a um custo. Também se deve ter cuidado ao mover os elementos corretamente para o *array* expandido - lembre-se que a frente da fila pode estar numa posição ao meio de um *array* completo com elementos envolvendo a frente original do *array*.

A implementação das operações da `QueueADT` com recurso a esta estrutura de dados é bastante direta. Por exemplo, para implementar a operação `dequeue()`, primeiro é feita uma verificação de que a pré-condição da operação (que a fila não está vazia) não foi violada testando se `count` é igual a `0`. Caso contrário, o valor em `frontIndex` é guardado numa variável temporária, `frontIndex` é definida como `(frontIndex+1) mod capacity`, `count` é decrementado e o valor armazenado na variável temporária é retornado.

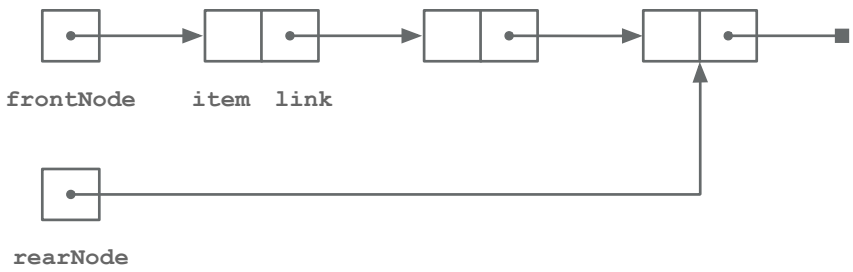
Uma classe que especifica esta implementação pode ser chamada de `ArrayQueue<T>`. Esta irá implementar a interface `QueueADT<T>` e terá o *array* de armazenamento e as variáveis `frontIndex` e `count` como atributos privados. As operações de fila serão métodos públicos. O construtor irá criar uma fila vazia. O valor da capacidade pode ser uma constante definida em tempo de compilação ou um parâmetro do construtor.

7.7. Implementar uma Queue ADT com recurso a uma lista ligada

Uma implementação ligada de uma `QueueADT` usa uma estrutura de dados ligada para representar os valores do conjunto. Uma lista simplesmente ligada é tudo o que é necessário,

portanto os nós da lista precisam conter apenas um valor do tipo `T` e uma referência ao próximo nó. Poderíamos manter uma referência apenas para o início da lista mas isso exigiria mover a lista do início para o fim sempre que uma operação exigisse a manipulação da outra extremidade da fila, portanto, é mais eficiente manter uma referência em cada extremidade da lista. Assim, iremos usar as referências `frontNode` e `rearNode` para acompanhar as duas extremidades da lista.

Se `rearNode` se referir ao início da lista e `frontNode` ao final será impossível remover elementos da fila sem percorrer a lista a partir do seu topo; por outras palavras, não ganhamos nada ao usar uma referência extra. Assim, devemos fazer com que `frontNode` se refira à cabeça da lista e `rearNode` à cauda. A figura seguinte ilustra essa estrutura de dados.



Cada nó possui um campo de dados (para valores do tipo `T`) e um campo de ligação (para as referências). A figura anterior apresenta uma fila com três elementos. A fila está vazia quando as referências `frontNode` e `rearNode` são nulas; de realçar que a fila nunca fica cheia (a menos que a memória seja esgotada).

Implementar as operações da `QueueADT` com recurso a uma estrutura de dados ligada é bastante simples embora seja necessário algum cuidado para manter as duas referências sincronizadas. Por exemplo, para implementar a operação `dequeue()`, primeiro é feita uma verificação de que a fila não está vazia. Caso contrário, o campo de valor do nó frontal é atribuído a uma variável temporária. À variável `frontNode` é então atribuída o campo de ligação do nó a remover. Se `frontNode` for `null` a lista ficou vazia, portanto, ao `rearNode` também deve ser atribuído `null`. Por fim, o valor guardado na variável temporária é retornado.

Uma classe que realiza esta implementação pode ser chamada de `LinkedQueue<T>`. Esta classe irá implementar a interface `Queue<T>`, manter `frontNode` e `rearNode` como atributos privados e as operações de fila como métodos públicos. À semelhança da implementação da pilha também iremos ter uma classe nó interna e privada para instâncias dos vários nós, é esta a estrutura ligada que irá manter os dados. O construtor deve inicializar os valores de `frontNode` e `rearNode` a `null` e `count` a `0`. À medida que são adicionados novos nós, a fila irá crescer através da instanciação de novos nós com os elementos que constitui a lista ligada.

7.8. Notas Finais

Ambas as implementações de fila são simples e eficientes mas a implementação contígua coloca uma restrição de tamanho na fila ou usa uma técnica de realocação pesada se uma fila ficar muito grande. Se as filas implementadas de forma contígua forem muito grandes para garantir que não iremos ter problemas, o espaço pode ser desperdiçado.

Uma implementação em lista ligada é essencialmente ilimitada, portanto, a fila nunca fica cheia. Também é muito eficiente no uso de espaço pois aloca apenas os nós suficientes para armazenar os valores realmente mantidos na fila.
No geral, a implementação em lista ligada de filas é melhor do que a implementação contígua.

7.9. Exercícios Propostos

Exercício 1

Porquê que usamos uma implementação em *array* circular? Porque não um *array* normal como numa *Stack*?

Exercício 2

Qual o *Big O* para a operação `dequeue` nas diferentes implementações de um *Queue*: *array*, *array* circular e lista ligada?

Exercício 3

Implementar um programa que dadas duas *Queues* ordenadas as junte numa também ordenada.

Exercício 4

Implementar uma nova *Queue* que obedeça à interface `QueueADT` através de duas *Stacks*.

Exercício 5

Dadas **N** *strings*, criar **N** *Queues* cada uma contendo uma das *strings*. De seguida criar uma *Queue* das N *Queues*. Repetidamente aplicar uma operação de junção ordenada às primeiras duas *Queues* e reinserir a nova *Queue* no final. Repetir até que a *Queue* contenha apenas uma *Queue*.

Exercício 6

Na implementação de armazenamento contíguo de uma fila, é possível acompanhar apenas a localização do elemento na cabeça (através da variável `frontIndex`) e na cauda (através da variável `rearIndex`), sem a variável de contagem? Em caso afirmativo, explique como.

Exercício 7

Uma `LinkedQueue` pode ser implementada através de uma lista duplamente ligada? Quais são as vantagens ou desvantagens dessa abordagem?