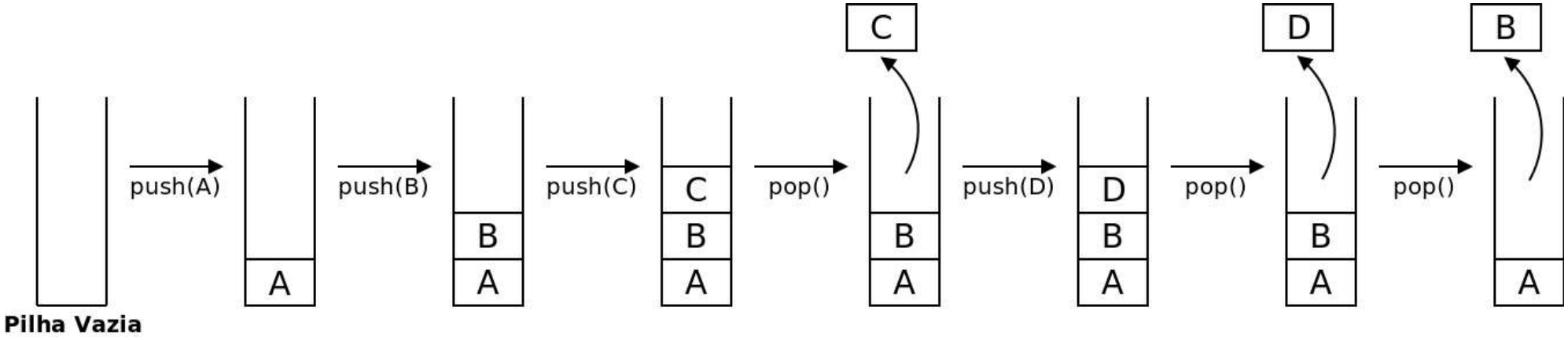
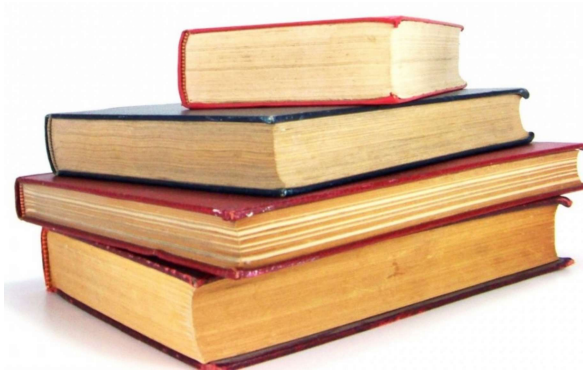
Pilhas

- Uma **Pilha** (**Stack**) é um TAD para guardar uma coleção de elementos suportando duas operações principais:
  - **push(x)** que adiciona um elemento x à coleção
  - **pop()** que retira o último elemento que foi adicionado



- Por ter estas propriedades diz-se que é **LIFO** (*Last In, First Out*)
- Uma pilha simula precisamente uma "pilha de coisas", de objectos *empilhados* uns em cima dos outros.



Pilhas - Interface MyStack

- Para além do push(x) e pop() é usual ter-se operações **top()** para ver o elemento no topo da pilha, **size()** (para saber o tamanho) e **isEmpty()** (para saber se está vazia)
- Vamos criar um interface **MyStack** para representar este TAD. Recordem que o interface só define a API (os métodos), mas não como implementá-los. (usamos o nome *MyStack* para distinguir do *Stack* que existe na própria linguagem Java)

```
public interface MyStack<T> {

    // Métodos que modificam a pilha
    public void push(T v); // Coloca um valor no topo da pilha
    public T pop();        // Retira e retorna o valor no topo da pilha

    // Métodos que acedem a informação (sem modificar)
    public T top();            // Retorna valor no topo da pilha
    public int size();         // Retorna quantidade de elementos na pilha
    public boolean isEmpty(); // Indica se a pilha está vazia
}
```

Pilhas - Uma possível implementação

- Para implementar podemos por exemplo usar... **listas ligadas**
- A implementação quase *direta* a partir dos métodos que já temos. Estamos só a **adaptar** uma classe existente, expondo-a num interface.
- Usaremos **listas duplamente ligadas** (como também poderiam ter sido listas ligadas simples ou circulares)

```
public class LinkedListStack<T> implements MyStack<T> {
   private DoublyLinkedList<T> list;

   LinkedListStack() { list = new DoublyLinkedList<T>();}

   public void push(T v) { list.addFirst(v); }
   public T pop() {
      T ans = list.getFirst();
      list.removeFirst();
      return ans;
   }
   public T top() { return list.getFirst();}
   public int size() {return list.size();}
   public boolean isEmpty() {return list.isEmpty();}
   public String toString() {return list.toString();}
}
```

Pilhas - Um exemplo de utilização

- Agora estamos prontos para criar e usar pilhas:

```
public class TestMyStack {
   public static void main(String[] args) {
      // Criação da pilha
      MyStack<Integer> s = new LinkedListStack<Integer>();

      // Exemplo de inserção de elementos na pilha
      for (int i=1; i<=8; i++)
         s.push(i); // insere i no topo da pilha
      System.out.println(s);
      // Exemplo de remoção de elementos na pilha
      for (int i=0; i<4; i++) {
         int aux = s.pop(); // retira o elemento no topo da pilha
         System.out.println("s.pop() = " + aux);
      }
      System.out.println(s);
      // Exemplo de utilização dos outros métodos
      System.out.println("s.size() = " + s.size());
      System.out.println("s.isEmpty() = " + s.isEmpty());
      System.out.println("s.top() = " + s.top());
   }
}
```

Pilhas - Usando outra implementação

- Notem como foi feita a criação da pilha:

```
MyStack<Integer> s = new LinkedListStack<Integer>();
```

  - A variável *s* é do tipo `MyStack` (um interface). Isto permite que no resto do código possam ser usadas operações como $s.push(x)$ ou $s.pop()$
  - À variável é atribuida uma nova instância de... `LinkedListStack`. Isto é possível porque essa classe **implementa** o MyStack e define qual a implementação real da pilha.

- Imaginando que tínhamos uma outra implementação de pilhas baseada em arrays chamada `ArrayStack`, bastaria mudar a linha para:

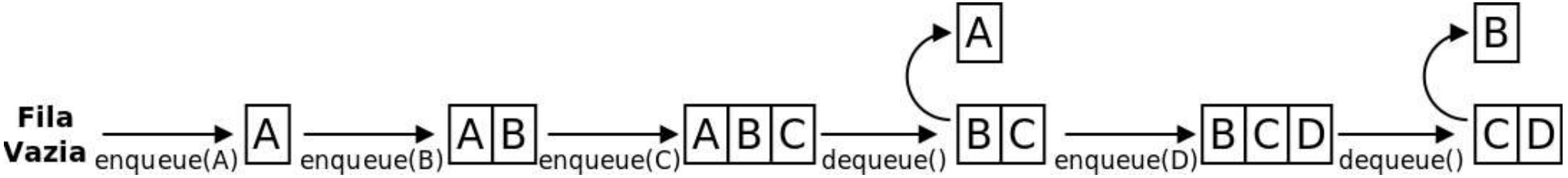
```
MyStack<Integer> s = new ArrayStack<Integer>();
```

  Todo o resto do programa continuaria a funcionar (sem mudar mais nada), sendo que agora estaria a ser usada a implementação baseada em arrays (podem ver um ex. de implementação com arrays no site).

- Isto dá-vos logo uma boa ideia do potencial de usar *interfaces*!

Filas

- Uma **Fila** (**Queue**) é um TAD para guardar uma coleção de elementos suportando duas operações principais:
  - **enqueue(x)** que adiciona um elemento x à coleção
  - **dequeue()** que retira o elemento que foi adicionado há mais tempo



- Por ter estas propriedades diz-se que é **FIFO** (*First In, First Out*)
- Uma pilha simula precisamente uma "fila de objectos", como uma fila de atendimento ao público num supermercado ou num banco

Filas - Interface MyQueue

- Para além do push(x) e pop() é usual ter-se operações **first()** para ver o elemento no início da fila, **size()** (para saber o tamanho) e **isEmpty()** (para saber se está vazia)
- Vamos criar um interface **MyQueue** para representar este TAD. Recordem que o interface só define a API (os métodos), mas não como implementá-los. (usamos o nome *MyQueue* para distinguir da *Queue* que existe na própria linguagem Java)

```
public interface MyQueue<T> {

    // Métodos que modificam a fila
    public void enqueue(T v); // Coloca um valor no final da fila
    public T dequeue();       // Retira e retorna o valor no inicio da fila

    // Métodos que acedem a informação (sem modificar)
    public T first();         // Retorna valor no inicio da fila
    public int size();        // Retorna quantidade de elementos na fila
    public boolean isEmpty(); // Indica se a fila está vazia
}
```

Filas - Uma possível implementação

- Vamos implementar com listas ligadas tal como fizemos com as pilhas
- Tal como anteriormente, estamos só a **adaptar** uma classe existente, com implementação quase directa a partir dos métodos que já temos.

```
public class LinkedListQueue<T> implements MyQueue<T> {
   private DoublyLinkedList<T> list;

   LinkedListQueue() { list = new DoublyLinkedList<T>();}

   public void enqueue(T v) { list.addLast(v); }
   public T dequeue() {
      T ans = list.getFirst();
      list.removeFirst();
      return ans;
   }
   public T first() { return list.getFirst();}
   public int size() {return list.size();}
   public boolean isEmpty() {return list.isEmpty();}

   public String toString() {return list.toString();}
}
```

Filas - Um exemplo de utilização

- Agora estamos prontos para criar e usar filas:

```
public class TestMyQueue {
   public static void main(String[] args) {
      // Criação da fila
      MyQueue<Integer> q = new LinkedListQueue<Integer>();

      // Exemplo de inserção de elementos na fila
      for (int i=1; i<=8; i++)
         q.enqueue(i); // insere i no final da fila
      System.out.println(q);
      // Exemplo de remoção de elementos da fila
      for (int i=0; i<4; i++) {
         int aux = q.dequeue(); // retira o elemento no topo da pilha
         System.out.println("q.dequeue() = " + aux);
      }
      System.out.println(q);
      // Exemplo de utilização dos outros métodos
      System.out.println("q.size() = " + q.size());
      System.out.println("q.isEmpty() = " + q.isEmpty());
      System.out.println("q.first() = " + q.first());
   }
}
```

Deques - Interface

- Um **Deque** (*double ended queue*) é um TAD que generaliza uma fila e permite inserir e remover elementos no início ou no final da fila (mas não inserir ou remover no meio).
- Corresponde quase à implementação que fizemos de lista ligadas:

```
public interface MyDeque<T> {

    // Métodos que modificam o deque
    public void addFirst(T v); // Coloca um valor no início da fila
    public void addLast(T v);  // Coloca um valor no final da fila
    public T removeFirst();    // Retira e retorna o valor no inicio da fila
    public T removeLast();     // Retira e retorna o valor no final da fila

    // Métodos que acedem a informação (sem modificar)
    public T first();          // Retorna valor no inicio da fila
    public T last();           // Retorna valor no final da fila
    public int size();         // Retorna quantidade de elementos na fila
    public boolean isEmpty();  // Indica se a fila está vazia
}
```

Deques - Implementação

- Uma implementação com listas duplamente ligadas está disponível no site:
  - **class** `LinkedListDeque<T>`
- Um exemplo de utilização está também disponível no site:
  - **class** `TestDeque`