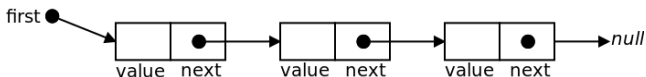


Listas Ligadas, Pilhas e Filas

Pedro Ribeiro

DCC/FCUP

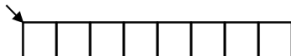
2021/2022



(baseado e/ou inspirado parcialmente nos slides de Luís Lopes e de Fernando Silva)

Estruturas de Dados Sequenciais

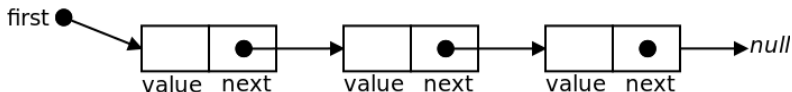
- **Estruturas de dados sequenciais:** armazenam **seqüências** finitas de valores do mesmo tipo (numa seqüência existe **ordem**: 1^o valor, 2^o valor, 3^o valor). Ex:
 - ▶ Números: 1, 2, 3, 4, 5, 6, 7, ...
 - ▶ Strings: "Portugal", "Espanha", "França", ...
 - ▶ Pessoas: ("Ana", 20), ("Carlos", 23), ("Diana", "19")
- Já conhecemos uma estrutura de dados sequencial: o **array**



- ▶ Guarda elementos em posições contíguas de memória
- ▶ Algumas **vantagens**:
 - ★ **Fácil** de usar
 - ★ **Acesso rápido** a uma dada posição
- ▶ Algumas **desvantagens**:
 - ★ **Tamanho é fixo** na criação do array (aumentar implica copiar elementos)
 - ★ **Inserções e remoções** podem ser custosas (muitos *shifts* de elementos)

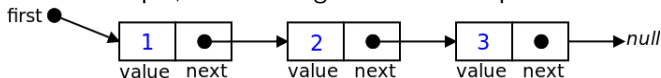
Listas Ligadas

- Precisamos de uma estrutura de dados que **possa "crescer"** o quanto quisermos e que seja **boa para inserir e remover** elementos
- Uma solução: **listas ligadas**
 - ▶ Cada **"nó"** da sequência contém não só o **valor** nessa posição, mas também uma referência para o **próximo** elemento da sequência
 - ▶ Precisamos de saber onde fica o **primeiro** elemento da lista
 - ▶ O último elemento aponta para o **"final"** da lista (vamos usar **null** - o objecto nulo - para denotar o final da lista)



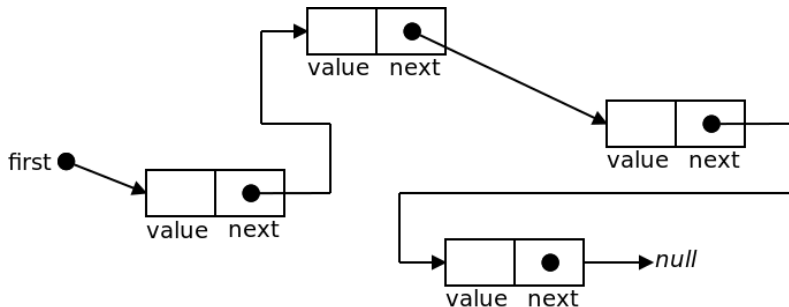
No atributo valor guardamos o "conteúdo" da sequência.

Por exemplo, uma lista ligada de inteiros poderia ser:



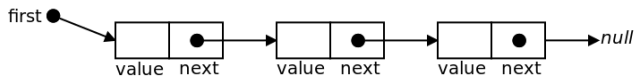
Listas Ligadas - Representação em memória

- As posições de memória da lista não têm de ser contíguas (porque cada nó "aponta" para o seguinte)



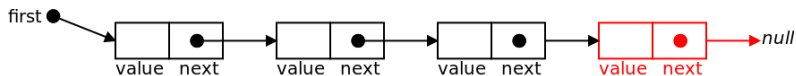
Listas Ligadas - Operações

- A listas facilitam operações como a **adição no final**:



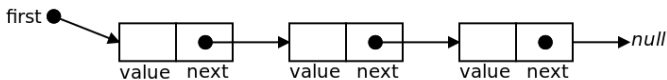
**adicionar
no final**

A single node with a red dot in the 'next' field, representing a new node to be added.

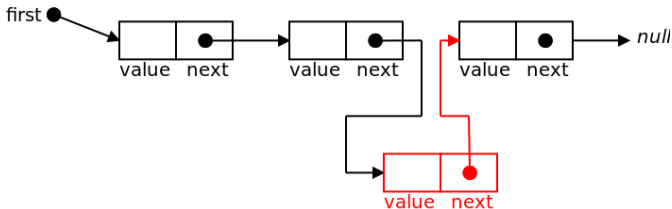


Listas Ligadas - Operações

- Inserção numa qualquer posição:



**inserir numa
qualquer posição**

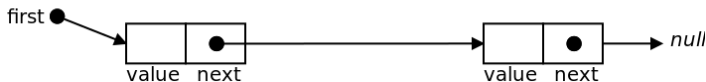


Listas Ligadas - Operações

- Remoção de um elemento:



apagar 



Listas Ligadas - Implementação com "live coding"

- Nas aula teórica fiz alguns dos métodos "ao vivo" (ver o vídeo) para que pudessem acompanhar e ver como surge o código e o que vai acontecendo.
- Nos slides está disponível uma implementação "completa" (o código está disponível no website)

Listas Ligadas - Implementação de um Nó

- Usaremos **genéricos**, para suportar qualquer tipo:
 - ▶ Pode ser uma lista de inteiros, ou uma lista de caracteres, ou uma lista de strings, ou uma lista de qualquer tipo
- Um nó da lista ficará na classe `Node<T>` que tem atributos:
 - ▶ `T value` → o valor a guardar no nó
 - ▶ `Node<T> next` → referência para o nó seguinte da lista

```
public class Node<T> {
    private T value;           // Valor guardado no nó
    private Node<T> next;    // Referência para o próximo nó da lista

    // Construtor
    Node(T v, Node<T> n) {
        value = v;
        next = n;
    }

    // Getters e Setters
    public T getValue() { return value; }
    public Node<T> getNext() { return next; }
    public void setValue(T v) { value=v; }
    public void setNext(Node<T> n) { next = n; }
}
```

Listas Ligadas - Implementação

- A lista em si ficará numa classe chamada `SinglyLinkedList<T>`
 - ▶ `Singly` para indicar que é uma lista ligada simples (vamos falar também de listas duplamente ligadas e de listas circulares)
 - ▶ Isto permite também distinguir da `LinkedList<T>` do próprio Java
- A lista precisa de uma referência para o primeiro nó da lista. Vamos também guardar o tamanho da lista. Os atributos são portanto:
 - ▶ `Node<T> first`
 - ▶ `int size`
- Métodos padrão para os quais vou dar implementação feita são:
 - ▶ `int size()` - devolve número de elementos na lista
 - ▶ `boolean isEmpty()` - devolve `true` se a lista está vazia, `false` caso contrário
 - ▶ `void addFirst(T value)` - adiciona um elemento ao início da lista
 - ▶ `void addLast(T value)` - adiciona um elemento ao final da da lista
 - ▶ `T getFirst()` - devolve o primeiro elemento da lista
 - ▶ `T getLast()` - devolve o último elemento da lista
 - ▶ `void removeFirst()` - remove o primeiro elemento da lista
 - ▶ `void removeLast()` - remove o último elemento da lista
 - ▶ `String toString()` - representação em `String` (para impressão)

Listas Ligadas - Implementação

- A declaração da classe e do construtor padrão é simples e intuitiva
 - ▶ O tamanho de uma lista vazia é zero
 - ▶ O elemento inicial é *null*
 - ▶ Vamos colocar os atributos como privados, seguindo a política de que só "abrimos" o que for mesmo necessário

```
public class SinglyLinkedList<T> {  
    private Node<T> first;    // Primeiro nó da lista  
    private int size;        // Tamanho da lista  
  
    // Construtor (cria lista vazia)  
    SinglyLinkedList() {  
        first = null;  
        size = 0;  
    }  
  
    // (...)  
}
```

- Nos próximos slides vêm métodos a incluir dentro desta classe

Listas Ligadas - Implementação

- O método para devolver o tamanho é simplesmente um *getter*:

```
// Retorna o tamanho da lista
public int size() {
    return size;
}
```

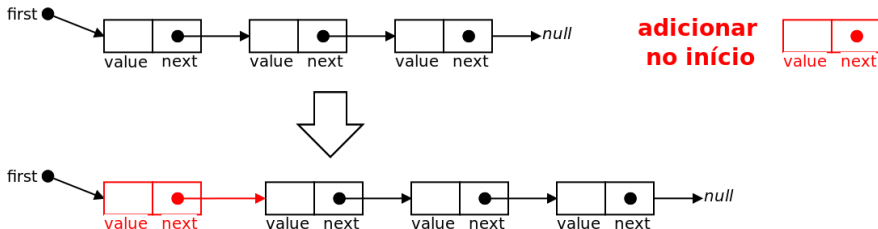
- Para verificar se uma lista está vazia, basta ver se o tamanho é zero:

```
// Devolve true se a lista estiver vazia ou falso caso contrário
public boolean isEmpty() {
    return (size == 0);
}
```

Listas Ligadas - Implementação

- Adicionar um elemento no início resume-se a:
 - ▶ Criar um novo nó (`newNode`) com o valor desejado e o atributo `next` a apontar para o anterior primeiro
 - ▶ Dizer que esse novo nó passa a ser o primeiro da lista
 - ▶ Não esquecer de aumentar o tamanho da lista

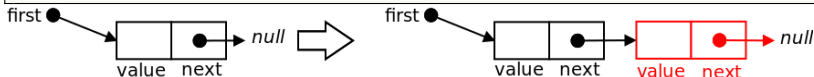
```
// Adiciona v ao início da lista
public void addFirst(T v) {
    Node<T> newNode = new Node<T>(v, first);
    first = newNode;
    size++;
}
```



Listas Ligadas - Implementação

- Adicionar um elemento no final é mais complicado:
 - ▶ É preciso percorrer a lista até chegar ao final (*poderia ser evitado se tivessemos sempre referência para o último*)
 - ▶ É necessário parar "antes" do último para poder mexer no next respectivo (*não temos referência para o anterior*)
 - ▶ Temos de ter cuidado com a lista vazia (caso excepcional)

```
// Adiciona v ao final da lista
public void addLast(T v) {
    Node<T> newNode = new Node<T>(v, null);
    if (isEmpty()) {
        first = newNode;
    } else {
        Node<T> cur = first;
        while (cur.getNext() != null)
            cur = cur.getNext();
        cur.setNext(newNode);
    }
    size++;
}
```



Listas Ligadas - Implementação

- Devolver o primeiro valor é trivial:
(só temos de ter cuidado com o exceção da lista vazia)

```
// Retorna o primeiro valor da lista (ou null se a lista for vazia)
public T getFirst() {
    if (isEmpty()) return null;
    return first.getValue();
}
```

- Devolver o último valor não é muito complicado, mas implica percorrer a lista:

```
// Retorna o último valor da lista (ou null se a lista for vazia)
public T getLast() {
    if (isEmpty()) return null;
    Node<T> cur = first;
    while (cur.getNext() != null)
        cur = cur.getNext();
    return cur.getValue();
}
```

Listas Ligadas - Implementação

- Remover o primeiro elemento é trivial:

```
// Remove o primeiro elemento da lista (se for vazia não faz nada)
public void removeFirst() {
    if (isEmpty()) return;
    first = first.getNext();
    size--;
}
```

- Remover o último é mais complicado, porque temos de parar antes (passa também a ser exceção o caso da lista só ter um elemento):

```
// Remove o último elemento da lista (se for vazia não faz nada)
public void removeLast() {
    if (isEmpty()) return;
    if (size == 1) { first = null; }
    else { // Ciclo com for e size para mostrar alternativa ao while
        Node<T> cur = first;
        for (int i=0; i<size-2; i++) cur = cur.getNext();
        cur.setNext(cur.getNext().getNext());
        // Também se poderia fazer simplesmente cur.setNext(null);
    }
    size--;
}
```


Listas Ligadas - Implementação

- Criar uma String com os elementos delimitada por chavetas não é muito complicado e basta ir concatenando enquanto se percorre a lista
- Cuidado para colocar vírgulas somente entre elementos (o *if* adiciona sempre uma vírgula depois excepto no caso do último elemento)

```
// Converte a lista para uma String
public String toString() {
    String str = "{";
    Node<T> cur = first;
    while (cur != null) {
        str += cur.getValue();
        cur = cur.getNext();
        if (cur != null) str += ",";
    }
    str += "}";
    return str;
}
```

Dados Circulares

- Tipicamente listas guardam sequência de um **início** até a um **fim**
- Em algumas aplicações, os dados podem mais naturalmente ser vistos como tendo **ordem cíclica**: cada elemento tem um anterior e um seguinte, mas não existe um princípio ou fim bem definido.
- Exemplos:
 - ▶ **Jogo por turnos**: joga jogador A, depois jogador B, depois jogador C e por aí adiante até voltar ao jogador A e continuar (ex: jogos de cartas)

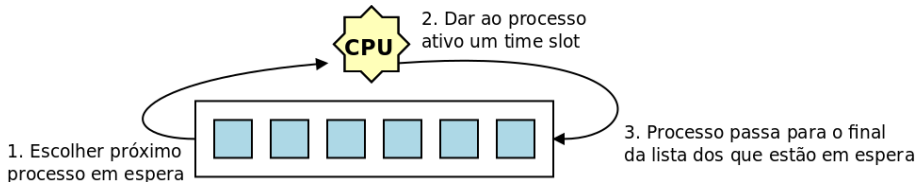


- ▶ **Rotas de Transportes**: algumas rotas de transportes são circulares (ex: metro numa sequência definida, mas em *loop* contínuo)



Escalonamento Round-Robin

- **Multitasking**: um sistema operativo moderno (SO) tem a capacidade de ter vários processos a serem executados "*em simultâneo*"
- Como suportar tantos processos quantos os necessários?
- SOs permite partilha de tempo (*time-sharing*) do CPU tipicamente usando um algoritmo de escalonamento do tipo **round-robin**.
 - ▶ Cada processo recebe um pequeno intervalo de tempo (*time slot*)
 - ▶ No final desse intervalo desse *time slot* o processo é parado (mesmo que não tenha terminado)
 - ▶ Outro processo vai ficando activo e recebendo à vez um *time slot* seguindo uma **ordem cíclica**.
 - ▶ Novos processos podem ir sendo adicionados e os processos que terminam vão sendo removidos

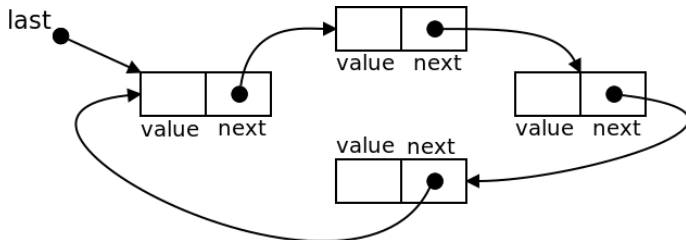


Implementação de Round-Robin

- Como implementar um algoritmo de round-robin?
- Uma hipótese seria usar uma lista ligada simples (`SinglyLinkedList<Process> list`) e repetidamente fazer o seguinte:
 - ▶ `Process p = list.getFirst(); list.removeFirst()`
 - ▶ Dar um *time slot* ao processo *p*
 - ▶ `list.addLast(p);`
- Não é o mais "natural": obriga a ir retirando e depois novamente adicionar um nó (e mexer em atributos como *size*)
- Vamos mostrar como uma pequena alteração à nossa lista ligada tornaria tudo mais simples
(*mostrando também a vantagem de conhecer as estruturas de dados e poder adaptá-las às nossas necessidades*)

Listas Ligadas Circulares

- Uma **Lista Ligada Circular** é essencialmente uma lista ligada onde o último nó "aponta" para o primeiro
- Vamos implementá-la: `CircularLinkedList<T>`
 - ▶ Os nós são iguais aos da lista ligada simples: `Node<T>`
 - ▶ Também para mostrar uma variante, ao invés de uma referência para o primeiro, vamos manter apenas uma referência para o último (**last**)
(*o primeiro é sempre o seguinte ao último*)



Listas Ligadas Circulares - Implementação

- Construtor, size() e isEmpty() são essencialmente os mesmos:

```
public class CircularLinkedList<T> {
    private Node<T> last; // Último nó da lista
    private int size;     // Tamanho da lista

    // Construtor (cria lista vazia)
    CircularLinkedList() {
        last = null;
        size = 0;
    }

    // Retorna o tamanho da lista
    public int size() {
        return size;
    }

    // Devolve true se a lista estiver vazia ou falso caso contrário
    public boolean isEmpty() {
        return (size == 0);
    }

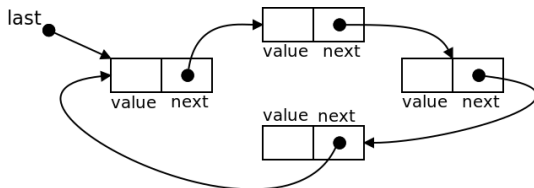
    // (...) O resto dos métodos vêm nos slides a seguir
}
```

Listas Ligadas Circulares - Implementação

- `getFirst()` e `getLast()` ficam triviais:

```
// Retorna o primeiro valor da lista (ou null se a lista for vazia)
public T getFirst() {
    if (isEmpty()) return null;
    return last.getNext().getValue();
}

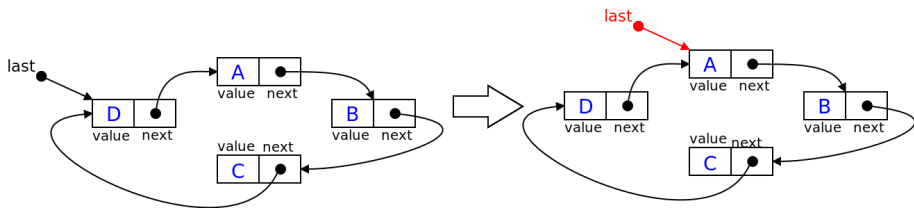
// Retorna o último valor da lista (ou null se a lista for vazia)
public T getLast() {
    if (isEmpty()) return null;
    return last.getValue();
}
}
```



Listas Ligadas Circulares - Implementação

- Para poder aplicar *round-robin* vamos implementar um método `rotate()` que essencialmente avança um elemento na lista
- Chamando repetidas vezes `rotate()` temos o desejado *round-robin*!

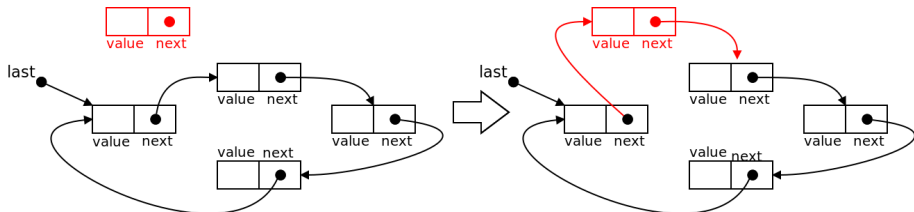
```
// Roda a lista (o primeiro passa a ser o novo ultimo da lista)
public void rotate() {
    if (!isEmpty()) // Se estiver vazia não faz nada
        last = last.getNext();
}
```



Listas Ligadas Circulares - Implementação

- Adicionar um elemento no início (exceção: lista vazia)

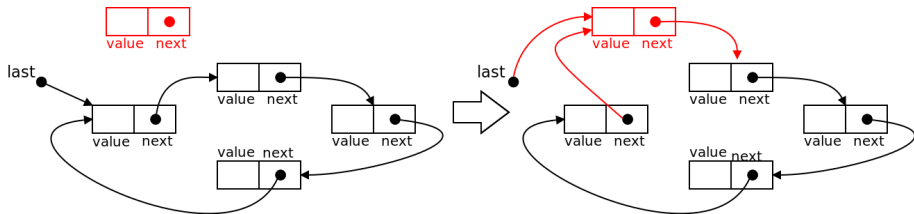
```
// Adiciona v ao início da lista
public void addFirst(T v) {
    if (isEmpty()) {
        last = new Node<T>(v, null);
        last.setNext(last); // Apontar para si próprio em "loop"
    } else {
        Node<T> newNode = new Node<T>(v, last.getNext());
        last.setNext(newNode);
    }
    size++;
}
```



Listas Ligadas Circulares - Implementação

- Adicionar no final pode ser feito... usando o `addFirst`

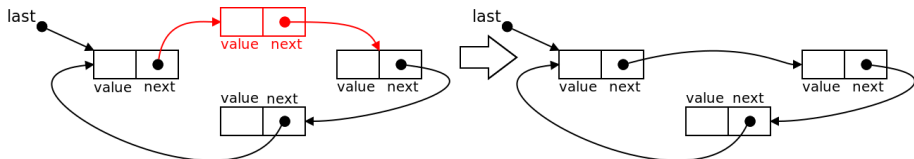
```
// Adiciona v ao final da lista
public void addLast(T v) {
    addFirst(v);
    last = last.getNext();
}
```



Listas Ligadas Circulares - Implementação

- Para remover o primeiro, basta actualizar para quem o último nó aponta:

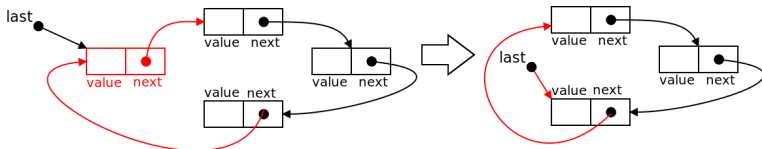
```
// Remove o primeiro elemento da lista (se for vazia não faz nada)
public void removeFirst() {
    if (isEmpty()) return;
    if (size == 1) last = null;
    else last.setNext(last.getNext().getNext());
    size--;
}
```



Listas Ligadas Circulares - Implementação

- Remover o último é mais complicado porque temos de conseguir chegar à penúltima posição (para a actualizar)

```
// Remove o último elemento da lista (se for vazia não faz nada)
public void removeLast() {
    if (isEmpty()) return;
    if (size == 1) {
        last = null;
    } else {
        Node<T> cur = last.getNext();
        for (int i=0; i<size-2; i++)
            cur = cur.getNext();
        last = cur;
        last.setNext(last.getNext().getNext());
    }
    size--;
}
```



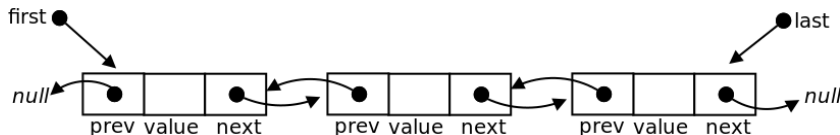
Listas Ligadas Circulares - Implementação

- Para converter para String é só percorrer a lista (e ter cuidado com as vírgulas)

```
// Converte a lista para uma String
public String toString() {
    String str = "{";
    if (!isEmpty()) {
        Node<T> cur = last.getNext();
        for (int i=0; i<size; i++) {
            str += cur.getValue();
            if (cur != last) str += ",";
            cur = cur.getNext();
        }
    }
    str += "}";
    return str;
}
```

Listas Duplamente Ligadas

- Um dos problemas com as listas ligadas anteriores (simples e circular) é que não conseguem **remover o último** de forma eficiente
 - ▶ Para remover o último precisamos de mudar o *next* do penúltimo
 - ▶ Para chegar ao penúltimo precisamos.. de **percorrer a lista**
- Uma maneira de "resolver" isto é ter em cada nó uma referência para o... **anterior** (para além da referência para o **próximo**)!
- É este o conceito de **listas duplamente ligadas**:



Listas Duplamente Ligadas - Implementação

- Os nós têm portanto de ter também referência para o anterior:

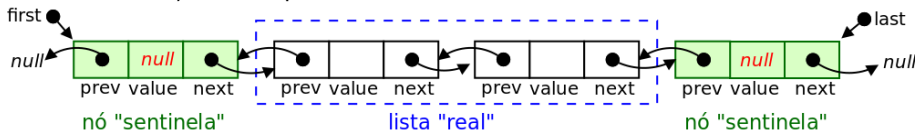
```
public class DNode<T> {
    private T value;           // Valor guardado no nó
    private DNode<T> prev;    // Referência para o nó anterior da lista
    private DNode<T> next;    // Referência para o próximo nó da lista

    // Construtor
    DNode(T v, DNode<T> p, DNode<T> n) {
        value = v;
        prev = p;
        next = n;
    }

    // Getters e Setters
    public T getValue() { return value; }
    public DNode<T> getPrev() { return prev; }
    public DNode<T> getNext() { return next; }
    public void setValue(T v) { value=v; }
    public void setPrev(DNode<T> p) { prev = p; }
    public void setNext(DNode<T> n) { next = n; }
}
```

Listas Duplamente Ligadas - Implementação

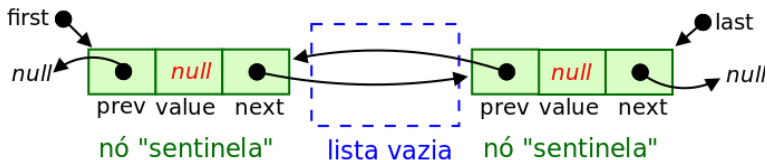
- Não vamos discutir nos slides toda a implementação
- Uma implementação possível (`DoublyLinkedList<T>`) está disponível em: <http://www.dcc.fc.up.pt/~pribeiro/aulas/edados2021/codigo/>
- Vamos apenas abordar uma técnica muito útil usada na implementação
- Por vezes quando existem muitos casos "excepcionais", é mais "fácil" criar **sentinelas**: dados "fictícios" / "dummy" que nos permitem deixar de ter esses casos limite.
- Vamos adicionar dois nós sentinelas: um no **início** e outro no **fim**.
Com isto:
 - ▶ Casos excepcionais da lista "vazia" ou tamanho 1 nunca acontecem
 - ▶ Nunca temos de adicionar ao verdadeiro "início" ou "fim": qualquer inserção é sempre no meio de dois nós



Listas Duplamente Ligadas - Implementação

- No início basta criar a lista já com os dois nós sentinelas:

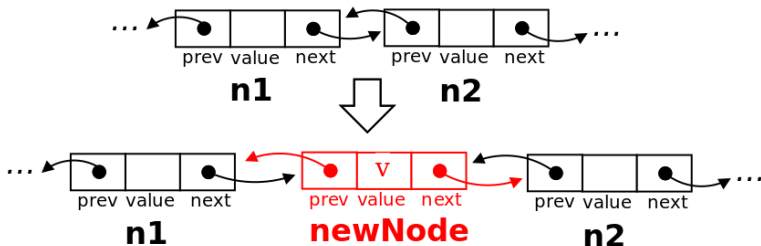
```
// Construtor (cria lista vazia com dois nós sentinelas)
DoublyLinkedList() {
    first = new DNode<T>(null, null, null);
    last  = new DNode<T>(null, first, null); // Antes do último vem o 1º
    first.setNext(last); // A seguir ao primeiro vem o último
    size = 0;
}
```



Listas Duplamente Ligadas - Implementação

- Qualquer **inserção** passa a ser agora **inserir entre dois nós**
 - ▶ Nunca inserimos antes do primeiro nó sentinela
 - ▶ Nunca inserimos depois do segundo nó sentinela
- Código para inserir entre dois nós:

```
// Adiciona elemento entre dois nós n1 e n2
private void addBetween(T v, DNode<T> n1, DNode<T> n2) {
    DNode<T> newNode = new DNode<T>(v, n1, n2);
    n1.setNext(newNode);
    n2.setPrev(newNode);
    size++;
}
```



Listas Duplamente Ligadas - Implementação

- Para inserir no início ou no fim é só chamar o **addBetween**:
 - ▶ **addFirst** é inserir entre `first` e `first.next`
 - ▶ **addLast** é inserir entre `last.prev` e `last`

```
// Adiciona v ao início da lista
public void addFirst(T v) {
    addBetween(v, first, first.getNext());
}

// Adiciona v ao final da lista
public void addLast(T v) {
    addBetween(v, last.getPrev(), last);
}
```

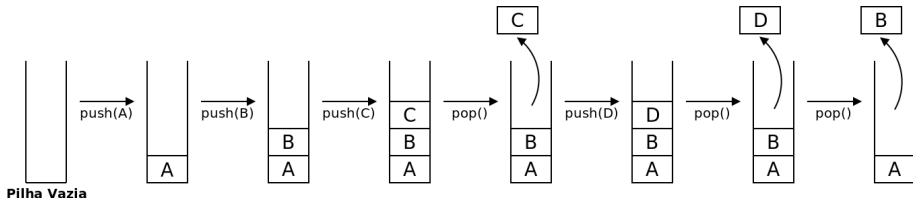
- Podem ver o resto da implementação no código disponibilizado. Ex:
 - ▶ Para remover um nó n é só colocar o $n.prev$ a apontar para $n.next$ (e vice-versa)

Tipos Abstractos de Dados (TADs) sequenciais

- Listas ligadas e arrays são exemplos de **estruturas de dados "primárias"** (cada uma com as suas vantagens e desvantagens)
- Subindo um pouco o **nível de abstração**, vamos agora abordar **TADs sequenciais**: para que servem e como usá-las.
- Vamos falar de 3 TADs diferentes:
 - ▶ **Pilhas (Stacks)**
 - ▶ **Filas (Queues)**
 - ▶ **Filas com 2 extremos (Dequeus - Double Ended Queues)**
- Estes TADs podem ser implementados de diferentes maneiras (ex: usando arrays ou listas ligadas)
- Estes TADs vão ser mais tarde "*legos básicos*" para outros algoritmos que vão aprender nesta e noutras unidades curriculares. Por exemplo:
 - ▶ O algoritmo de **pesquisa em largura** num grafo usa a noção de *fila*
 - ▶ O algoritmo de Tarjan para descobrir **componentes fortemente conexos** usa a noção de *pilha*

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 é 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 atribuída 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:

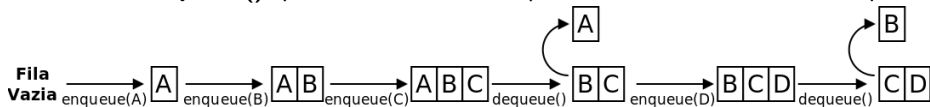
```
MyStack<Integer> s = new ArrayStack<Integer>();
```

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 fila simula precisamente uma "fila de objectos", como uma fila de atendimento ao público num supermercado ou num banco



Filas - Interface MyQueue

- Para além do `enqueue(x)` e `dequeue()` 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 início da fila  
  
    // Métodos que acedem a informação (sem modificar)  
    public T first();        // Retorna valor no início 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());
    }
}
```

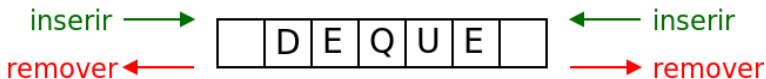
Deque - Interface

- 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 início 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 início 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  
}
```

Deque - 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`



Classes do Java

- Nas últimas aulas temos implementado as nossas próprias classes de listas ligadas e TADS como as pilhas e filas
- A linguagem Java também tem disponíveis estas estruturas de dados:
 - ▶ classe `LinkedList<T>`:
<https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html>
 - ▶ classe `Stack<T>`:
<https://docs.oracle.com/javase/8/docs/api/java/util/Stack.html>
 - ▶ interface `List<T>`:
<https://docs.oracle.com/javase/8/docs/api/java/util/List.html>
 - ▶ interface `Queue<T>`:
<https://docs.oracle.com/javase/8/docs/api/java/util/Queue.html>
 - ▶ interface `Deque<T>`:
<https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html>
- A documentação diz classes que implementam cada interface.
Exemplo:
 - ▶ `Deque` é implementado, entre outros, por `LinkedList` e `ArrayDeque`
 - ▶ `List` é implementado, entre outros, por `LinkedList` e `ArrayList` e `Vector`

Exemplo de uso das classes do Java

- Um exemplo de uso de um Deque do Java

```
import java.util.*; // Incluir todas as classes do package java.util

class TestDeque {
    public static void main(String[] args) {
        Deque<Integer> d = new LinkedList<Integer>();

        d.addLast(1);
        d.addLast(2);
        d.addFirst(3);
        d.addFirst(4);
        System.out.println(d);
    }
}
```

Sobre o uso de iteradores

- Algumas classes de sequências de Java implementam o interface `Iterable<T>`
- Isto significa que são "percorríveis", e disponibilizam iteradores que assentam nos seguintes dois métodos:
 - ▶ `hasNext()`: devolve *true* se ainda existe outro elemento na sequência, ou *false* caso contrário
 - ▶ `next()`: devolve o próximo elemento na sequência
- A combinação destes dois métodos permitiria uma maneira genérica de percorrer todos os elementos. Imagine por exemplo uma instância *iter* de `Iterator<String>`. Pode ser percorrida usando algo como:

```
while (iter.hasNext()) {  
    String value = iter.next();  
    System.out.println(value);  
}
```

Sobre o uso de iteradores: foreach

- Este padrão é tão comum que o Java disponibiliza um sintaxe de ciclo para facilitar a maneira como percorremos uma sequência iterável (um "foreach"):

```
for (ElementType variable : sequence) {  
    LoopInstructions  
}
```

- Um exemplo com listas ligadas, supondo que *list* é uma `LinkedList<Integer>`:

```
LinkedList<Integer> list = new LinkedList<Integer>();  
  
// (...) instruções para colocar elementos na lista  
  
for (int value : list) {  
    System.out.println(value);  
}
```