

Aula 10

Pilhas, Filas e Listas Biligadas

Programação II, 2019-2020

v1.3, 2019-05-14

DETI, Universidade de Aveiro

10.1

Objectivos:

- Saber implementar e utilizar estruturas tipo “pilha”.
- Saber implementar e utilizar estruturas tipo “fila”.
- Saber implementar e utilizar estruturas tipo “lista biligada”.

Conteúdo

1 Pilhas e filas	1
1.1 Definições e tipos de dados abstratos	1
1.2 Implementação em lista ligada	3
1.3 Implementação em vector	6
2 Listas biligadas	10
3 Comparação entre diferentes tipos de listas ligadas	12

10.2

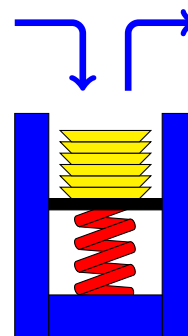
1 Pilhas e filas

1.1 Definições e tipos de dados abstratos

Pilha: definição

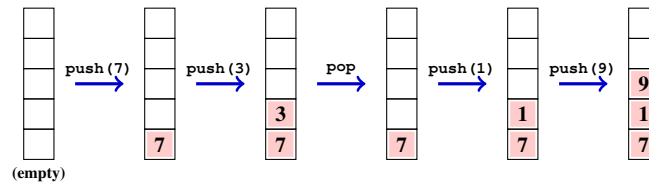
- É uma estrutura de dados sequencial que só pode ser modificada por uma das suas extremidades usualmente denominada como “topo”.

- Estrutura com gestão *LIFO*: *Last In First Out*;
 - O último elemento a entrar é o primeiro a sair.



10.3

Pilha: as operações `push` / `pop`



10.4

Pilha: exemplos de utilização

- Armazenamento de contextos de execução de subrotinas.
- Análise e avaliação de expressões matemáticas.
- Travessia *depth-first* de árvores e grafos.
- Detecção de marcas de início/fim em texto formatado. Por exemplo, parênteses ou marcas HTML ou XML.
- ...

10.5

Pilha: tipo de dados abstrato

- Nome do módulo:
 - `Stack`
- Serviços:
 - `push`: insere (empilha) um elemento no topo da pilha
 - `pop`: remove (desempilha) o elemento no topo da pilha
 - `top`: devolve o elemento no topo da pilha
 - `isEmpty`: verifica se a pilha está vazia
 - `isFull`: verifica se a pilha está cheia
 - `size`: retorna a dimensão actual da pilha
 - `clear`: limpa a pilha (retira todos os elementos)

10.6

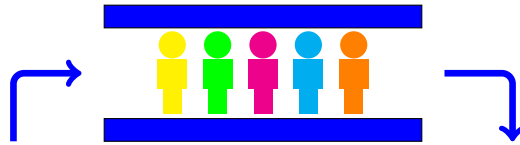
Pilha: semântica

- **`push(e)`**
 - Pré-condição: `!isFull()`
 - Pós-condição: `!isEmpty() && (top() == e)`
- **`pop()`**
 - Pré-condição: `!isEmpty()`
 - Pós-condição: `!isFull()`
- **`top()`**
 - Pré-condição: `!isEmpty()`

10.7

Fila: definição

- É uma estrutura de dados cujo acesso é feito por ambas as extremidades:
 - uma apenas para colocar elementos, e a outra apenas para os retirar.



- Gerida segundo uma política *FIFO* (*First In First Out*)
 - extrai-se sempre o valor mais antigo primeiro.

10.8

Fila: tipo de dados abstrato

- Nome do módulo:
 - Queue
- Serviços:
 - in: insere um elemento no fim da fila
 - out: retira elemento do início da fila
 - peek: retorna o elemento do início da fila
 - isEmpty: verifica se a fila está vazia
 - isFull: verifica se a fila está cheia
 - size: retorna a dimensão actual da fila
 - clear: limpa a fila (retira todos os elementos)

10.9

Fila: semântica

- **in(v)**
 - Pré-condição: `!isFull()`
 - Pós-condição: `!isEmpty()`
- **out()**
 - Pré-condição: `!isEmpty()`
 - Pós-condição: `!isFull()`
- **peek()**
 - Pré-condição: `!isEmpty()`

10.10

1.2 Implementação em lista ligada

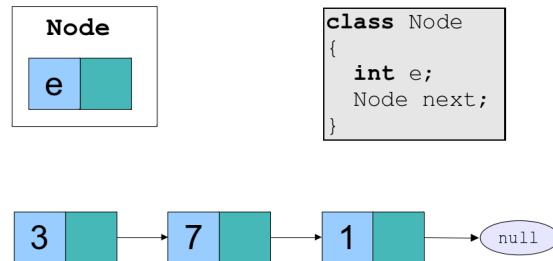
Implementação em lista ligada

- Numa aula anterior, estudámos as listas ligadas.
- Comparando com os vectores, vimos que:
 - A grande vantagem das listas ligadas é serem estruturas de dados dinâmicas, portanto sem limitação na sua capacidade.
 - A grande desvantagem das listas ligadas é não facilitarem o acesso direto a cada elemento.
- No caso particular das pilhas e das filas:

- Pode ser difícil prever o número de elementos.
- Não há necessidade de aceder a elementos abaixo do topo da pilha.
- Não há necessidade de aceder a elementos no meio da fila.
- Assim, em geral, a implementação de pilhas e filas em lista ligada é vantajosa, quando comparada com a implementação em vector.

10.11

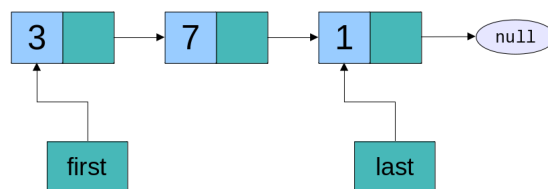
Relembrando: lista ligada simples



10.12

Relembrando: lista ligada com dupla entrada

- A lista possui acesso direto ao primeiro e último elementos.
- É simples acrescentar elementos no início e no fim da lista.
- É simples remover elementos do início da lista.



10.13

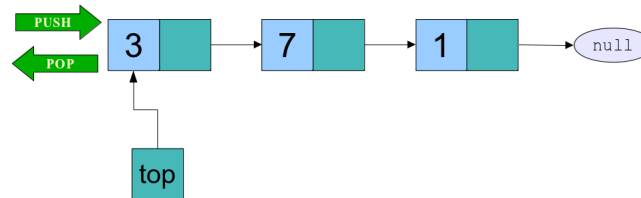
Relembrando: lista ligada - tipo de dados abstrato

- Nome do módulo:
 - LinkedList
- Serviços:
 - addFirst: insere um elemento no início da lista
 - addLast: insere um elemento no fim da lista
 - first: retorna o primeiro elemento da lista
 - last: retorna o último elemento lista
 - removeFirst: retira o elemento no início da lista
 - size: retorna a dimensão actual da lista
 - isEmpty: verifica se a lista está vazia
 - clear: limpa a lista (remove todos os elementos)

10.14

Pilha: implementação em lista ligada

- Usa uma gestão *LIFO* (*Last In First Out*)
- O último elemento empilhado (*top*) é o primeiro a desempilhar.
 - Método *push* corresponde a *addFirst* da lista ligada.
 - Método *pop* corresponde a *removeFirst* da lista ligada.
- O elemento no topo da pilha fica armazenado no primeiro nó da lista.
- O elemento na base da pilha fica armazenado no último nó da lista.



10.15

Pilha genérica: implementação em lista ligada

```
public class Stack<E> {

    private LinkedList<E> list = new LinkedList<E>();

    public void push(E element) {
        list.addFirst(element);
    }

    public E top() {
        return list.first();
    }

    public void pop() {
        list.removeFirst();
    }

    public int size() {
        return list.size();
    }

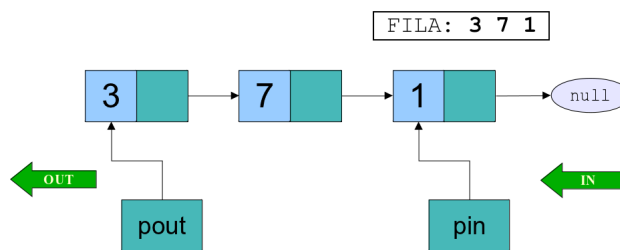
    public boolean isEmpty() {
        return list.isEmpty();
    }

}
```

10.16

Fila: implementação em lista ligada

- Usa uma gestão *FIFO* (*First In First Out*).
- O primeiro elemento introduzido é o primeiro a remover, por isso tem que ficar no primeiro nó da lista.
 - Método *out* corresponde a *removeFirst* da lista ligada.
- O último elemento introduzido fica armazenado no último nó da lista e será o último a ser removido.
 - Método *in* corresponde a *addLast* da lista ligada.



Fila genérica: implementação em lista ligada

```
public class Queue<E> {
    private LinkedList<E> list = new LinkedList<E>();

    public void in(E element) {
        list.addLast(element);
    }

    public E peek() {
        return list.first();
    }

    public void out() {
        list.removeFirst();
    }

    public int size() {
        return list.size();
    }

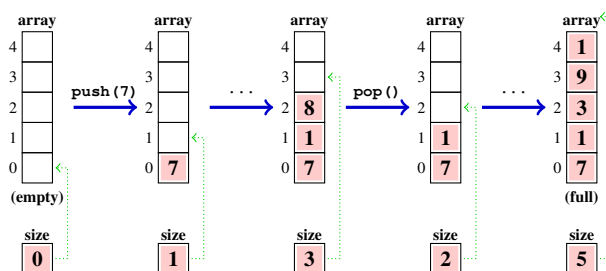
    public boolean isEmpty() {
        return list.isEmpty();
    }
}
```

1.3 Implementação em vector

Para certas aplicações específicas de pilhas ou filas, o número máximo de elementos pode ser conhecido à partida. Nesses casos, a implementação em vector (array) pode ser preferível, porque evita perder tempo com a alocação e libertação dinâmica de memória.

Pilha: implementação em vector

- Precisamos de dois atributos:
 - O vector que armazena os elementos
 - O número de elementos, que funciona também como índice da primeira posição livre



Pilha genérica: implementação em vector

```

public class Stack<E> {
    private E[] array;
    private int size;

    public Stack(int maxSize) {
        assert maxSize >= 0;
        array = (E[]) new Object[maxSize];
        size = 0;
    }

    public void push(E e) {
        assert !isFull();
        array[size] = e;
        size++;
        assert !isEmpty() && top() == e;
    }

    public void pop() {
        assert !isEmpty();
        size--;
        assert !isFull();
    }

    public E top() {
        assert !isEmpty();
        return array[size-1];
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public boolean isFull() {
        return size == array.length;
    }

    public int size() {
        return size;
    }

    public void clear() {
        size = 0;
        assert isEmpty();
    }
}

```

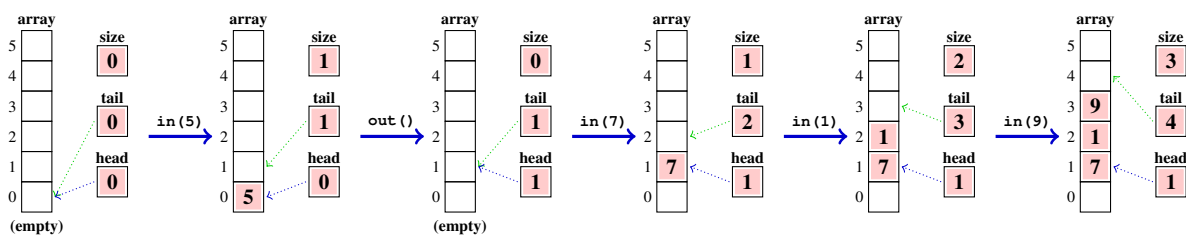
10.20

Fila: implementação em vector

- A forma mais eficiente de implementar é uma estrutura conhecida como *buffer circular*.
- Requer 4 atributos:
 - O vector que armazena os elementos.
 - O número de elementos.
 - O índice do próximo elemento a ser retirado (*cabeça* da fila).
 - O índice do próximo elemento a ser ocupado (*cauda* da fila).
- Sempre que se insere ou retira um elemento, incrementa-se o índice respetivo em *aritmética modular*.
 - Ou seja, quando o índice atinge o limite, é reposto a zero.

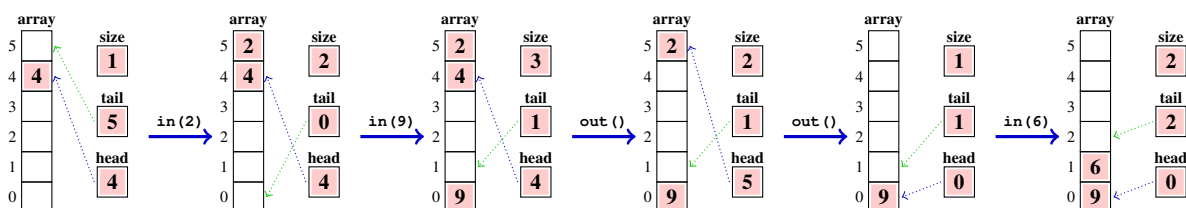
10.21

Fila: exemplo



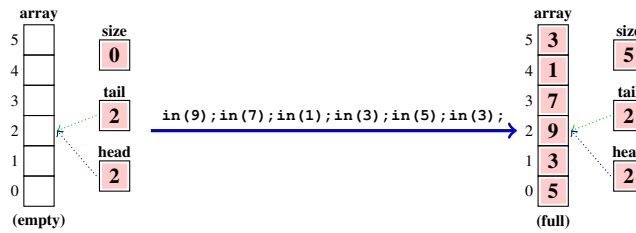
10.22

Fila: exemplo - gestão circular



10.23

Fila: exemplo - empty/full



10.24

Fila genérica: implementação em vector

```
public class Queue<E> {
    private E[] array;
    private int size;
    private int head;
    private int tail;

    public Queue(int maxSize) {
        assert maxSize >= 0;

        array = (T[]) new Object[maxSize];
        size = head = tail = 0;
    }

    public void in(E e) {
        assert !isFull();
        array[tail] = e;
        tail = nextPosition(tail);
        size++;
    }

    public void out() {
        assert !isEmpty();
        head = nextPosition(head);
        size--;
    }

    public E peek() {
        assert !isEmpty();
        return array[head];
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public boolean isFull() {
        return size == array.length;
    }

    public void clear() {
        head = tail = size = 0;
    }

    private int nextPosition(int p) {
        return (p + 1) % array.length;
    }
}
```

10.25

Lista	descrição	Pilha	Fila
addLast	acrescenta um elemento no fim da lista	-	in
addFirst	acrescenta um elemento no início da lista	push	-
first	devolve o primeiro elemento da lista	top	peek
removeFirst	remove o primeiro elemento da lista	pop	out

- Os tipos de dados abstratos das pilhas e filas correspondem a subconjuntos do tipo de dados abstrato da lista ligada.
- Podemos dizer que os tipos de dados abstratos das pilhas e filas são *açúcar sintático* para certos perfis de utilização das listas.

10.26

Pilhas e filas: complexidade

- Implementação em lista ligada:
 - Todos os métodos do tipo de dados abstrato têm complexidade constante ($O(1)$).
- Implementação em vector com dimensão fixa:
 - Todos os métodos do tipo de dados abstrato têm complexidade constante ($O(1)$).
- Implementação em vector com re-dimensionamento:

- Sempre que a pilha ou fila enche, temos que criar um novo vector e transferir a informação para esse vector.
- Nesses casos, a operação `push` passa a ter complexidade linear ($O(n)$).
- Os restantes métodos do tipo de dados abstrato têm complexidade constante ($O(1)$).

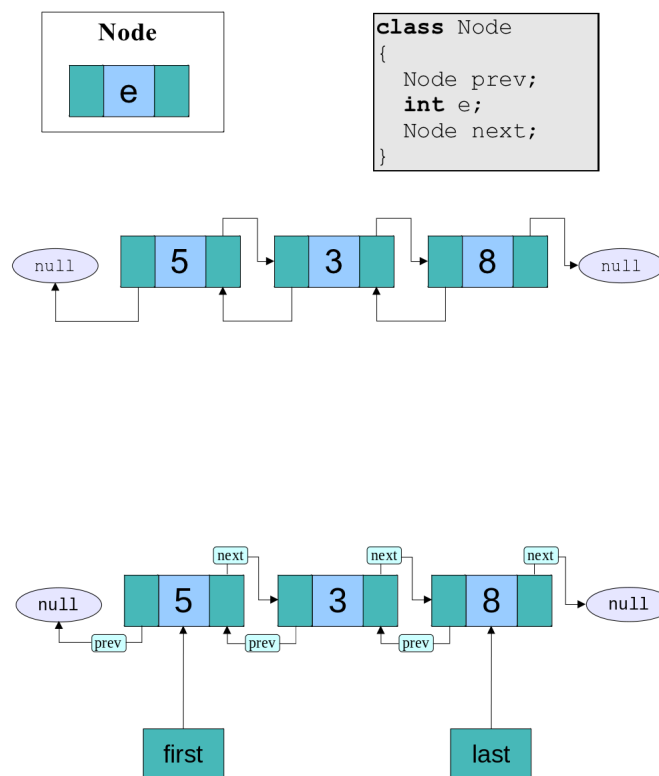
2 Listas biligadas

Lista biligada

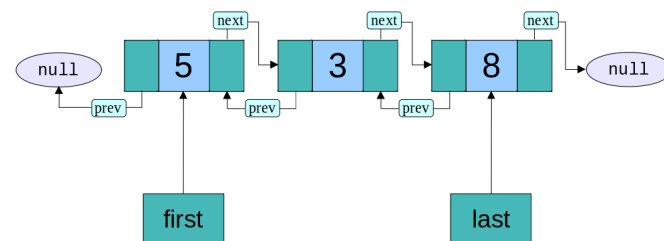
- Estrutura de dados sequencial em que cada elemento da lista contém uma referência para o próximo elemento e outra para o anterior.
 - Cada uma dessas referências terá o valor `null` caso o elemento a que se refere não exista.
- Ao contrário da lista ligada, permite um acesso sequencial do fim para o início.
- Facilita a remoção do último elemento (`removeLast`).

10.28

Lista biligada: nós e ligações



10.29



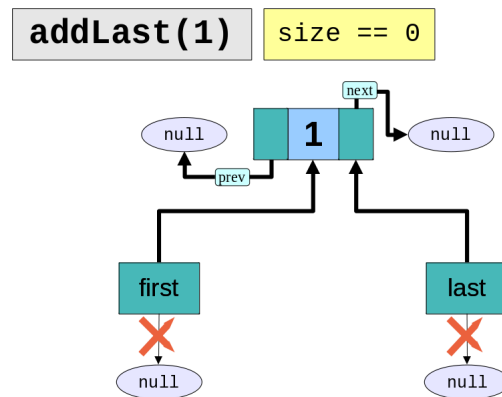
10.30

Lista biligada: tipo de dados abstrato

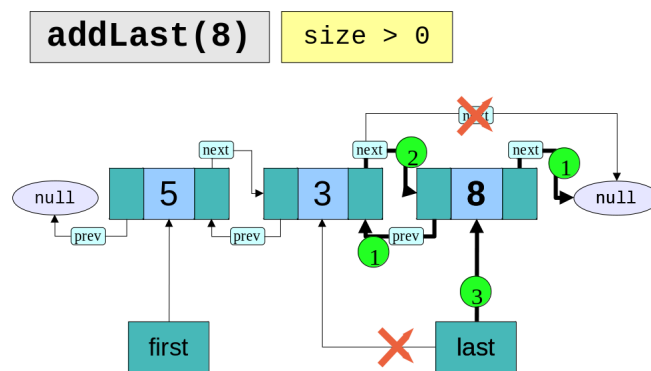
- Nome do módulo:
 - `LinkedList`
- Serviços:
 - `addFirst`: insere um elemento no início da lista.
 - `addLast`: insere um elemento no fim da lista.
 - `first`: devolve o primeiro elemento da lista.
 - `last`: devolve o último elemento lista.
 - `removeFirst`: retira o elemento no início da lista.
 - `removeLast`: retira o elemento no início da lista.
 - `size`: devolve a dimensão actual da lista.

- isEmpty: verifica se a lista está vazia.
- clear: limpa a lista (remove todos os elementos).

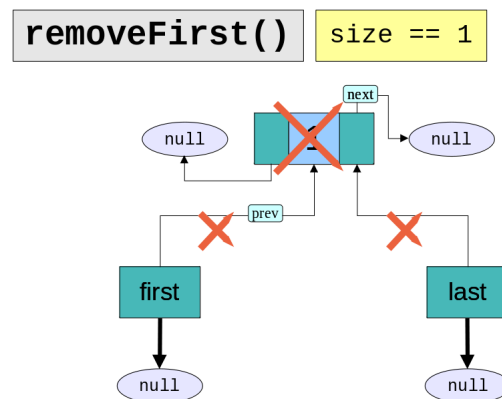
10.31



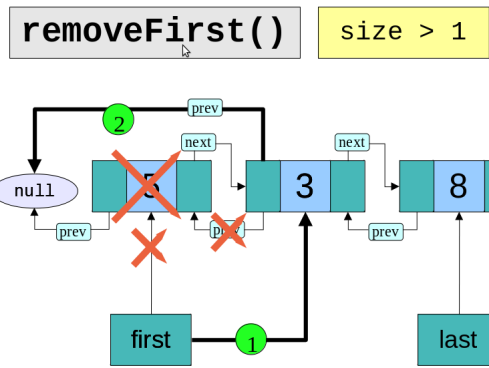
10.32



10.33



10.34



10.35

3 Comparação entre diferentes tipos de listas ligadas

Tipo de Lista	Simple	Simple	Circular Simple	Biligada	Circular Biligada
Atributos	first	first last	last	first last	first (last)
Operações					
insert first	O(1)	O(1)	O(1)	O(1)	O(1)
remove first	O(1)	O(1)	O(1)	O(1)	O(1)
insert last	O(n)	O(1)	O(1)	O(1)	O(1)
remove last	O(n)	O(n)	O(n)	O(1)	O(1)
scan forward	O(n)	O(n)	O(n)	O(n)	O(n)
scan backward	O(n²)	O(n²)	O(n²)	O(n)	O(n)
insert middle	O(n)	O(n)	O(n)	O(n)	O(n)
remove middle	O(n)	O(n)	O(n)	O(n)	O(n)

10.36