

Aula 6

Coleções

Pilhas e Filas

Algoritmos e Estruturas de Dados

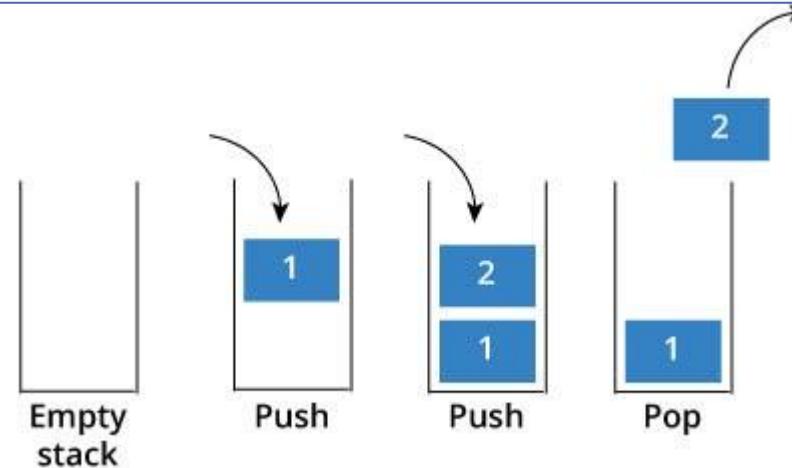
Pilha (Stack)

Implementação como Lista Ligada

Especificação Interface Pilha

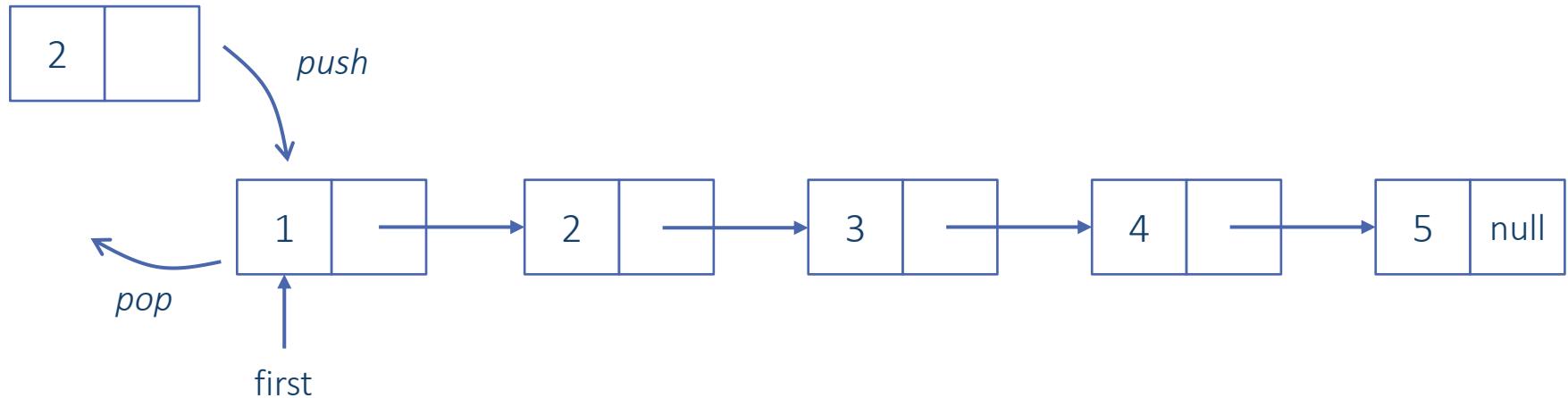
```
public interface IStack<Item> extends Iterable<Item>
```

void	push (Item item)	coloca um item no topo da pilha
Item	pop ()	remove e retorna o item no topo da pilha. Caso a pilha esteja vazia deverá retornar <i>null</i>
Item	peek ()	retorna o item no topo da pilha, mas não o remove. Retorna null caso a pilha esteja vazia.
boolean	isEmpty ()	retorna <i>true</i> se a pilha estiver vazia e <i>false</i> caso contrário
Int	size ()	devolve o tamanho (número de elementos) da pilha
Istack<Item>	shallowCopy ()	retorna uma cópia superficial da pilha. Uma cópia superficial copia a estrutura da pilha sem copiar cada item individualmente.



Pilha – implementação como lista

- Podemos usar uma lista simples como pilha
 - Topo da pilha é o ínicio da lista



Implementação

```
public interface IStack<Item> extends Iterable<Item>
```

void	push (Item item)	coloca um item no topo da pilha
Item	pop ()	remove e retorna o item no topo da pilha. Caso a pilha esteja vazia deverá retornar <i>null</i>
Item	peek ()	retorna o item no topo da pilha, mas não o remove. Retorna <i>null</i> caso a pilha esteja vazia.
boolean	isEmpty ()	retorna <i>true</i> se a pilha estiver vazia e <i>false</i> caso contrário
Int	size ()	devolve o tamanho (número de elementos) da pilha
Istack<Item>	shallowCopy ()	retorna uma cópia superficial da pilha. Uma cópia superficial copia a estrutura da pilha sem copiar cada item individualmente.

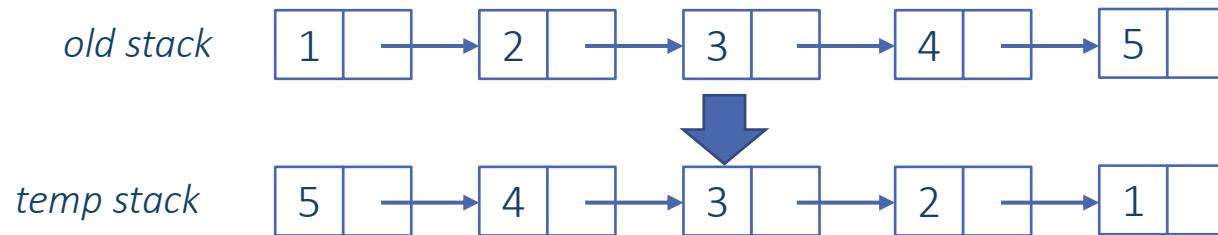
- *push, pop, size, isEmpty*
 - ver métodos correspondentes lista
 - verificações adicionais para casos da lista vazia
- *peek* – implementação trivial

Fila – shallow copy

- 3 formas de executar uma shallow copy sobre Listas Ligadas

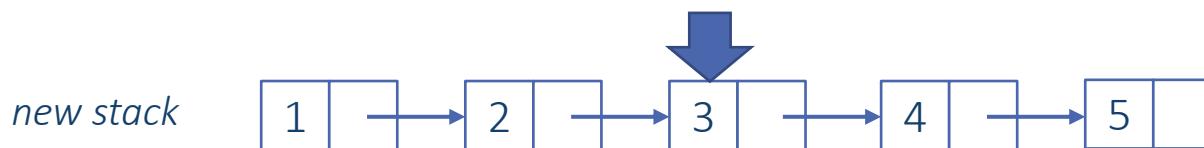
1. Criar uma nova stack

Iterar a stack para copiar os elementos da stack para a nova stack



Problema: elementos ficam com a ordem trocada

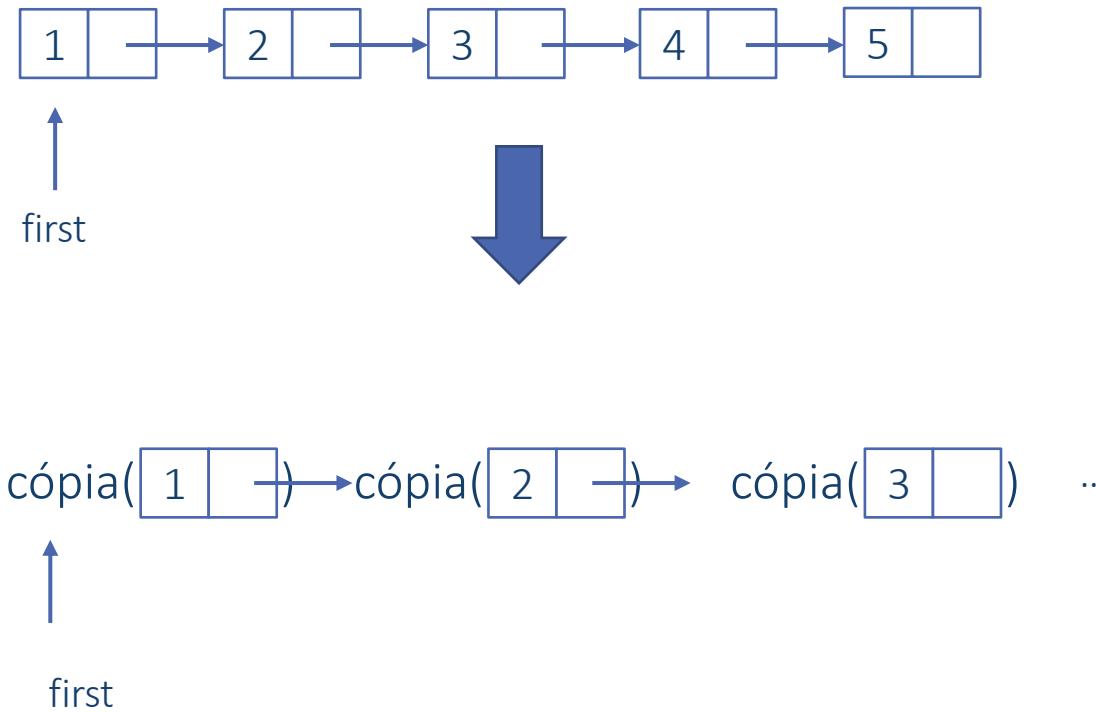
Temos de repetir o processo para voltar a colocar pela ordem certa



FÁCIL de implementar, mas INEFICIENTE

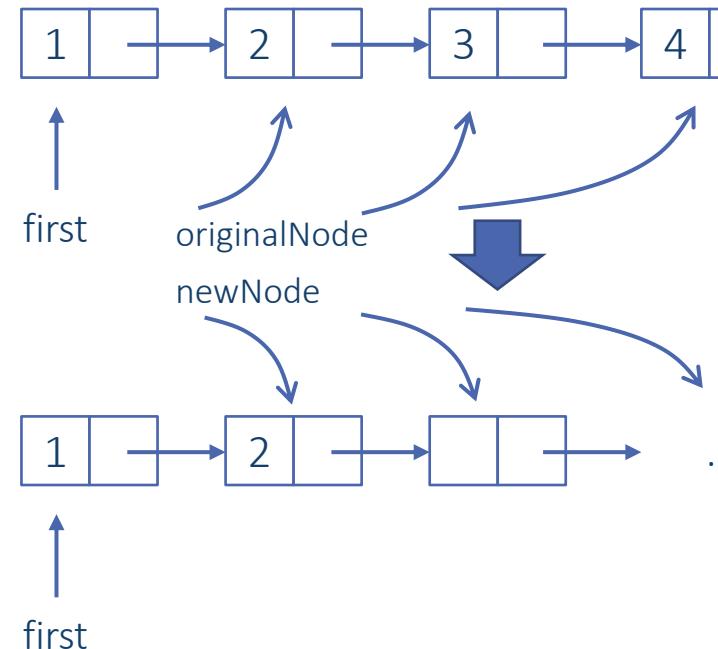
Fila – shallow copy

- 3 formas de executar uma shallow copy sobre Listas Ligadas
- 2. CÓPIA recursiva dos nós que compõem a lista



Fila – shallow copy

- 3 formas de executar uma *shallow copy* sobre Listas Ligadas
3. Cópia através de ciclo *for* que vai criando novos nós
(eficiente, mas + difícil)



Temos que trabalhar com 2 ponteiros para nós

Usar um ciclo for avançar nos dois ponteiros de forma sincronizada

Ligaçāo para próximo nō requer criação antecipada do nō

- Complexidade temporal

Worst case	
construtor	$O(1)$
push	$O(1)$
pop	$O(1)$
size	$O(1)$

- Complexidade espacial

- 40 bytes por nó
- $40N$ bytes



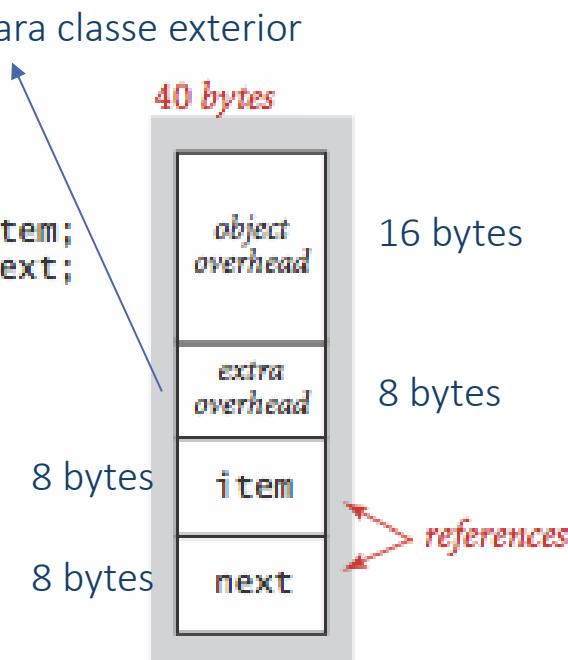
é possível reduzir para **32 bytes** se a classe nó não for inner class

- $32N$ bytes

```

referência para classe exterior
node object (inner class)
public class Node
{
    private Item item;
    private Node next;
    ...
}

```



Pilha: *Lista* vs *Array*

- Complexidade temporal

	List	Array	
	worst	worst	amortized
construtor	O(1)	O(1)	O(1)
push	O(1)	O(N)	O(1)
pop	O(1)	O(N)	O(1)
size	O(1)	O(1)	O(1)

- Complexidade espacial

List	Array
$32N$	$8N - 32N$

Filas (Queue)

Especificação

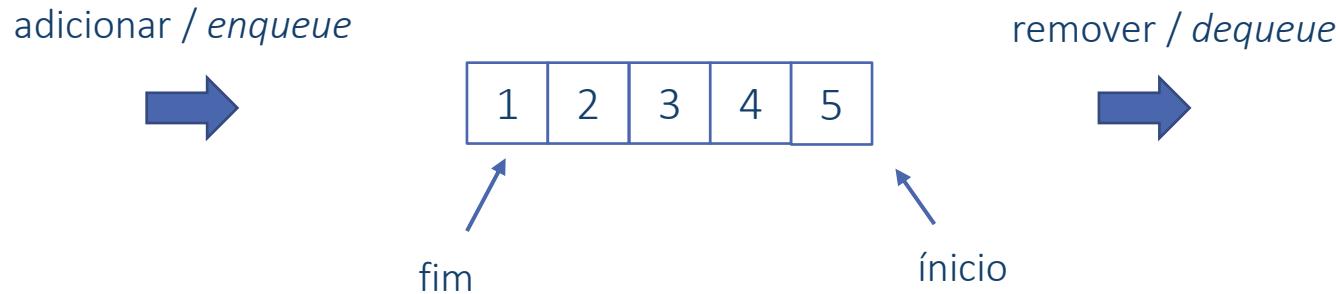
Filas (Queues)

- Colecção em que só se pode aceder ao ínicio da mesma
 - Mas novos items são colocados no fim da fila
 - Respeita a ordem de chegada na fila
 - First In First Out (FIFO)



Filas (Queues)

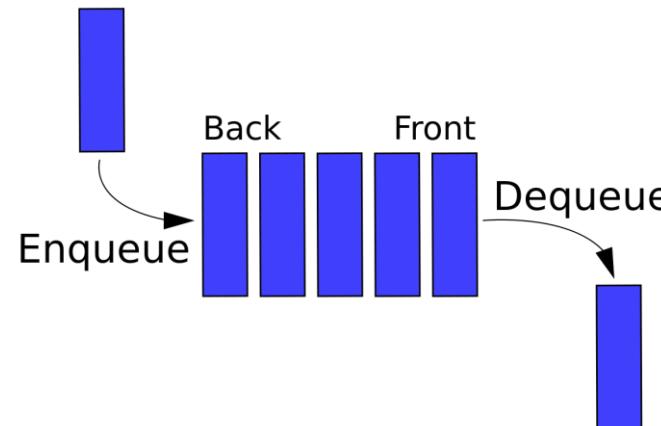
- Colecção em que só se pode aceder ao inicio da mesma
 - Mas novos items são colocados no fim da fila
 - Respeita a ordem de chegada na fila
 - First In First Out (FIFO)



Especificação Interface Fila

```
public interface IQueue<Item> extends Iterable<Item>
```

void	enqueue (Item item)	coloca um item no fim da fila
Item	dequeue ()	remove e retorna o item no ínicio da fila.
Item	peek ()	retorna o item no ínicio da fila, mas não o remove.
boolean	isEmpty ()	retorna <i>true</i> se a fila estiver vazia e <i>false</i> caso contrário
Int	size ()	devolve o tamanho (número de elementos) da fila

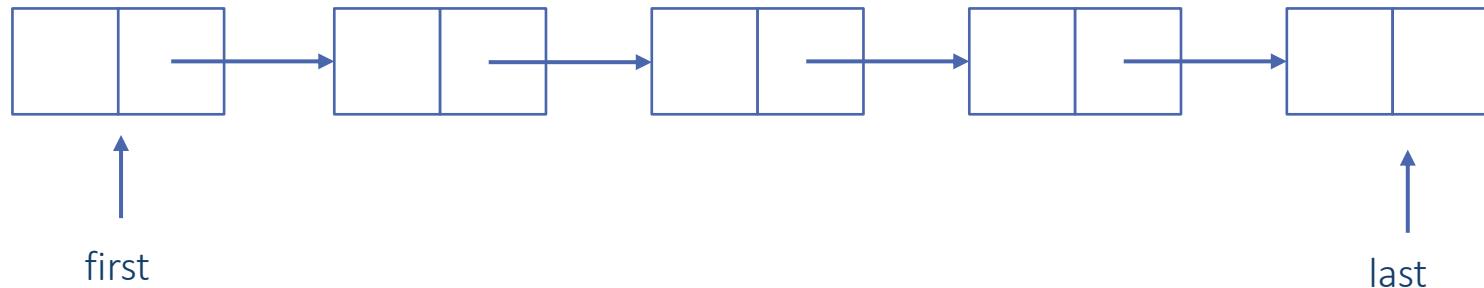
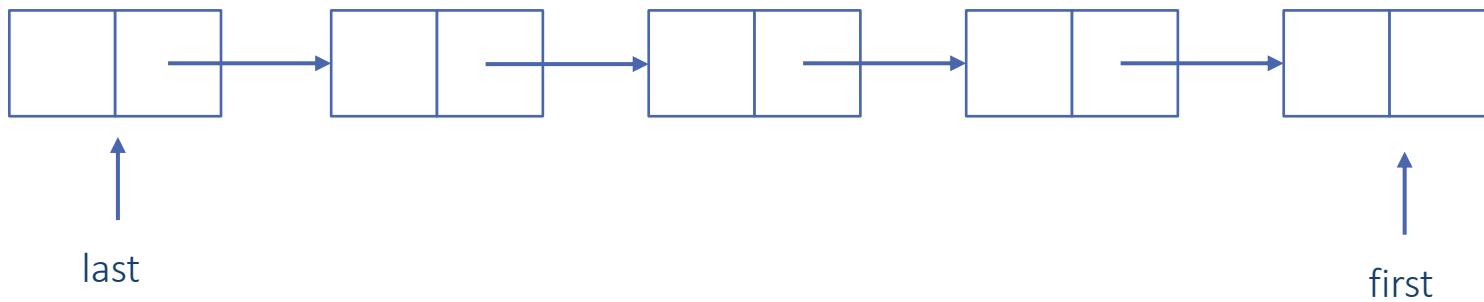


Fila (Queue)

Implementação como lista ligada

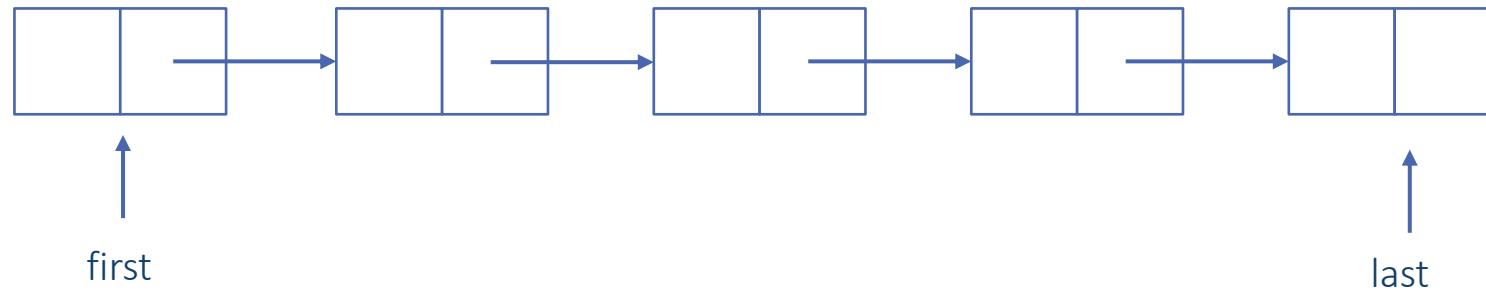
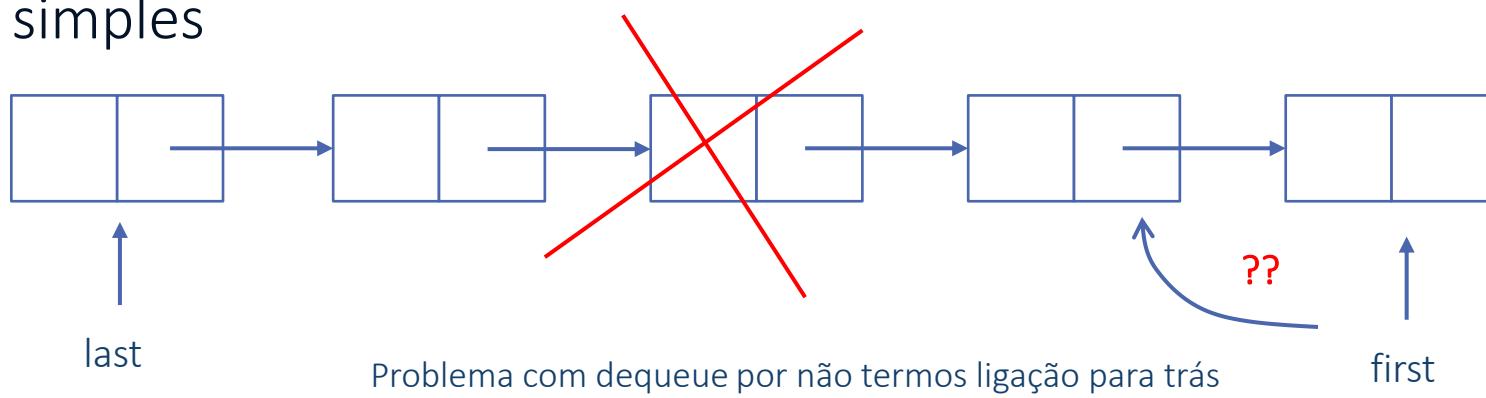
Fila – implementação como Lista

- Uma destas configurações não é eficiente para uma lista simples



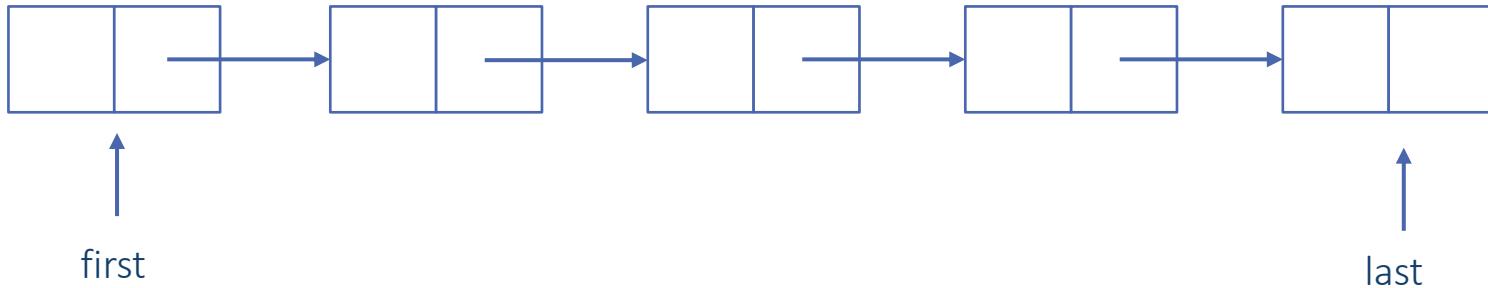
Fila – implementação como Lista

- Uma destas configurações não é eficiente para uma lista simples



Fila – implementação como Lista

- *Dequeue* a partir do primeiro nó
- *Enqueue* depois do último nó



Fila – implementação como lista

```
public class QueueList<T> implements IQueue<T> {
    private class Node
    {
        T item;
        Node next;
        Node(T item, Node next)
        {
            this.item = item;
            this.next = next;
        }
    }
    Node first;
    Node last;
    int size;

    public QueueList()
    {
        this.first = null;
        this.last = null;
        this.size = 0;
    }
}
```

```

public class QueueList<T> implements IQueue<T> {
  private class Node
  {
    T item;
    Node next;
    Node(T item, Node next)
    {
      this.item = item;
      this.next = next;
    }
  }
  Node first;
  Node last;
  int size;

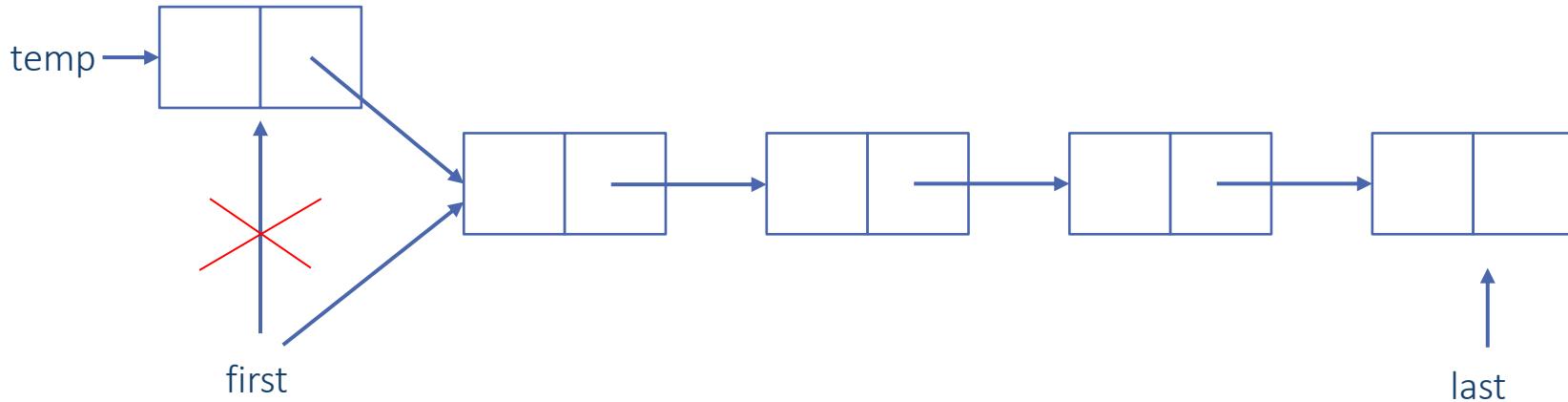
  public QueueList()
  {
    this.first = null;
    this.last = null;
    this.size = 0;
  }
}
  
```

```

3   typedef struct Node
4   {
5     void* item;
6     struct Node* next;
7   } Node;
8
9   typedef struct Queue_list
10  {
11    Node* first;
12    Node* last;
13    int size;
14  } Queue;
15
16 Queue* create_queue_list()
17 {
18   Queue* q = (Queue*) malloc(sizeof(Queue));
19   q->first = NULL;
20   q->start = NULL;
21   q->size = 0;
22   return q;
23 }
  
```

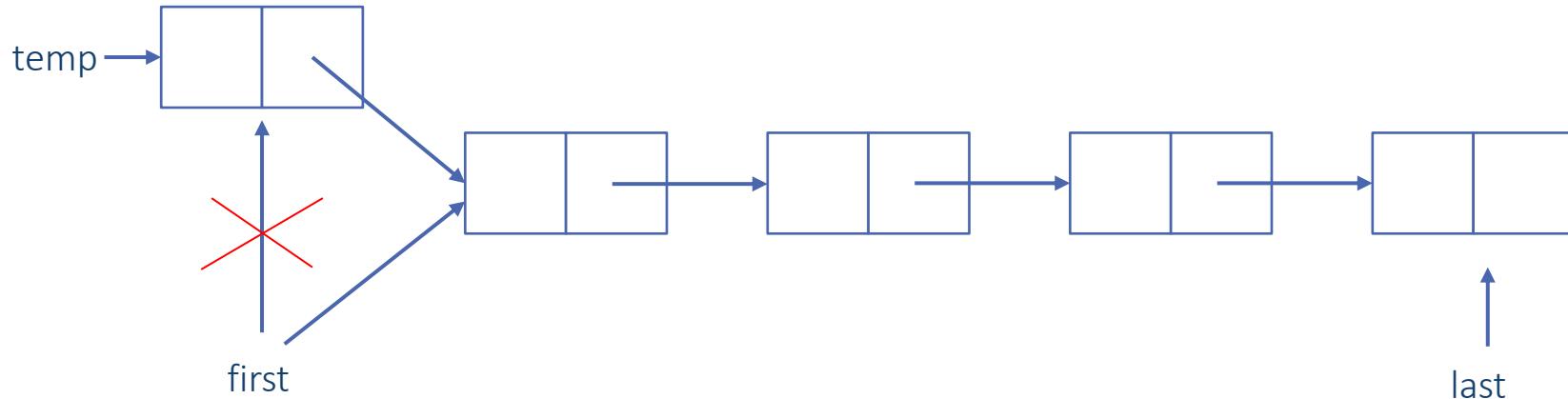
Fila – método *dequeue*

- Guardar cópia do primeiro nó
- Avançar o ponteiro para o próximo nó
- Retornar o item
- Diminuir o tamanho



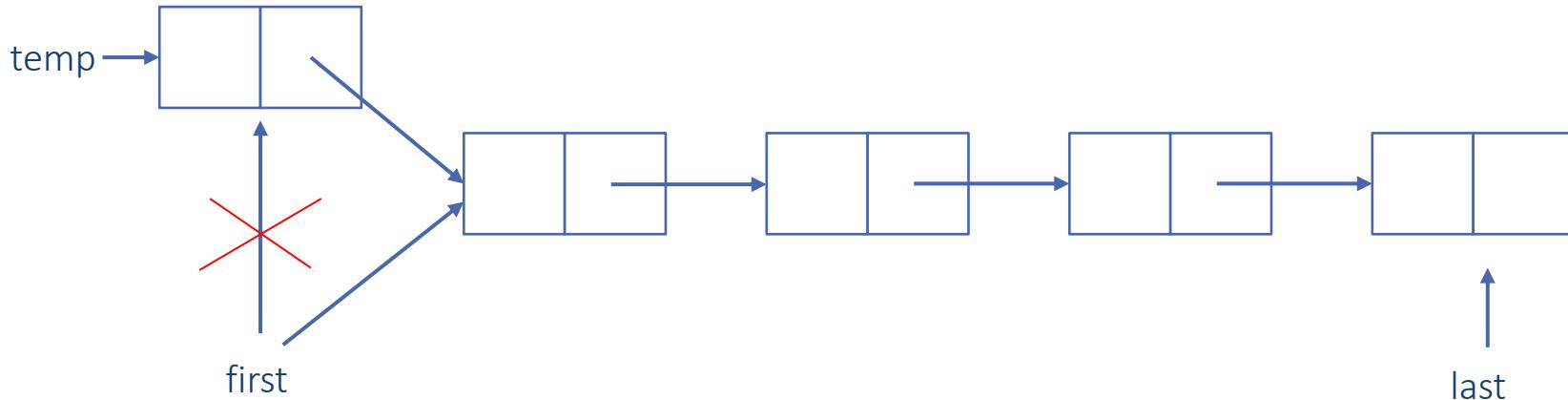
Fila – método *dequeue*

```
public T dequeue()
{
    Node temp = this.first;
    this.first = this.first.next;
    size--;
    return temp.item;
}
```



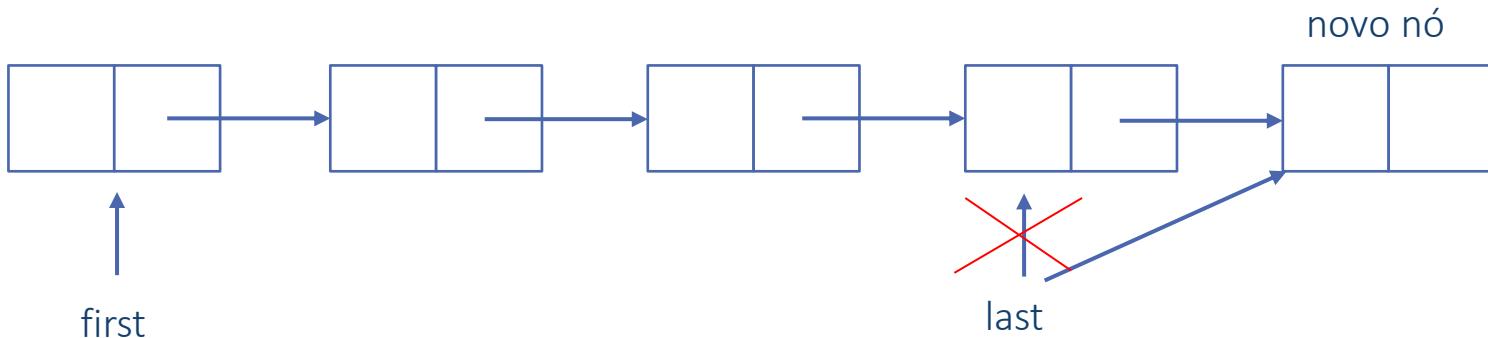
```
public T dequeue()
{
    Node temp = this.first;
    this.first = this.first.next;
    size--;
    return temp.item;
}
```

```
23 void* dequeue(Queue* q)
24 {
25     Node* temp = q->first;
26     void* temp_item = temp->item;
27     q->first = q->first->next;
28     //libertar o espaço do nó apagado
29     free(temp);
30     q->size--;
31     return temp->item;
32 }
```



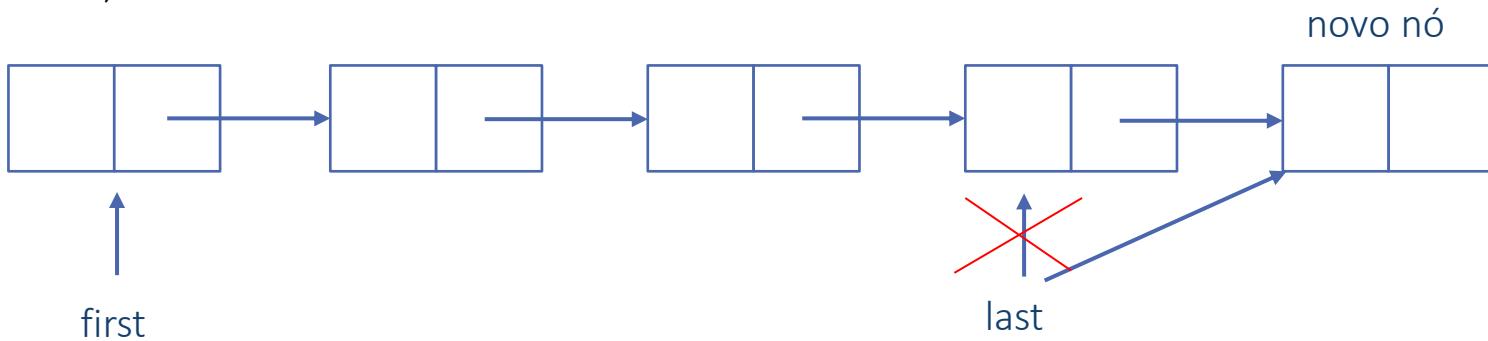
Fila – método enqueue

- Criar novo nó
- Se Fila vazia
 - Apontar *first* e *last* para novo nó
- Se Fila não vazia
 - Colocar último nó a apontar para novo nó
 - Actualizar *last* para novo nó
- Aumentar tamanho



Fila – método enqueue

```
public void enqueue(T item) {  
    Node n = new Node(item,null);  
    if(this.first == null)  
    {  
        this.first = n;  
        this.last = n;  
    }  
    else  
    {  
        this.last.next = n;  
        this.last = n;  
    }  
    size++;  
}
```



```

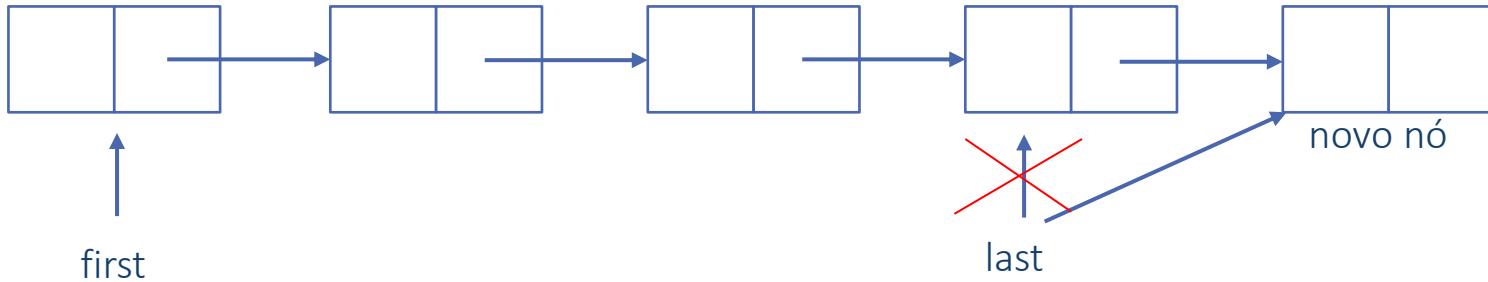
public void enqueue(T item) {
    Node n = new Node(item,null);
    if(this.first == null)
    {
        this.first = n;
        this.last = n;
    }
    else
    {
        this.last.next = n;
        this.last = n;
    }
    size++;
}

```

```

36     void enqueue(Queue* q, void* item)
37     {
38         Node* n = (Node*) malloc(sizeof(Node));
39         n->item = item;
40
41         if(q->first == NULL)
42         {
43             q->first = n;
44             q->last = n;
45         }
46         else
47         {
48             q->last->next = n;
49             q->last = n;
50         }
51
52     }
53 }

```



- Triviais

```
public T peek() {  
    return this.first.item;  
}  
  
public boolean isEmpty() {  
    return size == 0;  
}  
  
public int size() {  
    return this.size;  
}
```

Fila – métodos *iterator* e *iterable*

```
public Iterator<T> iterator() {
    return new QueueIterator();
}

private class QueueIterator implements Iterator<T>
{
    Node iterator;
    QueueIterator()
    {
        this.iterator = first;
    }
    public boolean hasNext() {
        return this.iterator != null;
    }

    public T next() {
        T result = this.iterator.item;
        this.iterator = this.iterator.next;
        return result;
    }

    public void remove() {
        throw new UnsupportedOperationException("Iterator doesn't support removal");
    }
}
```

- Complexidade temporal

Worst	
construtor	$O(1)$
enqueue	$O(1)$
dequeue	$O(1)$
size	$O(1)$

- Complexidade espacial

- 40 bytes por nó
- 40 N bytes



é possível reduzir para **32 bytes** se a classe nó não for inner class

- 32 N bytes

```

referência para classe exterior
node object (inner class)
public class Node
{
    private Item item;
    private Node next;
    ...
}
  
```

