4. Listas, Pilhas, e Filas

Fernando Silva

DCC-FCUP

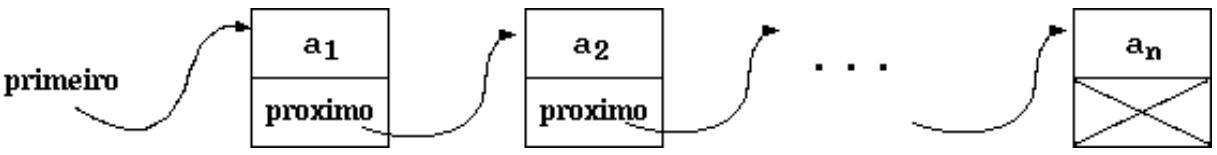
Estruturas de Dados

Definição de Lista (1)

Uma lista é uma sequência finita de elementos ligados entre si

Em que,

- cada elemento (ou nó) da lista tem a seguinte estrutura:
  - um atributo com o valor do elemento, e
  - um atributo com uma referência para o próximo elemento da lista (será nula se for o último elemento).
- a ordem dos elementos na lista é relevante.
- os elementos de uma lista são todos do mesmo tipo.



Definição Recursiva de Lista (2)

Uma lista L é uma sequência finita de elementos em que:

- L ou é uma lista vazia (0 elementos), ou
- L é uma lista composta por um elemento H (cabeça da lista) seguido de um resto de lista RL com os restantes elementos.

Escrito de outra forma (notação do Prolog):

- `L= []` (lista vazia)
- `L= [H|RL]` (cabeça H, seguida do resto de lista RL)

Verificar se `X` é membro de uma lista `L`:

```
member(X,[X|_]).                  % X ou está na cabeça da lista
member(X,[_|RL]):- member(X,RL).% ou está no resto da lista
```


Definição TAD-Lista (3)

Um TAD-lista define-se como uma sequência de elementos:

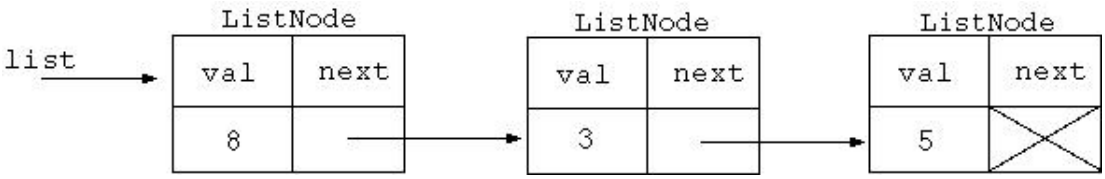
- onde cada elemento é caracterizado por uma estrutura com dois atributos:
  - um valor do elemento corrente (e.g. inteiro, objecto, etc), e
  - uma referência para elemento seguinte.
- e um conjunto de operações a realizar sobre a sequência:

`addFirst(val)`	inserir `val` no início da lista
`addLast(val)`	inserir `val` no fim da lista
`add(val, index)`	inserir `val` na posição `index`
`removeFirst()`	remover o primeiro elemento
`remove(index)`	remover o elemento na posição `index`
`removeLast()`	remover o último elemento
`get(index)`	retornar o elemento na posição `index`
`indexOf(val)`	retorna a posição da 1a ocorrência de `val`
`empty()`	verificar se a lista está vazia

Definição de Lista Ligada (class ListNode)

Comecemos por definir uma classe Java `ListNode` que represente um elemento (nó) de uma lista (e.g. de inteiros):

```
class ListNode {   // nó de uma lista
    int      val;  // valor do elemento
    ListNode next; // referencia para o próximo elemento

    ListNode(int v, ListNode n) {// Construtor de novo nó
       val= v;
       next= n;
    }
}
```



Percorrer Listas Ligadas

Como escrever os elementos da lista?

- começar num elemento da lista (o primeiro)
- enquanto não chegar ao fim da lista
- escrever o elemento corrente
- e avançar para o elemento seguinte

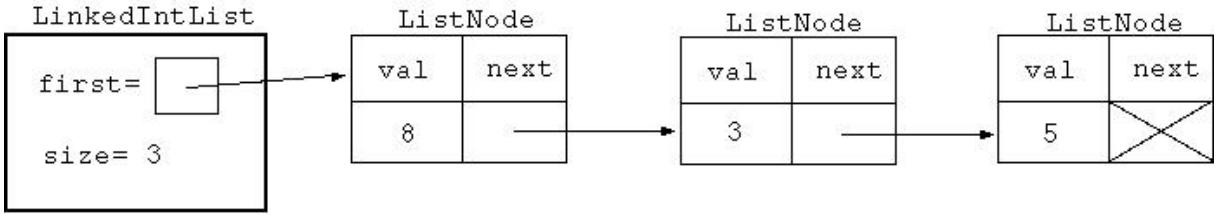
```
// cursor para percorrer a lista
// deve começar no primeiro elemento
ListNode cursor= first;

// chegamos ao fim da lista quando cursor==null
while (cursor!=null) {
    System.out.println(cursor.val);
    cursor= cursor.next; // avançar para o seguinte
}
```

Listas Ligadas (class LinkedIntList)

Na definição de lista é **determinante** saber qual é o primeiro elemento da lista.

Uma lista fica caracterizada por uma classe LinkedIntList com

- `first` - referencia 1º elemento (um `ListNode`).
- `size` - mantém o nº de elementos na lista.
- e os métodos que manipulam os elementos da lista.



Listas Ligadas (class LinkedIntList)

```
class LinkedIntList {       // Lista ligada de valores inteiros
    private ListNode first; // primeiro elemento da lista
    private int      size;  // número de elementos da lista

    LinkedIntList() { // construtor de lista vazia
        first= null;
        size= 0;
    }
    LinkedIntList(LinkedIntList l) { // lista a partir de lista
        first= l.getFirst();
        size= l.size();
    }
    ListNode getFirst() {return first;}
    boolean empty() {return size==0};
    int size() {return size;}

    // métodos que operam sobre a lista
}
```

Métodos que percorrem a lista (print())

Escrever os elementos da lista:

```
void print() {
    // cursor para percorrer a lista
    ListNode cursor= first;

    // chegamos ao fim da lista quando cursor==null
    while (cursor!=null) {
        System.out.println(cursor.val);
        cursor= cursor.next; // avançar para o seguinte
    }
}

void print() {
    for (ListNode cursor= first; cursor!=null; cursor=cursor.next)
        System.out.println(cursor.val);
}
```


Métodos que percorrem a lista: size() e indexOf()

Comprimento da lista: calcular o número de elementos na lista.

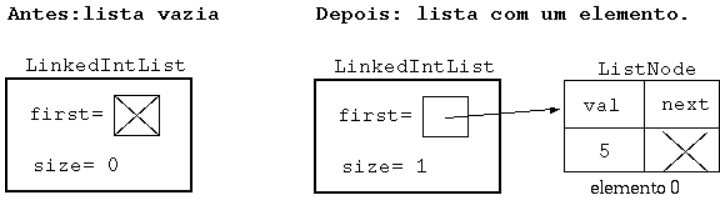
```
int size() {return size;}
// ou
int size2() {
    int ctr= 0;
    for (ListNode cursor= first; cursor!=null; cursor=cursor.next)
        ctr++;
    return ctr;
}
```

Procurar a posição na lista da 1a. ocorrência de v; retornar -1 se v não existir na lista (podia gerar uma excepção).

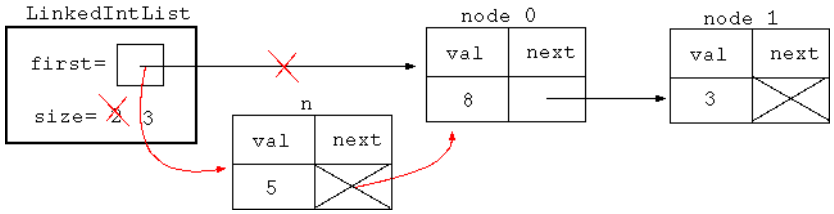
```
int indexOf(int v) { // posição da primeira ocorrência de v
    int index= 0;
    LinkedNode cursor;
    for (cursor= first; cursor!=null && cursor.val!=v; cursor= cursor.next)
        index++;
    if (cursor==null) index= -1; // caso em que v não está na lista
    return index;
}
```

Inserir um elemento numa lista

- Inserir numa lista vazia:



- Inserir à cabeça da lista:

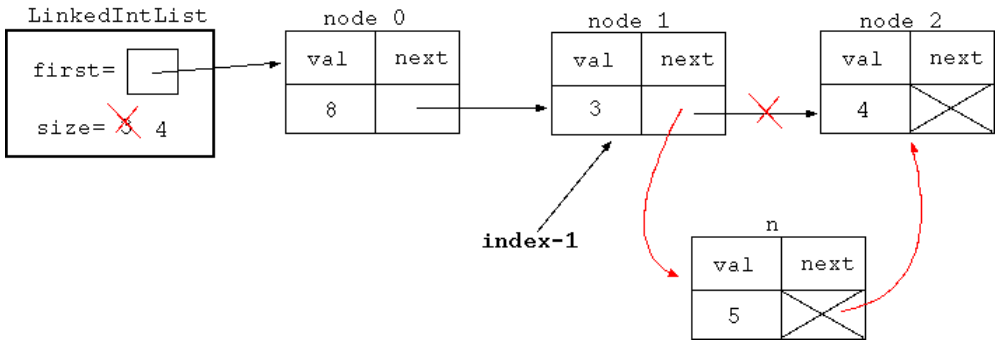


- Implementação:

```
void addFirst(int v) {
    // liga novo nó com o primeiro anterior
    // funciona mesmo quando a lista está vazia
    first= new ListNode(v, first);
    size++;
}
```


Inserir um elemento numa lista

- Inserir na posição dada por `index` (e.g. `l.add(5, 2);`):



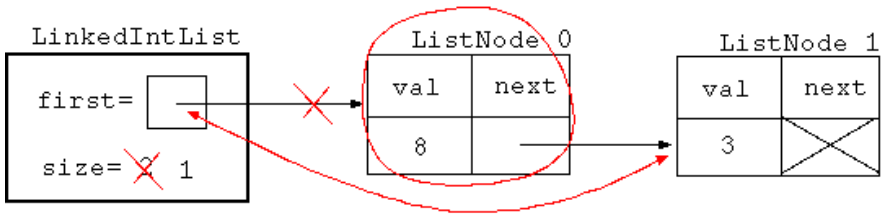
- A posição `index` tem de obdecer a `0 <= index <= size-1`, em que `size` é o número de elementos na lista.
- Se `index==0` então corresponde a inserir no início da lista.

Implementação do método `add()`

```
// inserir valor v na posição index da lista
// precondicao: 0 <= index < size
void add(int v, int index) {
    if (index==0) { //
        // igual a addFirst()
        first= new ListNode(v, first);
    }
    else {
        // primeiro avança até à posição index-1
        ListNode cursor= first;
        for (int i= 0; i< index-1; i++)
            cursor= cursor.next;
        // insere entre cursor e actual cursor.next
        cursor.next = new ListNode(v,cursor.next);
    }
    size++;
}
```


Remover o primeiro elemento da lista

- Remover o primeiro elemento da lista:



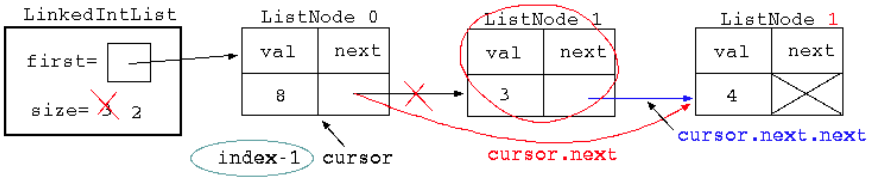
- Implementação:

```
void removeFirst() {
    // se alista for vaiza devia gerar excepção
    // assim, é preciso garantir que o método só é
    // chamado com lista não vazia!
    first= first.next;
    size--;
}
```

- Notas:
  - Não estamos a tratar o caso de a lista poder estar vazia.
  - Não retornamos o elemento devolvido! Deve usar-se o método `get()` antes de remover.

Remover um elemento da lista `remove()`

- Remover da lista o elemento da posição `index`:




```
// remover da lista o elemento da posição index
// pre-condição: 0 <= index < size
void remove(int index) throws EmptyStackException {
    if (empty())
        throw new EmptyStackException(``List is empty!'');
    if (index == 0) {
        // igual a removeFirst()
        first= first.next;
    }
    else {
        // avança até à posição index-1
        ListNode cursor= first;
        for (int i= 0; i< index-1; i++)
            cursor= cursor.next;
        // remove fazendo avançar um elemento
        cursor.next = cursor.next.next;
    }
    size--;
}
```


Obter o valor de um elemento da lista `get()`

- Retornar o valor do elemento na posição `index` da lista:

```
// retorna o valor da lista na posição index
// pré-condição: 0<= index < size
int get(int index) {
    ListNode cursor;
    for (int i= 0; i<index; i++)
        cursor= cursor.next;
    return cursor.val;
}
```

- Notas:
  - Nos vários métodos, por simplicidade, não verificamos situações de excepção!
  - Os métodos `get()` e `remove()` têm de satisfazer $0 \leq index \leq size - 1$ (o que garante que a lista não está vazia)!
  - Os métodos `getFirst()` e `removeFirst()` têm de satisfazer que a lista não está vazia.

Exemplo com a classe LinkedIntList

Programa que manipula objectos da classe `LinkedIntList`:

```
class ExemploLista {
    public static void main(String args[]) {
        LinkedIntList l= new LinkedIntList();

        for (int i= 8; i>0; i--)
            l.addFirst();
        l.add(55,5); // insere o 55 na 5a posição da lista
        System.out.println("Comprimento da lista: " + l.size());
        l.print(); // escreve conteúdo da lista

        l.removeFirst(); // remove o 1° elem. da lista
        l.remove(3);     // remove o 3° elem. da lista

        System.out.println("Comprimento da lista: " + l.size());
        l.print(); // escreve conteúdo da lista
    }
}
```


Listas de objectos genéricos

Em vez de definirmos listas de inteiros podemos ter listas de objectos genéricos, usando `Object`:

```
class Node {
   Object val;
   Node   next;

   Node(Object v, Node n) {
      val= v;
      next= n;
   }
}
class LinkedList {
   Node first;
   int  size;

   // ...
}
```

Veremos mais adiante uma definição mais completa, usando tipos genéricos.

TAD Pilha (Stack)

Uma pilha é um caso especial de uma lista.
Podemos definir uma pilha restringindo as operações sobre o TAD-lista do seguinte modo:

1. apenas podemos adicionar na primeira posição da lista,
2. apenas podemos remover o primeiro elemento da lista.

Estas restrições fazem com que uma pilha seja também designada por uma lista LIFO (last-in-first-out).

TAD Pilha (Stack) - definição

Um TAD-pilha é uma sequência de elementos, $S = [a_1, a_2, \ldots, a_n]$, em que a_n é o elemento mais recente da sequência (ou elemento do topo da pilha), juntamente com as operações:

1. $x = pop(S)$ remove e retorna o elemento no topo de S
2. $push(x, S)$ insere x no topo de S
3. $top(S)$ retorna a_n, o elemento no topo de S (mas não altera S)
4. $isEmpty(S)$ retorna `true` se S estiver vazio, `false` caso contrário.
5. $isFull(S)$ retorna `true` se S estiver cheio, `false` caso contrário.

Uma interface Pilha (Stack)

A interface define as assinaturas dos métodos públicos do TAD-Pilha (comentários em Javadoc).

```
/**
 * Interface para uma stack: conjunto de valores (objectos) em que
 * o último a ser inserido é o primeiro a ser removido (Last-In-First-Out)
 * @author Fernando Silva
 * @see EmptyStackException
 */
public interface Stack {
 /**
  * Retorna o número de elementos na Stack
  * @return o número de elementos na Stack
  */
  public int size();
 /**
  * Retorna se uma stack está ou não vazia
  * @return true se a stack está vazia, false caso contrário
  */
  public boolean isEmpty();
 /**
  * Retorna o elemento no topo da stack, sem o remover
  * @return o elemento do topo da stack
  * @exception EmptyStackException se a stack estiver vazia
  */
```


Uma interface Pilha (Stack) (2)

```
  public Object top() throws EmptyStackException;
 /**
  * Insere um novo elemento no topo da stack
  * @param new elemento a ser inserido
  */
  public void push(Object new);
 /**
  * Remove o elemento no topo da stack
  * @return elemento removido
  * @exception EmptyStackException se a stack estiver vazia
  */
  public Object pop() throws EmptyStackException;
}
```

Duas implementações possíveis:

- com vectores
- com listas ligadas

Pilhas em Java usando vectores (1)

Podemos implementar a classe Stack usando vectores e objectos genéricos.

```
class ArrayStack implements Stack {
   public static final int MAXSIZE=100; // tamanho por defeito
   private Object val[]; // elementos
   private int    top;   // elemento no topo
   private int    cap;   // capacidade

   ArrayStack() {      // constructor
     this(MAXSIZE);
   }
   ArrayStack(int c) { // constructor
     cap= c;
     val= new Object[cap];
     top=0;
   }
   public int size() { return top; }
   public boolean isEmpty() { return (top==0); }
   public boolean isFull() { return (top==cap);}
}
```


Pilhas em Java usando vectores (2)

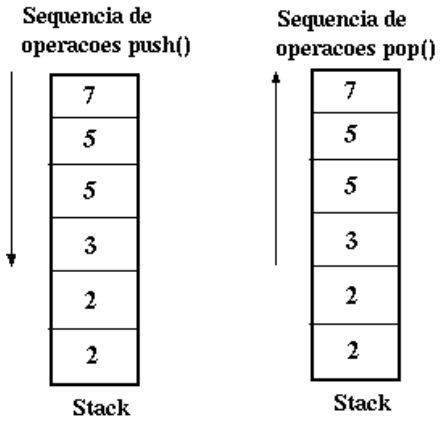
```
   public void push(Object x) throws FullStackException {
      if (isFull())
         throw new FullStackException("Stack overflow.");
      val[top]= x;
      top++;
   }
   public Object top() throws EmptyStackException {
      if (isEmpty())
        throw new EmptyStackException("Stack is empty.");
      return val[top-1];
   }
   public Object pop() throws EmptyStackException {
     if (isEmpty())
        throw new EmptyStackException("Stack is empty.");
     top--;
     return val[top];
  }
  // outros métodos incluindo clonagem de uma pilha
}
```

Pilhas em Java usando listas

```
class NodeStack implements Stack {
  Node top;   // referência ao topo da pilha
  int  size;  // tamanho da pilha

  public NodeStack() { top= null; size=0; }
  public boolean isEmpty() { return (top==null); }
  public int size() { return size; }

  public Object pop() throws EmptyStackException {
    if (isEmpty())
      throw new EmptyStackException(``Stack is empty.'');
    Object res= top.val();
    top= top.next();
    size--;
    return res;
  }
  public void push(Object x) {
    top= new Node(x, top);
    size++;
  }
  // outros métodos ...
}
```


Exemplo com pilhas: inverter uma sequência de valores

Problema: escrever um programa em que dado um valor inteiro, e por recurso a uma pilha, inverta uma lista com os seus divisores primos por ordem decrescente. Por exemplo, dado 2100 o resultado devia ser: 7 5 5 3 2 2.



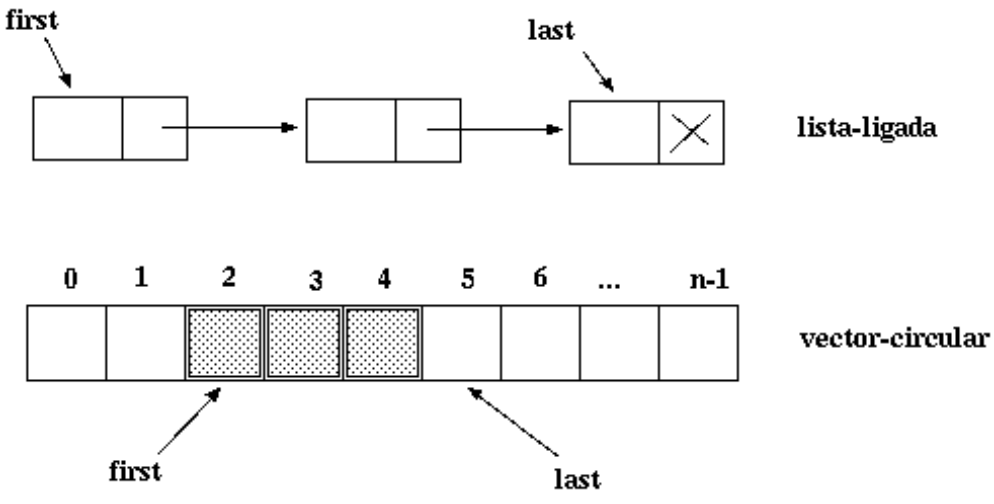
Exemplo com pilhas (cont.)

```
// usar a class NodeStack e Node definidas anteriormente
class InverteLista {
   public static void main(String[] args) {
     int d,x,ctr,n;
     NodeStack s= new NodeStack();    // cria stack vazia
     Scanner in = new Scanner(System.in);

     n= in.nextInt(); // numero inicial
     ctr=0;
     x=n;
     d=2;             // caso do divisor 2
     while (x%d==0) {
        s.push(d);
        ctr++;
        x=x/d;
     }
     for (d=3; d<Math.sqrt(n); d += 2)  // tenta divisores impares
        while (x%d==0) {
           s.push(d);
           ctr++;
           x=x/d;
        }
      for (int i=0; i<ctr; i++) // lê da pilha, ordem inversa
         System.out.print(" " + s.pop());
      System.out.println();
   }
}
```


TAD-Fila (Queue) (1)

Uma fila difere de uma pilha na medida em que opera na base de um FIFO (first-in-first-out). Assim,

1. adicionamos novos elementos ao fim da fila, e
2. removemos sempre do princípio da fila.



TAD-Fila (Queue) (2)

- Um TAD-fila é uma sequência de elementos, $F = [a_1, a_2, \ldots, a_n]$, em que a_1 é o primeiro elemento da fila e a_n é o último, juntamente com as seguintes operações (asseguram que funciona como um FIFO):

  1. *init()* inicializa a fila em vazio;
  2. *isEmpty()* verifica se a fila está vazia;
  3. *isFull()* verifica se a fila está cheia;
  4. *add(x)* adiciona x na última posição da fila;
  5. *peek()* retorna o valor do 1o elemento da fila;
  2. *remove()* remove o 1o. elemento da fila e retorna esse valor;

- **Aplicações:** simulação de filas (bancos, supermercados, atendimento público, etc.), implementação a baixo nível da leitura da linha de comando, pesquisa breadth-first de uma árvore, etc.

Uma interface Fila (Queue) (1)

Descreve os nomes dos métodos públicos do TAD-Fila, como são declarados e usados (comentários em Javadoc).

```
/**
 * Interface para uma queue: conjunto de valores (objectos) em que
 * o primeiro a ser inserido é o primeiro a ser removido (FIFO)
 * @author Fernando Silva
 * @see EmptyQueueException
 */
public interface Queue {
 /**
  * Retorna o número de elementos na fila
  * @return o número de elementos na fila
  */
  public int size();
 /**
  * Retorna se uma fila está ou não vazia
  * @return true se a fila está vazia, false caso contrário
  */
  public boolean isEmpty();
```

Uma interface Fila (Queue) (2)

```
 /**
  * Retorna o 1o. elemento da fila, sem o remover
  * @return o 1o. elemento da fila
  * @exception EmptyQueueException se a fila estiver vazia
  */
  public Object peek() throws EmptyQueueException;
 /**
  * Remove o primeiro elemento da fila
  * @return elemento removido
  * @exception EmptyQueueException se a fila estiver vazia
  */
  public Object remove() throws EmptyStackException;
 /**
  * Insere novo elemento na última posição da fila
  * @param elem elemento a ser inserido
  */
  public void add(Object elem);
}
```


Implementação de filas: vector circular (1)

- A ideia é representar a fila por um vector circular, em que `first` e `last` são cursores do vector que apontam para o início e fim da fila.
- Condições importantes a verificar:
  - quando os cursores estiverem na última posição do vector, `cap-1`, e forem incrementados devem passar para a posição `0`;
  - verificar se a lista está vazia:
    - ★ verificar se `size==0`, em que `size` é o número de elementos na fila,
    - ★ ou, a lista está vazia se `first==last`.
  - verificar se a lista está cheia:
    - ★ verificar se `size==cap`, em que `size` é o número de elementos na fila e `cap` a capacidade do vector.,
    - ★ ou, a lista está cheia se `((last+1)%MAX)==first` (obriga a deixar uma posição por usar no vector);

Implementação de filas: vector circular (2)

```
class ArrayQueue implements Queue {
    private static final int MAX= 100;
    private Object queue[];// fila
    private int size;       // num. elementos
    private int first;      // primeiro da fila
    private int last;       // ultimo da fila
    private int cap;        // capacidade

    ArrayQueue()  {       // construtor
        this(MAX);
    }
    ArrayQueue(int c)  {   // construtor
        cap= c;
        queue= new Object[cap];
        size= first= last=0;
    }
    public boolean isEmpty()  { return (size==0); }
    public boolean isFull()  { return (size==cap); }
    public int size() { return size; }
```


Implementação de filas: vector circular (3)

```
    public Object peek() throws EmptyQueueException {
        if (isEmpty())
            throw new EmptyQueueException("Fila vazia!");
        return queue[first];
    }
    public Object remove() throws EmptyQueueException {
        if (isEmpty())
            throw new EmptyQueueException("Fila vazia!");
        Object r= queue[first];  // remove primeiro da fila
        first= (first + 1) % cap;
        size--;
        return r;
    }
    public void add(Object item) throws FullQueueException {
        if (isFull())
            throw new FullQueueException("Fila cheia!");
        queue[last]= item;
        last= (last+1) % cap;
        size++;
    }
}
```

Implementação de filas: listas ligadas (1)

A implementação dos métodos segue de perto a implementação de listas ligadas.

```
class NodeQueue implements Queue {
  private Node first; // primeiro da fila
  private Node last;  // ultimo da fila
  private int size;   // num. elementos

  NodeQueue() { // construtor
    size=0;
    first= last= null;
  }
  public int size() { return size; }
  public boolean isEmpty()  { return (size==0); }

  public Object peek() throws QueueEmptyException
  { // aceder ao 1o. elem. da fila
    if (isEmpty())
          throw new QueueEmptyException("Fila vazia!");
    return first.val;
  }
```

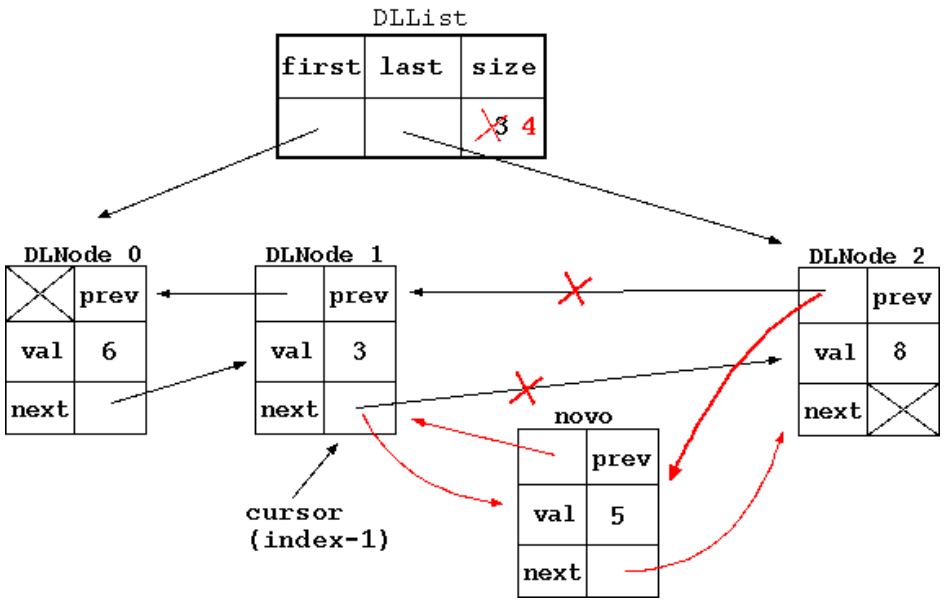

Implementação de filas: listas ligadas (2)

```
public Object remove() throws QueueEmptyException  {
  if (isEmpty())
        throw new QueueEmptyException("Fila vazia!");
  Object r= first.val;   // remove primeiro da fila
  first= first.next;
  size--;
  if (first==null) last= null; // se a lista ficou vazia
  return r;
}
public void add(Object item) {
  Node n= new Node(item, null);
  if (isEmpty()) {
    last=n;
    first=last;
  }
  else {
    last.next= n;
    last= n;
  }
  size++;
}
```

Listas duplamente ligadas

Uma **lista duplamente ligada**

é uma sequência de elementos em que cada elemento, com excepção do primeiro e último, contém um valor e referências para o elemento anterior e elemento seguinte.



Implementação de listas duplamente ligadas

Um nó de uma lista duplamente ligada fica caracterizado pela classe `DLNode`:

```
class DLNode {   // Double Linked Node
    Object val;  // valor do nó
    DLNode prev; // nó anterior
    DLNode next; // nó seguinte

    DLinkedNode(Object v, DLNode p, DLNode n) {
        val= v;
        prev= p;
        next= n;
    }
}
```

Nota: veremos mais tarde que podemos substituir o tipo `Object` por um tipo de dados genérico. Toda a restante definição será idêntica.

Implementação de listas duplamente ligadas

```
class DLList  {   // Double Linked List
    DLNode first; // primeiro elemento
    DLNode last;  // último elemento
    int    size;  // num. de elementos

    DLList() { // lista vazia
        first= last= null;
        size= 0;
    }
    // inserir um novo elemento com o valor v
    // na posição index da lista
    // pre-condição: 0<= index < size
    void add(Object v, int index) {
        DLNode cursor= first;
        for (int i=0; i<index-1; i++)
            cursor= cursor.next;
        DLNode novo= new DLNode(cursor, cursor.next);
        novo.next.prev= novo; // ou cursor.next.prev= novo;
        cursor.next= novo;
        size++;
    }
    // outros métodos...
}
```


Dequeues: Double-Ended Queues (deque ou dequeue)

Uma Deque (pronuniciar "Deck")

é uma fila com dupla terminação, sendo possível operações de inserção e remoção no início ou fim da fila.

Um **TAD-deque** pode ser visto como uma sequência de elementos duplamente ligados sobre os quais é possível ter as seguintes operações (além das habituais sobre filas):

`insertFirst(x)`	insere `x` no início da deque.
`insertLast(x)`	insere `x` no fim da deque.
`removeFirst()`	remove e retorna primeiro elemento.
`removeLast()`	remove e retorna último elemento.

A implementação de uma deque deve ser feita com listas duplamente ligadas. Usará a classe `DLNode`.

Implementação de Dequeues: classe e `addLast()`

```
class MyDeque {           // Double ended queue
    private int    size;  // num. elementos
    private DLNode first; // primeiro elemento
    private DLNode last;  // último elemento

    MyDeque() {
        size= 0;
        first=last=null;
    }
    boolean isDequeEmpty() { return size==0; }
    int size() {return size;}

    void addLast(Object x) {
        DLNode novo= new DLNode(x, null, null);

        if (isDequeEmpty())
            first=last=novo;
        else {
            last.next= novo;
            novo.prev= last;
            last= novo;
        }
        size++;
    }
```


Implementação de Dequeues: método `removeLast()`

```
    Object removeLast() {
        if (isDequeEmpty())
            throw new NoSuchElementException("Deque vazia!");
        // remove ultimo
        Object res= last.val;
        last= last.prev;
        if (last!=null)  // se existia mais do que um
            last.next= null;
        size--;
        if (size==0)     // se só havia um elemento
            first=null;  // last já está em nulo
        return res;
    }
}
```

Exercício: Implemente os restantes métodos de uma dequeue: `addFirst()`, `removeFirst()`, `isDequeueEmpty()`.

Implementação de Dequeues: classe exemplo de uso

```
class DequeEx { // exemplo de uso da classe MyDeque
    public static void main(String[] args) {
        int i, v1, v2;
        int n=10;
        MyDeque Q= new MyDeque();

        for (i=n; i>0; i--)
            Q.addFirst(i);
        for (i=0; i<n; i++)
            Q.addLast(i+1+N);
        for (i=0; i<n; i++) {
            v1= Q.removeFirst();
            v2= Q.removeLast();
            System.out.println(v1+" "+ v2);
        }
    }
}
```

O programa coloca os números de 10 a 1 no início da fila, desse modo invertendo essa sequência, e coloca no fim da fila os números de 11 a 20. Ao retirarmos um elemento do início e outro do fim, obtem-se pares da forma: (1,20), (2,19), ...

Vectores vs. Listas Ligadas vs. Duplamente Ligadas

Muitos TADs podem ser implementados usando vectores ou listas ligadas. Qual será a melhor aproximação?

- os vectores são melhores para acesso aleatório;
- listas são melhores para operações de adição e remoção de elementos;
- listas duplamente ligadas para operações que requeiram movimentos nas duas direcções da lista;
- listas evitam as ineficientes operações de redimensionamento de capacidade.

Listas, pilhas e filas pré-definidas em Java

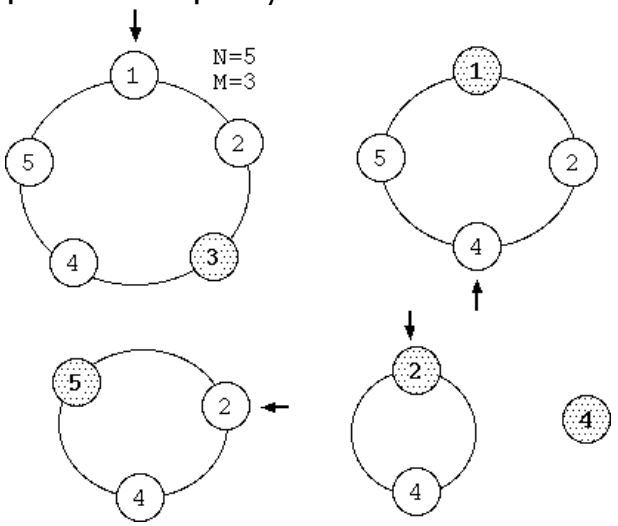
Existem classes pré-definidas no Java na package `java.util` que permitem usar estruturas como listas, dequeues, filas e pilhas.

- Stacks com objectos genéricos: `java.util.Stack<E>`
  Métodos: `push(obj)`, `pop()`, `peek()`, `size()`, e `empty()`
- Filas `Queue<E>` e Dequeues `Deque<E>` A classe Deque é mais geral.
- Listas ligadas (permitem implementar Filas/Pilhas/Dequeues): `java.util.LinkedList<E>`
  Para implementar filas, usar os métodos: `addLast(obj)` e `removeFirst()`.
  Esta classe é muito flexível.

Estas classes são novas na versão do Java, podendo ser parametrizadas por tipo de dados, por exemplo, posso ter uma stack de inteiros declarando um objecto do tipo `Stack<Integer>`.

Problema de Josephus: listas circulares sem vectores

Imagine que N pessoas decidem eleger um líder usando um método de eliminações sucessivas, ficando como líder o último a ser eliminado. As N pessoas dispõe-se num círculo e elimina-se a M-ésima pessoa, cerrando fileiras com os restantes.
A pessoa a ser eleita depende do N e do M. Para o exemplo seguinte, a pessoa na posição inicial 4 seria a eleita.



Listas circulares em Java (sem vectores) (1)

```
import java.io.*;
import java.util.*;

class Node {
    Object  val;
    Node   next;

    Node(Object v) { val=v; next=null; }
}
class CircularList {
    private Node head;

    CircularList() { head=null; }
    void next() { head=head.next; }
    Object value() { return head.val; }
    boolean lastElement() {
        return (head==head.next);
    }
    void advance(int m) {
        for (int i=1; i<m; i++) next();
    }
```


Listas circulares em Java (sem vectores) (2)

```
void addFirst(Object v) {
    Node t= new Node(v);
    if (head==null) { // 1o elemento
        t.next= t;    // fica circular
        head=t;
    }
    else {            // insere no início
        t.next= head.next;
        head.next= t;
        head=t;
    }
}
void removeFirst() {     // remove sucessor de head
  if (head!=null) {
    if (head==head.next) // só tem um elemento
        head= null;
    else                 // mais do que um elem.
      head.next= head.next.next;
  }
}
```

Classe principal para o problema do Josephus

```
class Josephus {
    public static void main(String[] args)
    {
        Scanner in= new Scanner(System.in);
        int n= in.nextInt();
        int m= in.nextInt();
        CircularList l= new CircularList();

        for (int i=1; i<= n; i++)
            l.addFirst(i);
        while (!l.lastElement()){
            l.advance(m);
            l.removeFirst();
        }
        System.out.println("Winner: " + l.value());
    }
}
```

Problemas semelhantes nas aulas: P06. Em Valladolid: 130-133-151-180-305-402-440-10015.