



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica



Tema 5.

Estructures de dades lineals

Programació (PRG)

Curs 2019/20

Departament de Sistemes Informàtics i Computació



Continguts

Duració: 8 sessions

1. Introducció. Tipus de dades lineals. Estructures de dades lineals.
2. Seqüències enllaçades. La classe Node. Recorreguts. Cerques. Inserció. Eliminació.
3. Representació enllaçada de tipus lineals.
4. Piles. Operacions. Implementació amb arrays. Implementació enllaçada. Comparació d'implementacions.
5. Cues. Operacions. Implementació amb arrays. Implementació enllaçada. Comparació d'implementacions.
6. Llistes amb punt d'interés. Operacions. Implementació amb arrays. Implementació enllaçada. Comparació d'implementacions.

- Pràctiques relacionades:

PL5. Implementació i ús d'estructures de dades lineals (2 sessions)

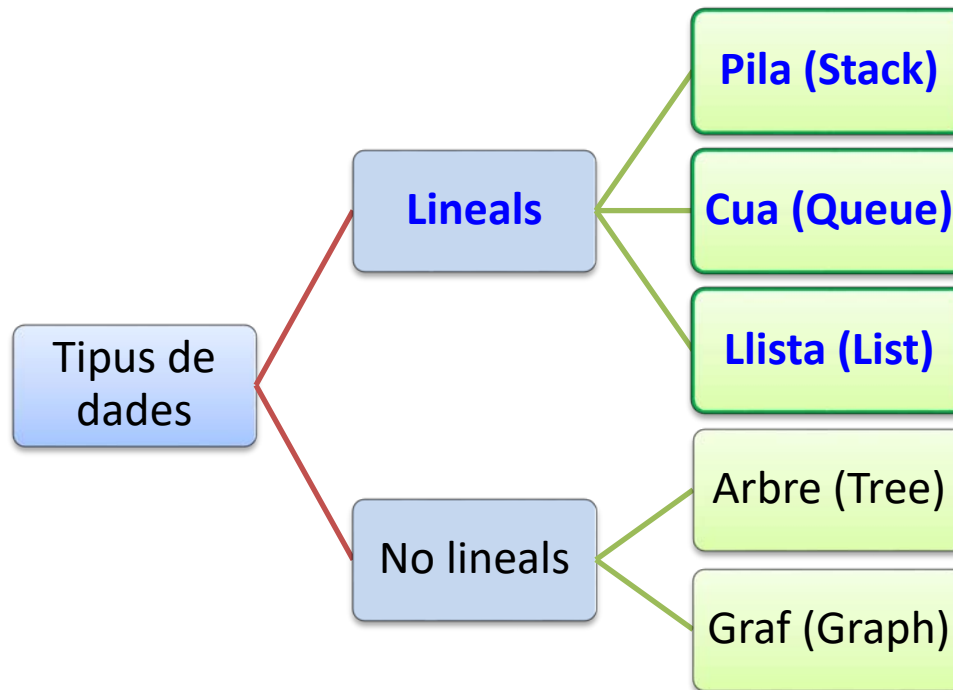


- **Descarrega** (del Tema 5 de PoliformaT) el fitxer **llibrariesPRG.jar** en una carpeta **PRG/Tema 5** dins del teu **disc W**
- Des de l'opció **Projecte** de **BlueJ**, usa l'opció **Open ZIP/JAR...** per tal d'obrir aquest com un projecte **BlueJ** que conté diferents paquets i prepara't per usar-los



Introducció. Tipus de dades lineals

- Els **tipus de dades lineals** són aquells tals que els seus elements estan formats per **linealitats** o seqüències $d_0 d_1 \dots d_{n-1}$, $n \geq 0$ en les que els d_i són dades del mateix tipus, i sobre les que es poden fer operacions d'inserció i eliminació de dades, consulta de la dada que ocupa determinada posició, etc.
- Segons la política de manipulació de les dades, es distingeixen les tres linealitats que es presenten en el tema: **piles**, **cues** i **llistes**, l'ús de les quals resulta idoni en una àmplia varietat d'aplicacions informàtiques.



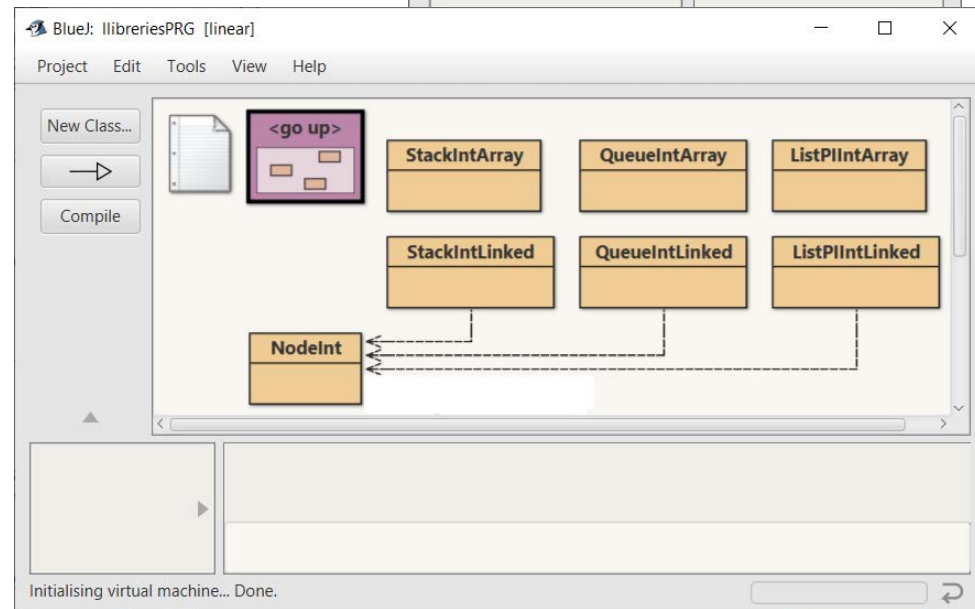
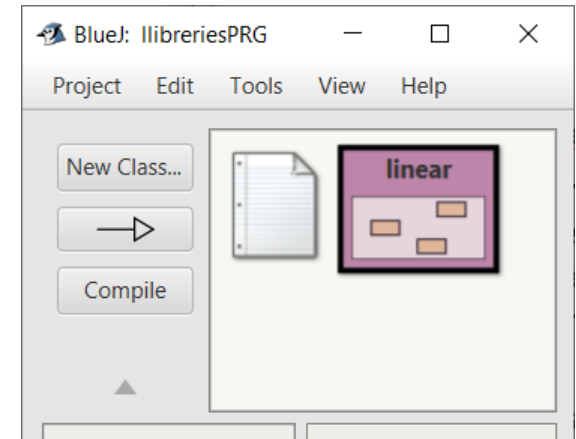
Diferents en la política de gestió de les dades

Introducció. Estructures de dades lineals

- El comportament temporal de les operacions d'un tipus de dades va a dependre de com s'implementen les operacions basant-se en la representació o estructuració de les dades escollida.
- *Estructura de Dades (ED)*: És un tipus de dades amb una representació de les dades determinada i la corresponent implementació de les operacions del tipus.
- En aquest tema es van a presentar les **estructures de dades lineals** basades en dues maneres alternatives de representar les seqüències (d'enters, per simplificar l'exposició):
 - Amb arrays: *StackIntArray*, *QueueIntArray*, *ListPIIntArray*.
 - Enllaçada: *StackIntLinked*, *QueueIntLinked*, *ListPIIntLinked*.

Introducció. Estructures de dades lineals

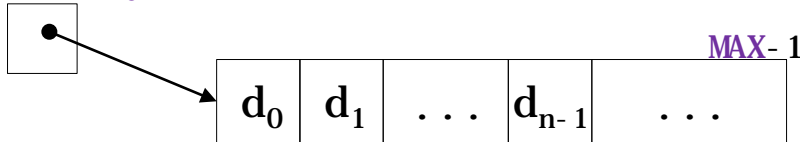
- Aquest conjunt de classes pot constituir un exemple de **llibreria d'usuari**, **linear**, organitzada com un **paquet** Java. Totes les classes han de contindre la declaració: `package linear;` i estar totes incloses en la mateixa carpeta **linear**.
- Es pot dedicar un projecte o carpeta, per exemple, **llibrariesPRG**, per a contindre aquest i altres paquets d'interès general.
- Per a instal·lar aquestes llibreries en el sistema, caldria afegir la ruta de **llibrariesPRG** en la variable d'entorn **CLASSPATH**.
- CLASSPATH** és una variable que conté la ruta de les carpetes que Java examina quan busca els paquets importats per una classe en compilar-la o executar-la.
- En **BlueJ** la variable **CLASSPATH** es modifica en la finestra desplegada per l'opció **Eines>Preferències>Llibreries**.



Seqüències enllaçades

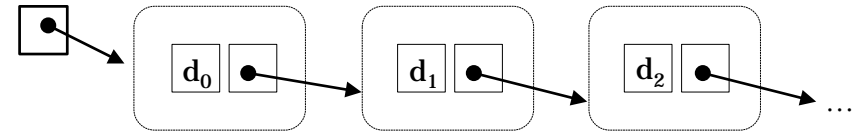
- Representació amb array.
- Representació d'una seqüència disposant els seus elements en les successives posicions d'un array de longitud suficientment gran.

theArray



- Aquesta representació permet que l'accés als elements de la seqüència es pugui realitzar amb cost constant.
- La talla de la seqüència està limitada a la longitud MAX de l'array.
- La inserció/eliminació d'una dada en la posició i -èsima de la seqüència exigeix reorganitzar `theArray[i .. n-1]`.

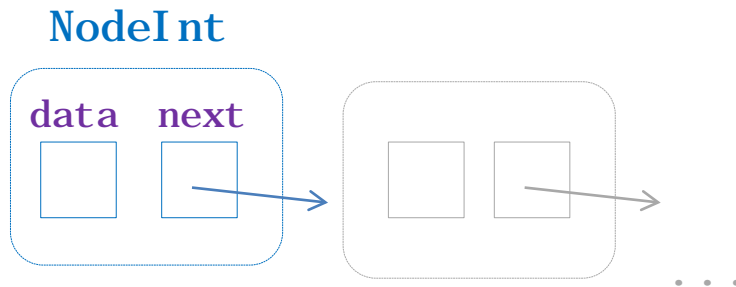
- Representació enllaçada.
- Representació d'una seqüència disposant de memòria per a les seues dades a mesura que es van inserint en la seqüència.
- Tota dada té associat un enllaç (o referència) a la posició en que es troba en el *heap* la següent dada de la seqüència.



- L'accés a les dades ja no és directe per la seua posició en la seqüència.

Seqüències enllaçades. La classe Node

- **Node:** Estructura que associa una dada i l'enllaç a la següent dada.



```
/**
 * Classe NodeInt: Node la
 * dada del qual és un int
 */
class NodeInt {

    int data;
    NodeInt next;

    ...

}
```

Classe amb atributs “friendly” a incloure en el mateix paquet que aquelles classes que hagen d’usar seqüències enllaçades mitjançant nodes (**Stack**, **Queue** i **List**).

Seqüències enllaçades. La classe Node

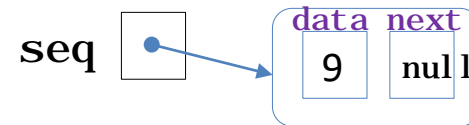
En la classe es van a considerar dos constructors:

- **Constructor A:** Crea un node amb una dada **d** que no té següent.
- **Constructor B:** Crea un node amb una dada **d** enllaçada a un node preexistent.

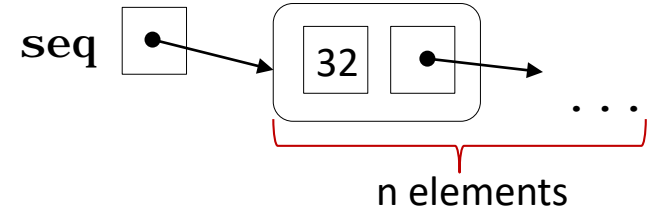
```
class NodeInt {  
  
    int data;  
    NodeInt next;  
  
    /** Constructor A */  
    NodeInt(int d) {  
        data = d;  
        next = null;    this(d, null);  
    }  
  
    /** Constructor B */  
    NodeInt(int d, NodeInt s) {  
        data = d;  
        next = s;  
    }  
}
```

Exemple: Constructor A

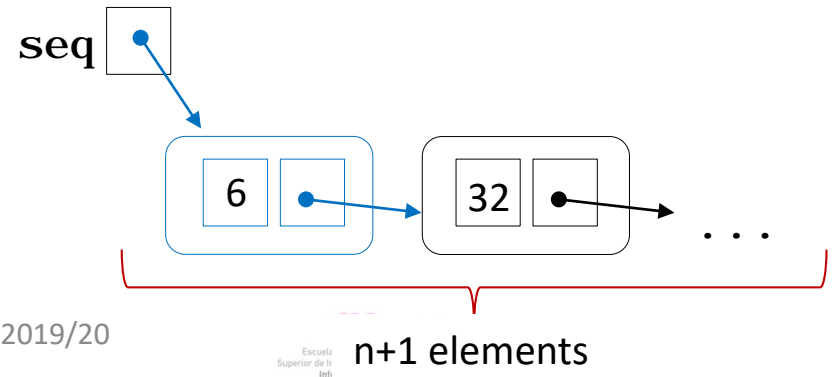
```
NodeInt seq = new NodeInt(9);
```



Exemple: Constructor B



```
seq = new NodeInt(6, seq);
```

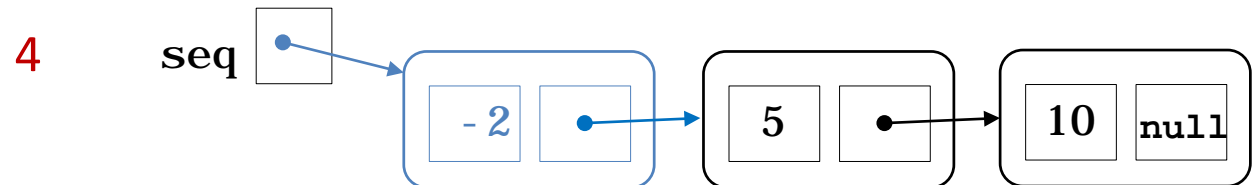
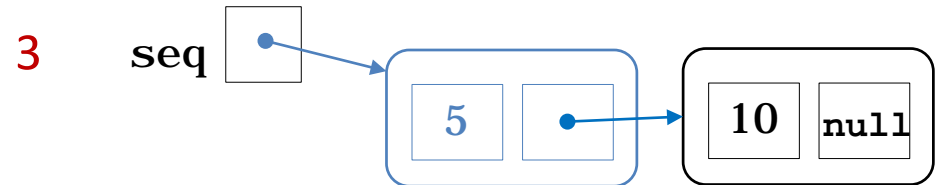
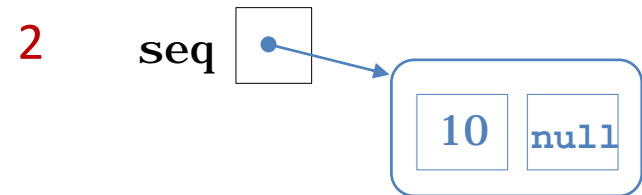


Seqüències enllaçades. La classe Node

- Exemple:

```
1 NodeInt seq = null;  
2 seq = new NodeInt(10);  
3 seq = new NodeInt(5, seq);  
4 seq = new NodeInt(-2, seq);
```

1 seq **null** (seqüència buida)



Seqüències enllaçades. La classe Node

- Els constructors de `NodeInt` faciliten la inserció en cap.
- La manipulació explícita d'enllaços permet accedir a altres posicions.
- **Exemple:** el següent codi es suposa en una classe amb accés als atributs de `NodeInt`.

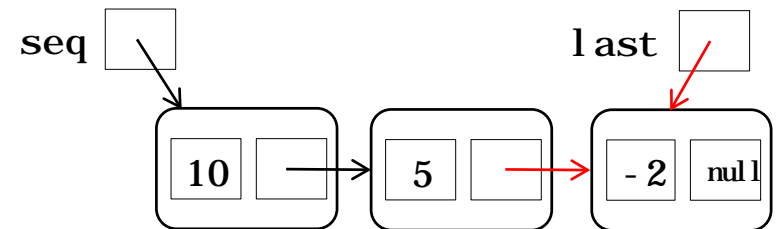
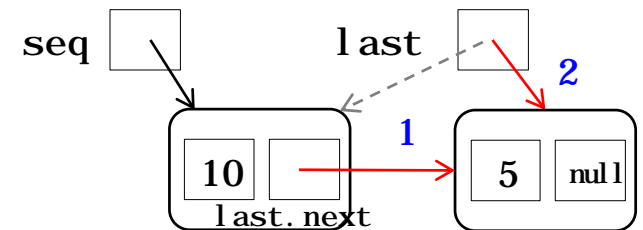
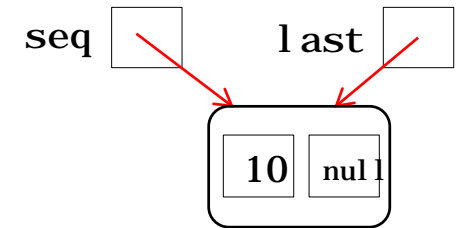
```
NodeInt seq = null, last = null;
```

```
seq = new NodeInt(10);  
last = seq;
```

```
last.next = new NodeInt(5); // 1  
last = last.next; // 2
```

```
last.next = new NodeInt(-2);  
last = last.next;
```

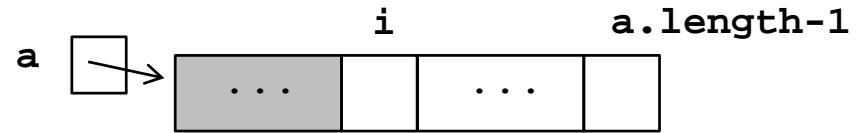
seq □ null last □ null



Seqüències enllaçades. Recorreguts

- **Recorregut** d'arrays i de seqüències enllaçades, realitzant una operació **tractar** a tots els seus elements.

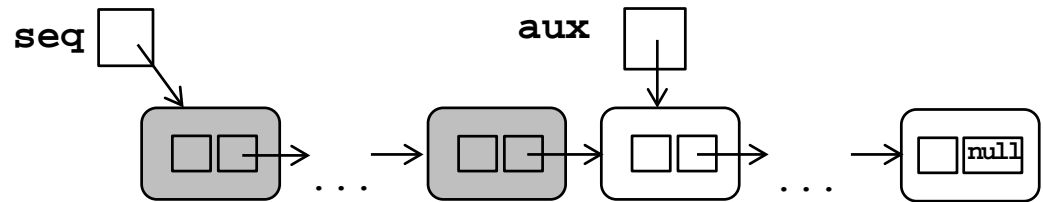
```
// recorregut d'un array
int i = 0;
while (i < a.length) {
    tractar(a[i]);
    i++;
}
```



Si i està fora de rang, l'accés a $a[i]$ provoca `ArrayIndexOutOfBoundsException`

```
// recorregut d'un array
for (int i = 0; i < a.length; i++) {
    tractar(a[i]);
}
```

```
// recorregut d'una
// seqüència enllaçada
NodeInt aux = seq;
while (aux != null) {
    tractar(aux.data);
    aux = aux.next;
}
```



Si aux és `null`, l'accés a $aux.data$ i $aux.next$ provoca `NullPointerException`

```
// recorregut d'una seqüència enllaçada
for (NodeInt aux = seq; aux != null; aux = aux.next) {
    tractar(aux.data);
}
```

Seqüències enllaçades. Recorreguts



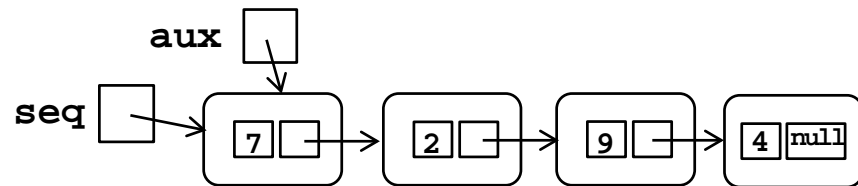
- **Exemple.** Donada una seqüència enllaçada, es desitja que els seus valors saturen a un cert valor **max**, és a dir, que els valors $> \mathbf{max}$ es canvien a **max**:

```
public static void saturar(NodeInt seq, int max) {  
    NodeInt aux = seq;  
    while (aux != null) {  
        if (aux.data > max) { aux.data = max; }  
        aux = aux.next;  
    }  
}
```

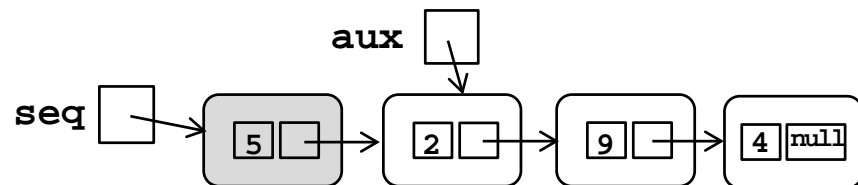
Seqüències enllaçades. Recorreguts

- Traça exemple de **saturar** amb un valor **max** igual a 5:

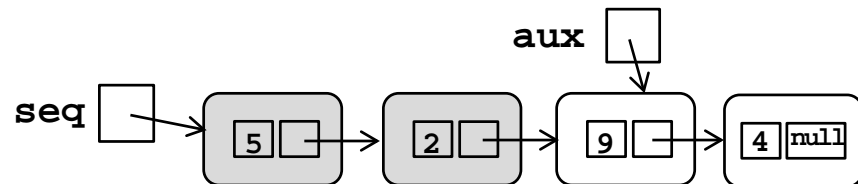
Després de la inicialització del bucle:



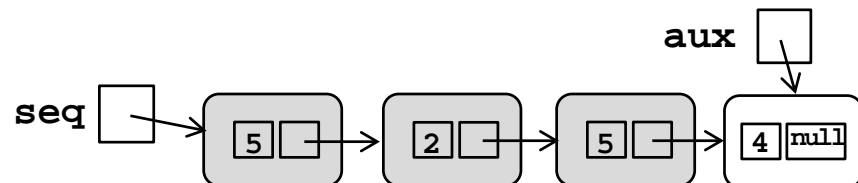
Després de la primera passada:



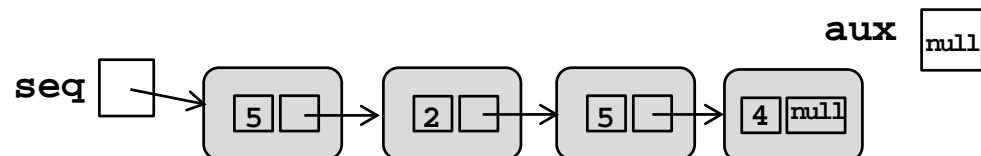
Després de la segona passada:



Després de la tercera passada:



Després de la darrera passada:

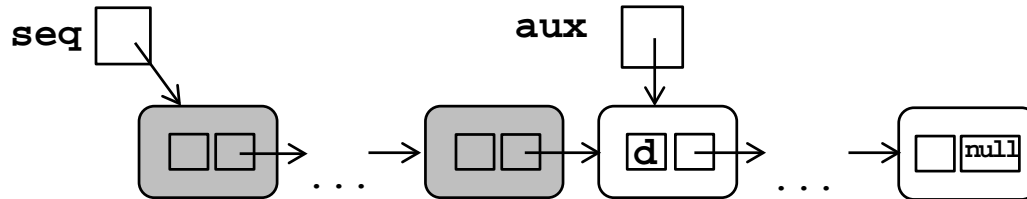


Seqüències enllaçades. Cerques

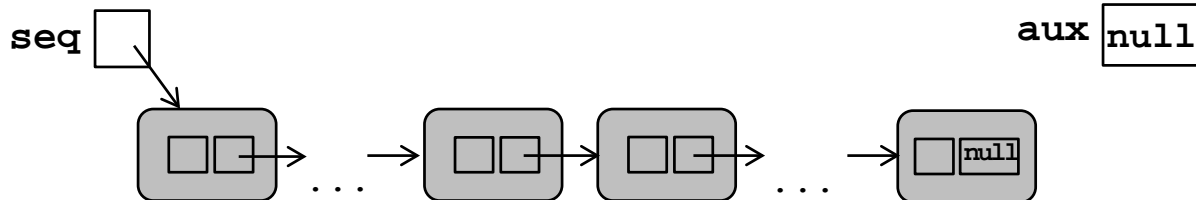
- **Exemple. Cercar** en quin node d'una seqüència enllaçada **seq** apareix la dada **d**.

```
public static NodeInt cercarDada(NodeInt seq, int d) {  
    NodeInt aux = seq;  
    while (aux != null && aux.data != d) {  
        aux = aux.next;  
    }  
    return aux;  
}
```

- Si al finalitzar el bucle **aux != null**: èxit en la cerca.



- Si **aux == null**: cerca sense èxit.



Seqüències enllaçades. Cerques

- **Exemple.** Donada la seqüència **seq** i la dada **d**, canviar el signe de la primera ocurrència de **d** en la seqüència. Si **d** no apareix, el mètode no fa res.

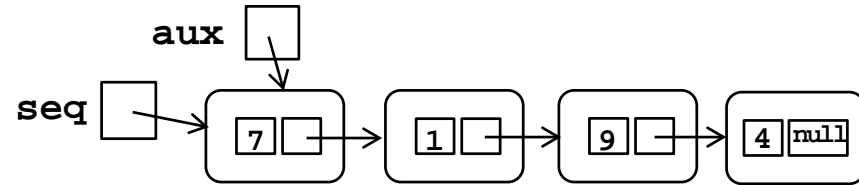
```
public static void canviarSigne(NodeInt seq, int d) {  
    // Cerca del primer node amb dada d  
    NodeInt aux = seq;  
    while (aux != null && aux.data != d) {  
        aux = aux.next;  
    }  
    // Si la cerca acaba amb èxit, es canvia  
    // el signe de la dada  
    if (aux != null) { aux.data = -d; }  
}
```

- A continuació, es presenten un parell de traces exemple, en les que sobre una mateixa seqüència, es farà primer una cerca amb fracàs, i després una altra amb èxit.

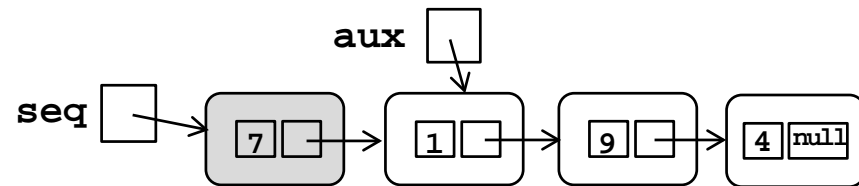
Seqüències enllaçades. Cerques

- Traça exemple del mètode `canviarSigne` amb un valor `d` igual a 25:

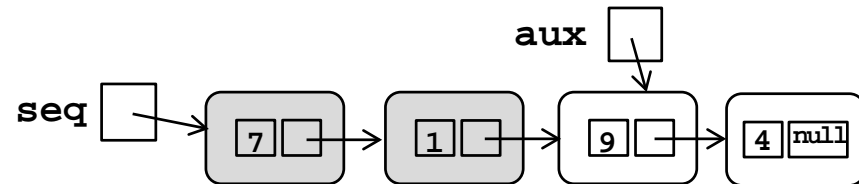
Després de la inicialització del bucle:



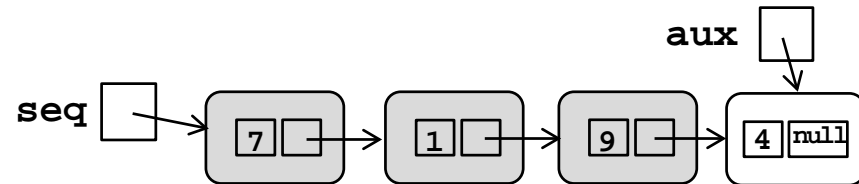
Després de la primera passada:



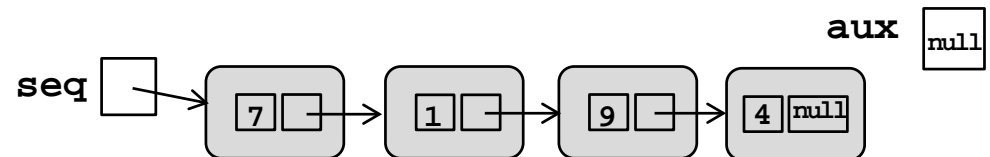
Després de la segona passada:



Després de la tercera passada:



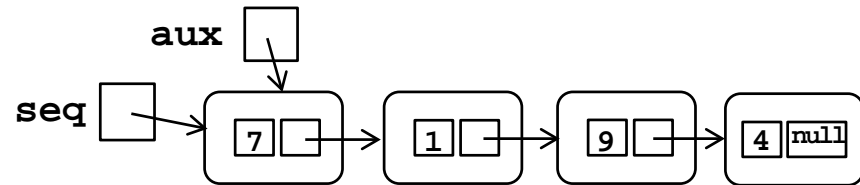
Després de la quarta i darrera passada:



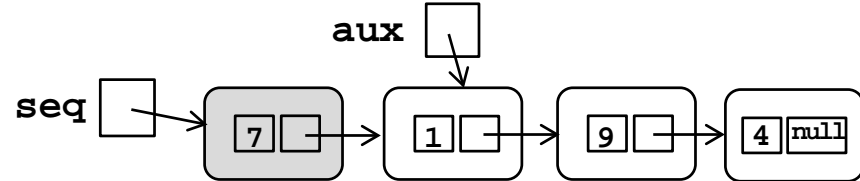
Seqüències enllaçades. Cerques

- Traça exemple del mètode **canviarSigne** amb un valor **d** igual a 1:

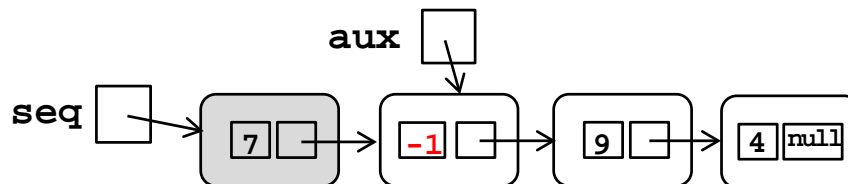
Després de la inicialització del bucle:



Després de la primera i darrera passada:



Com el bucle acaba amb **aux != null**, aleshores és **false** la condició **aux.data != d**, és a dir, s'ha trobat en el node referenciat per **aux** la dada **d**, i s'executa el canvi de signe de **aux.data**:



Seqüències enllaçades. Cerques

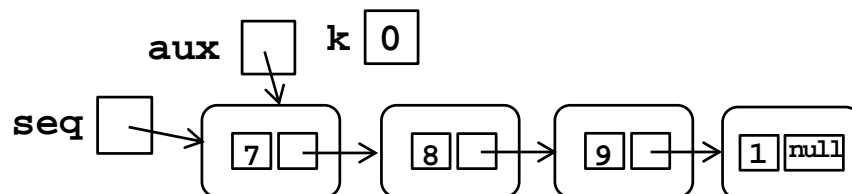
- **Exemple.** L'accés a l'*i*-èsim node d'una seqüència s'ha de resoldre cercant l'*i*-èsim node. Suposem els nodes numerats des del 0 endavant i siga **i** una variable **int** iniciada al número de node al que es vol accedir:

```
NodeInt aux = seq; int k = 0;
while (aux != null && k < i) {
    aux = aux.next;
    k++;
}
```

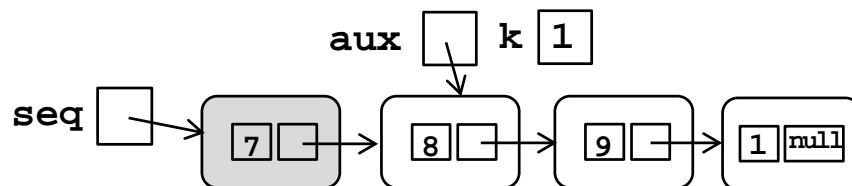
- La posició del node es va enregistrant en la variable **k**.
- En acabar el bucle:
 - Si **aux != null** aleshores **k == i**, i **aux** és l'*i*-èsim node,
 - sino, la seqüència és més curta, aquest node no existeix.

a) Traça exemple amb **i** igual a 2:

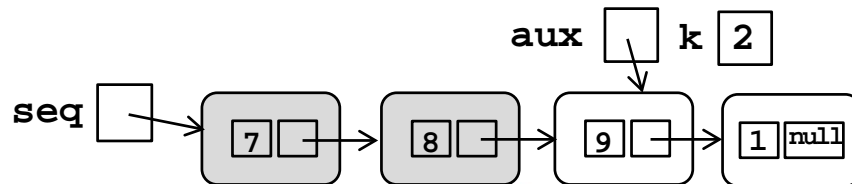
Després de la inicialització del bucle:



Després de la primera passada:



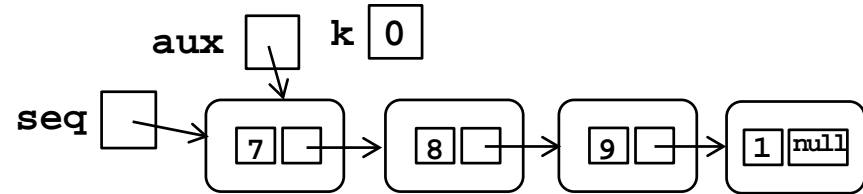
Después de la segona i última passada:



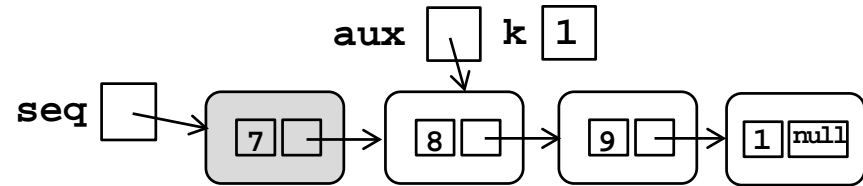
Seqüències enllaçades. Cerques

b) Traça exemple amb i igual a 5:

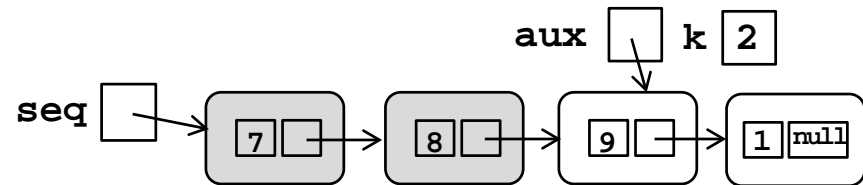
Després de la inicialització del bucle:



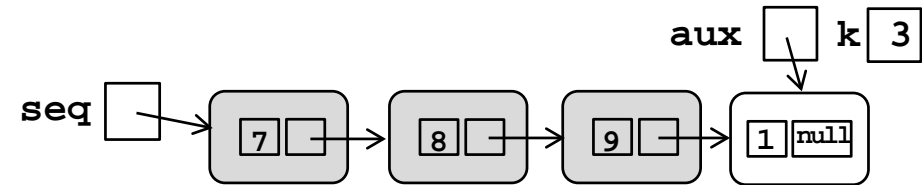
Després de la primera passada:



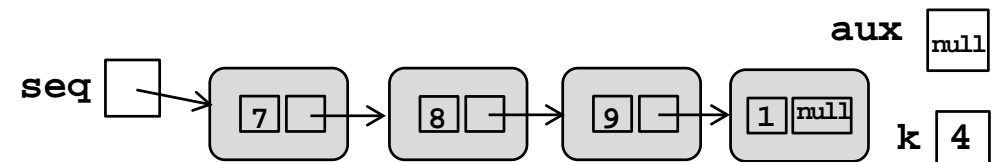
Después de la segona passada:



Després de la tercera passada:



Després de la quarta i última passada:



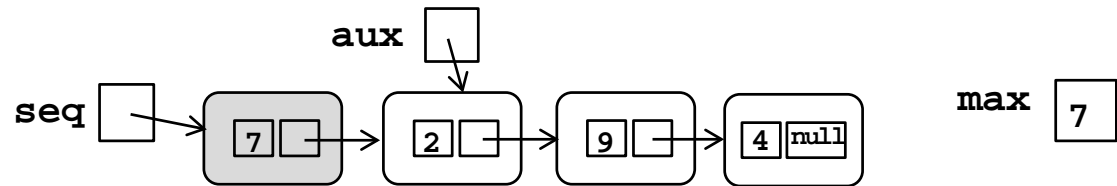
- El valor del comptador en aquest estat final significa que no existeixen nodes des de la posició 4 inclosa endavant.

Seqüències enllaçades. Exercicis

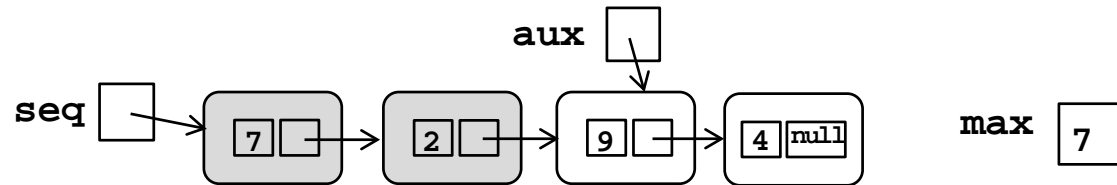
- **Completa** en la classe **UtilLinked**, del paquet **linear** del projecte BlueJ **llibreriesPRG**, els mètodes següents:
- `public static int maxim(NodeInt seq)`
que torna el màxim d'una seqüència enllaçada amb, al menys, un node.
- `public static int talla(NodeInt seq)`
que torna el nombre d'elements d'una seqüència enllaçada.
- `public static int[] toArray(NodeInt seq)`
que torna en un array (del tamany just) els elements d'una seqüència enllaçada.
- `public static String toString(NodeInt seq)`
que torna un `String` amb les dades d'una seqüència enllaçada. En l'`String` cada dada està separada de la següent per un espai en blanc.
- `public static int cercarPos(NodeInt seq, int d)`
que torna la posició de la primera aparició de **d** en **seq**. Si no apareix, ha de tornar **-1**.

- Traça exemple de **maxim**:

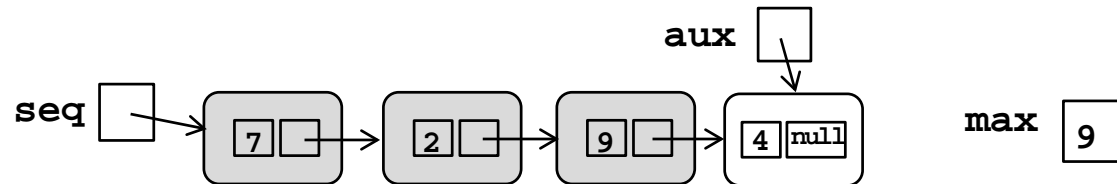
Després de la inicialització del bucle:



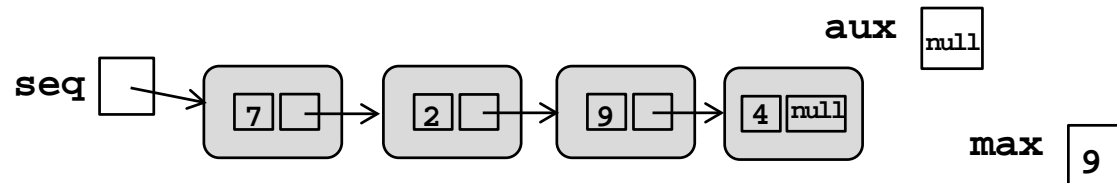
Després de la primera passada:



Després de la segona passada:

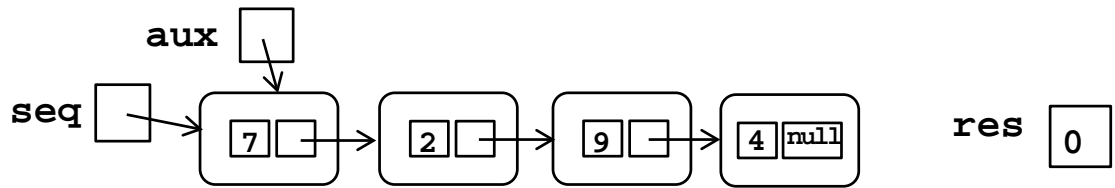


Després de la darrera passada:

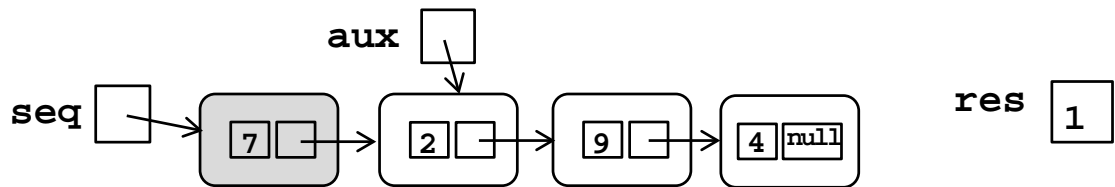


• Traça exemple de talla:

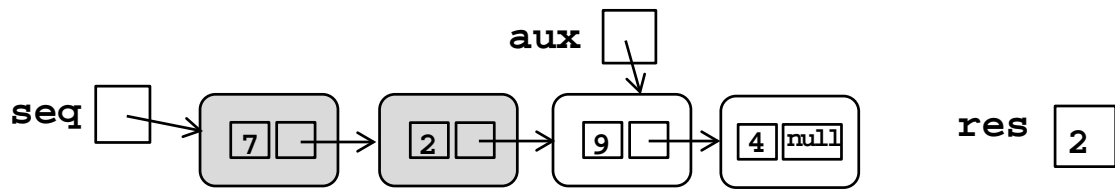
Després de la inicialització del bucle:



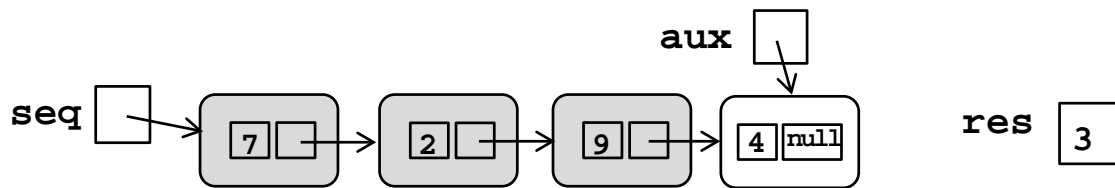
Després de la primera passada:



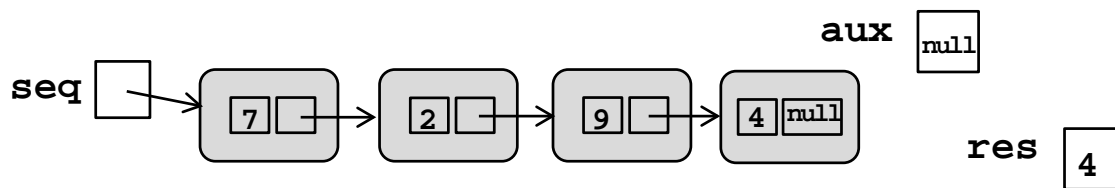
Després de la segona passada:



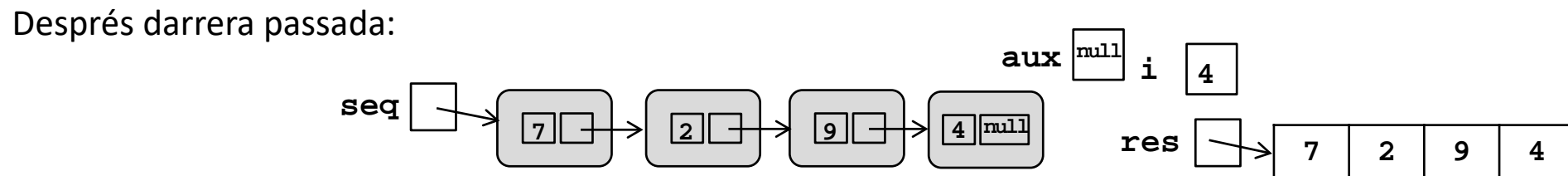
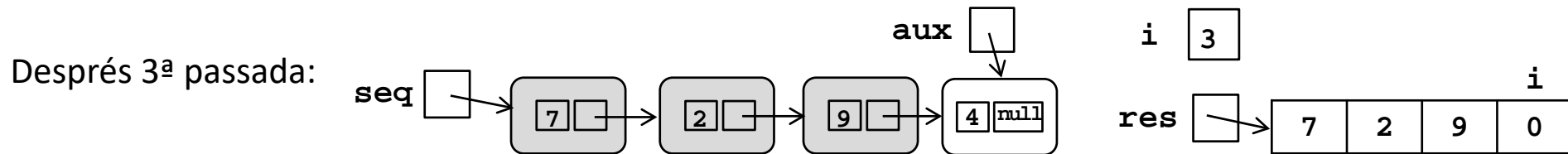
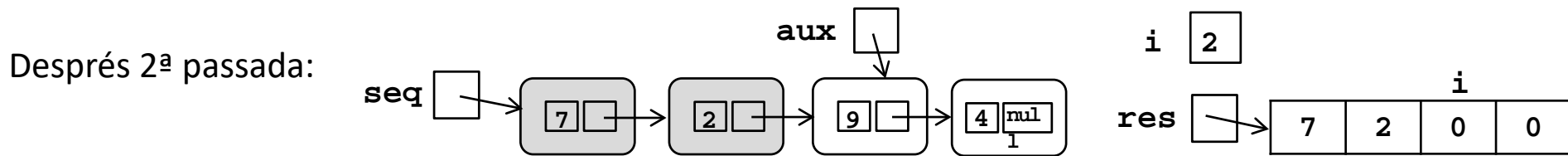
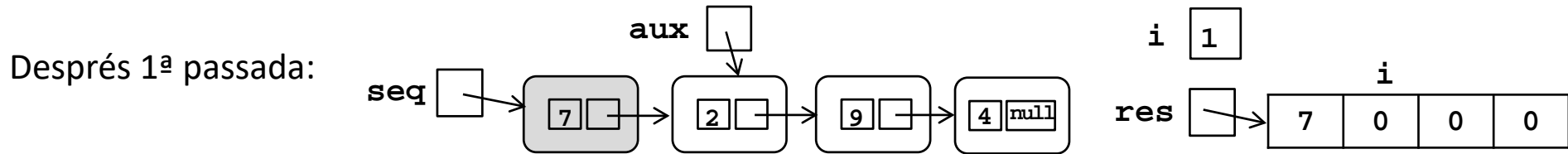
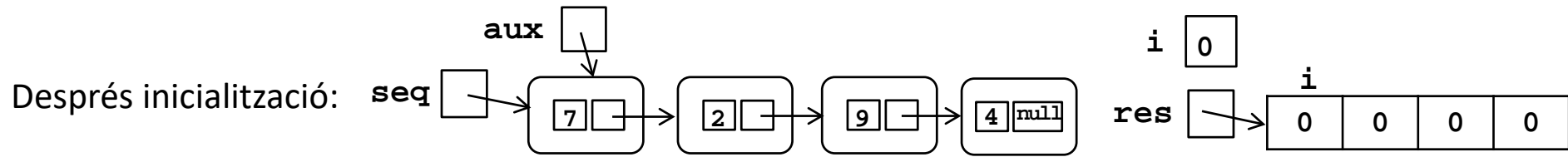
Després de la tercera passada:



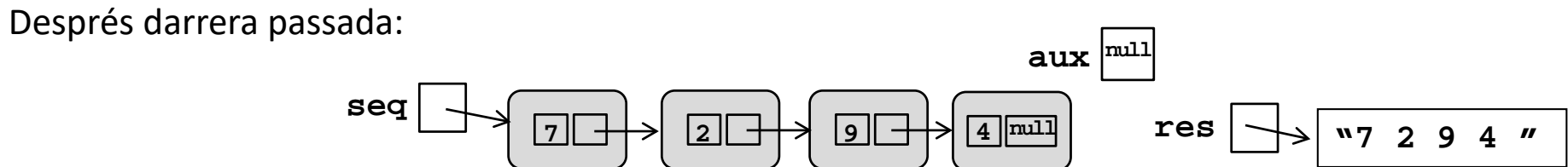
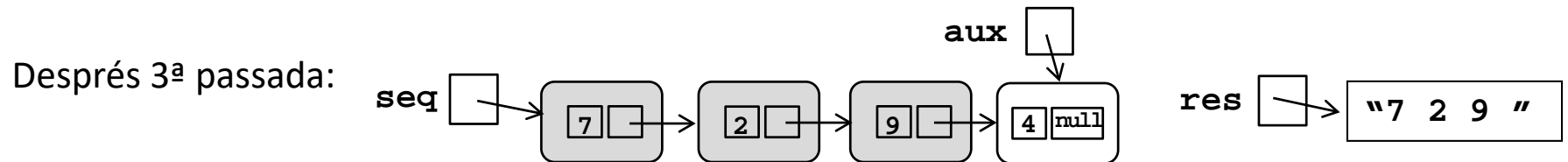
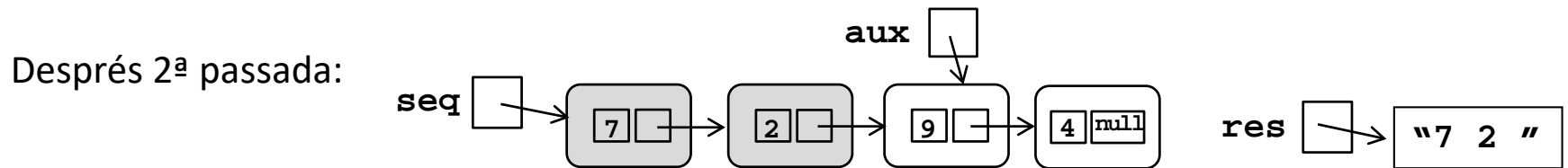
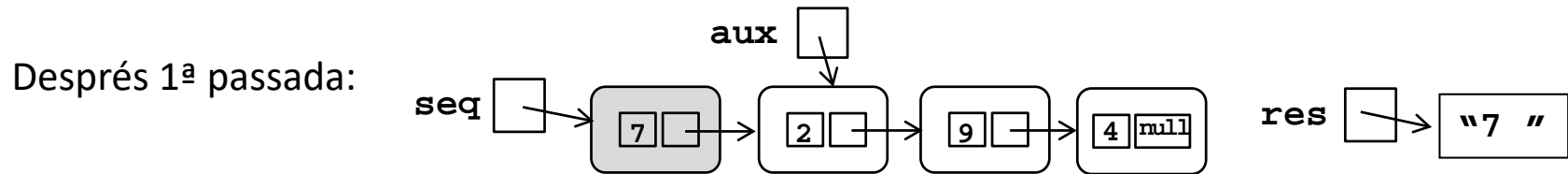
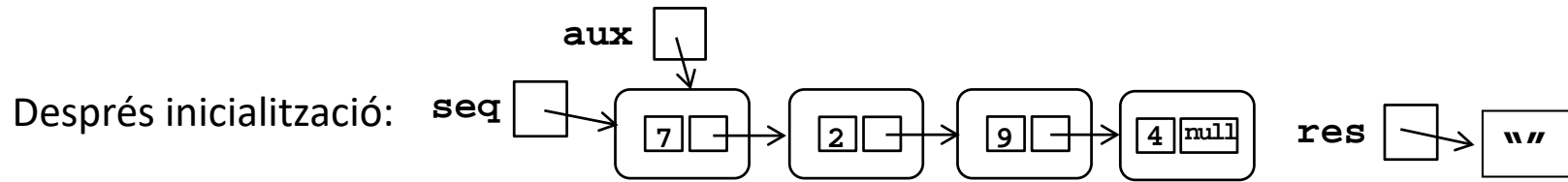
Després de la darrera passada:



- Traça exemple de `toArray`:



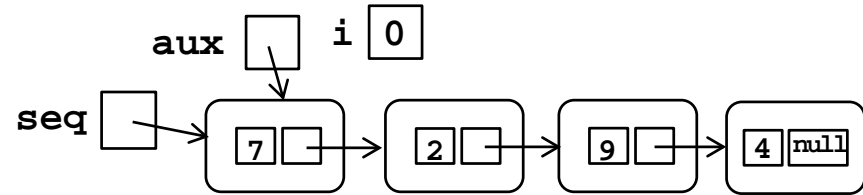
- Traça exemple de toString:



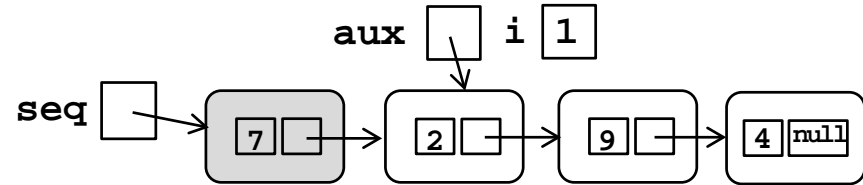
- Traça exemple de cercarPos:

a) Traça exemple amb **d** igual a 9:

Després de la inicialització del bucle:

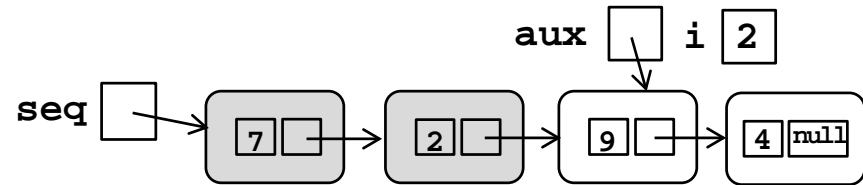


Després de la primera passada:



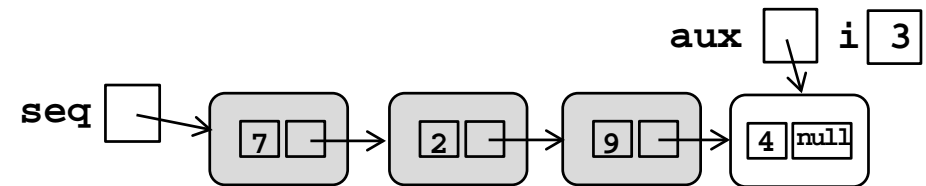
Después de la segona i última passada:

El mètode torna 2



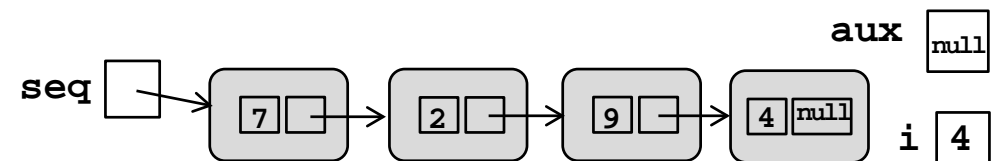
b) Traça exemple amb **d** igual a 5:

Després de la tercera passada:



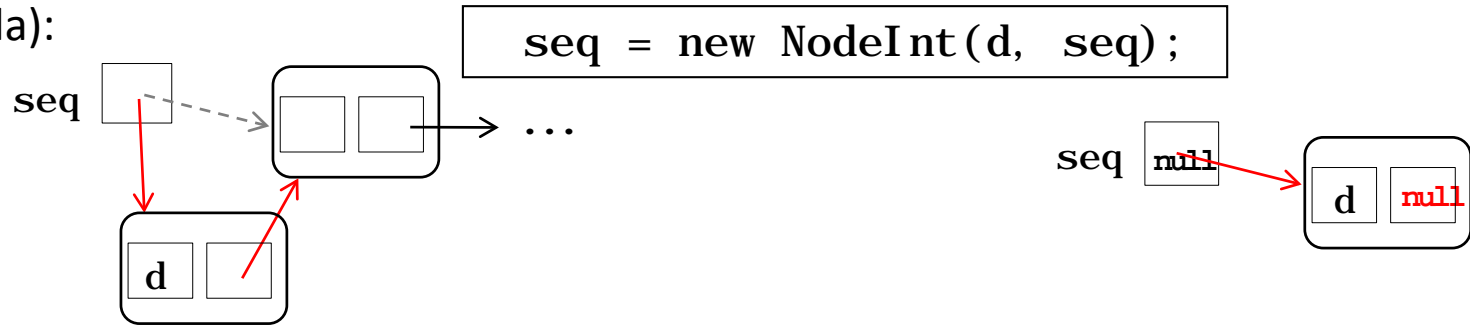
Després de la quarta i última passada:

El mètode torna -1

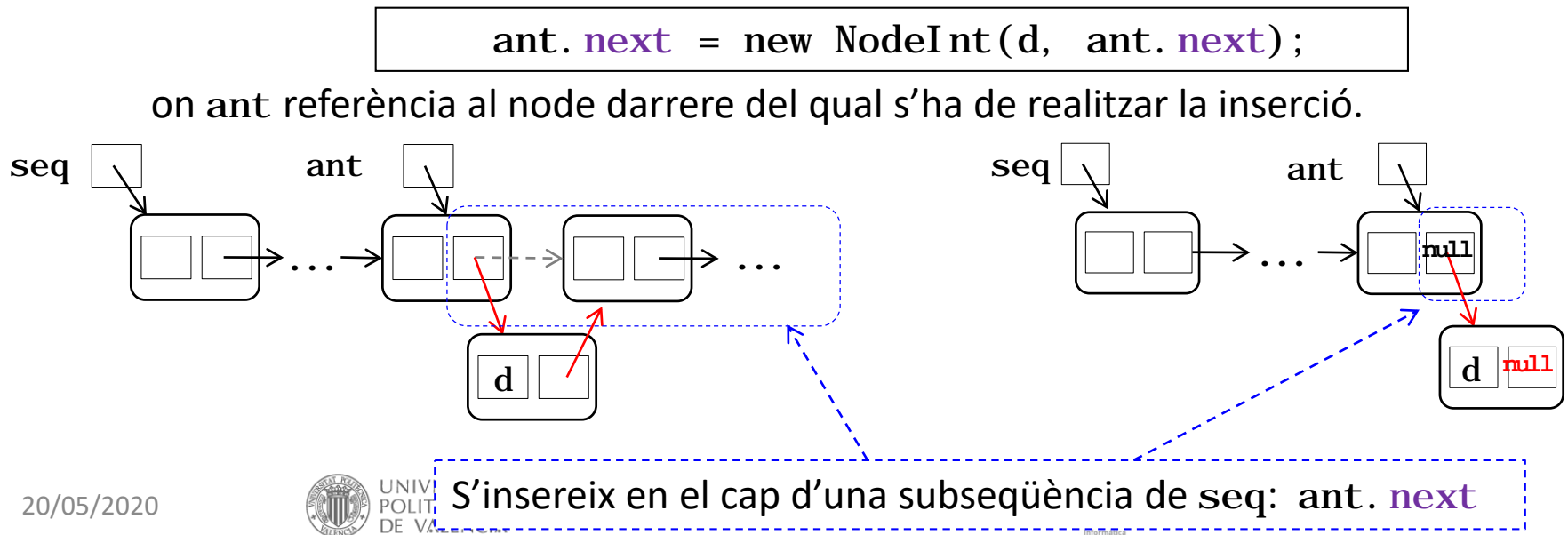


Seqüències enllaçades. Inserció

- Gràcies a l'ús explícit dels enllaços, la inserció d'una dada en una seqüència es pot resoldre sense realitzar cap moviment de les dades existents.
- Segons on s'ha de realitzar la inserció es poden donar dos casos:
 - El nou node s'insereix *en cap* o primera posició (inclou el cas d'inserir en una seqüència buida):



- El nou node s'insereix en qualsevol altra posició de `seq`, és a dir, **darrere d'**algun node (inclou el cas d'inserir darrere de l'últim):



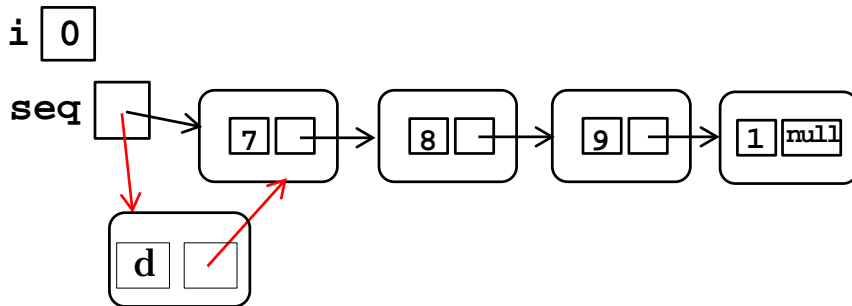
Seqüències enllaçades. Inserció

- Exemple.** Donada una seqüència i un índex $i \geq 0$, inserir la dada d en la posició i . Si l'índex sobrepassa la longitud de la seqüència, la inserció no es fa.

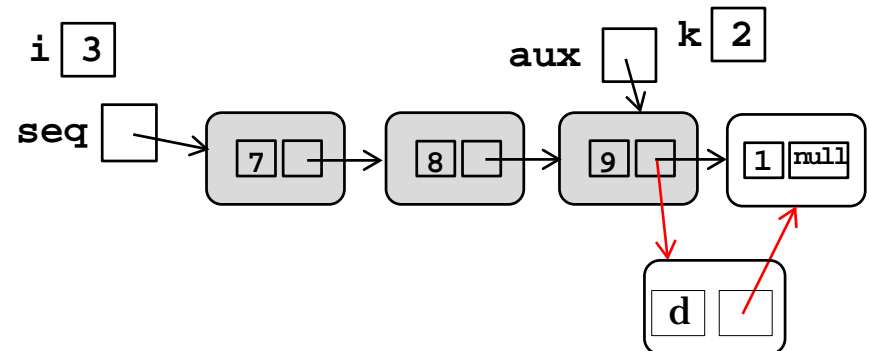
```
if (i == 0) { seq = new NodeInt(d, seq); }
else {
    NodeInt aux = seq; int k = 0;
    while (aux != null && k < i - 1) {
        aux = aux.next; k++;
    }
    if (aux != null) { // Èxit en la cerca
        aux.next = new NodeInt(d, aux.next);
    }
}
```

Casos:

a) $i == 0$: inserció en el cap.



b) $i > 0$: es busca el node $i - 1$, i si existeix, s'insereix el nou node a continuació.



Seqüències enllaçades. Inserció

- A l'encapsular en un mètode un codi com l'anterior, cal tenir compte amb `seq`, segons siga una variable en la pila o en el *heap*.
- **Exemple.** Escriure un **mètode estàtic** que, donada una seqüència `seq` i un índex $i \geq 0$, inserisca la dada `d` en la posició `i`. Si l'índex sobrepassa la longitud de la seqüència, la inserció no es fa.

seqüència resultant de la inserció

```
public static NodeInt insert(NodeInt seq, int d, int i) {  
    if (i == 0) { seq = new NodeInt(d, seq); }  
    else {  
        NodeInt aux = seq; int k = 0;  
        while (aux != null && k < i - 1) {  
            aux = aux.next; k++;  
        }  
        if (aux != null) { // Èxit en la cerca  
            aux.next = new NodeInt(d, aux.next);  
        }  
    }  
    // El paràmetre seq ha canviat,  
    // cal tornar el seu valor:  
    return seq;  
}
```

En execució, `seq` està en la pila del registre d'activació de la crida al mètode

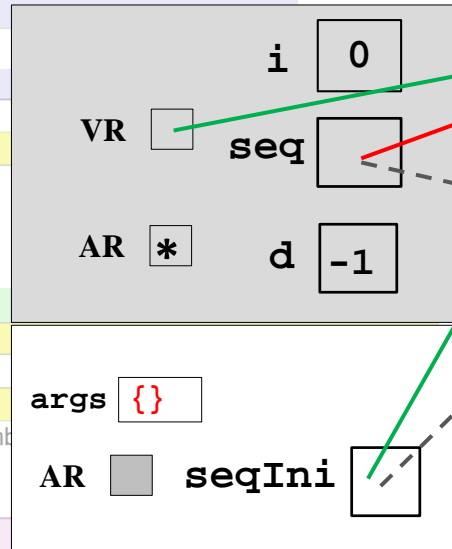
```
public class UtilLinked {
```

```
    public static NodeInt insert(NodeInt seq, int d, int i) {  
        if (i == 0) { seq = new NodeInt(d, seq); }  
        else {  
            ...  
        }  
        return seq;  
    }
```

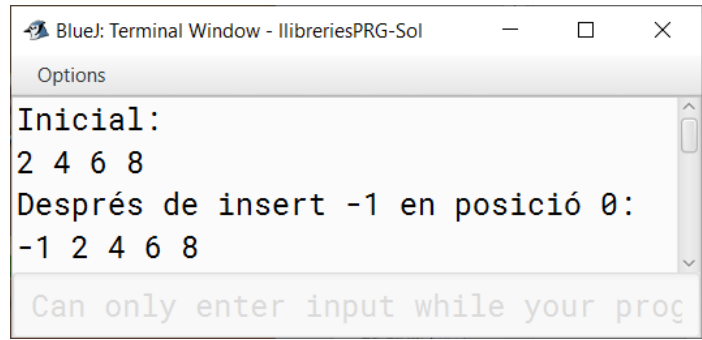
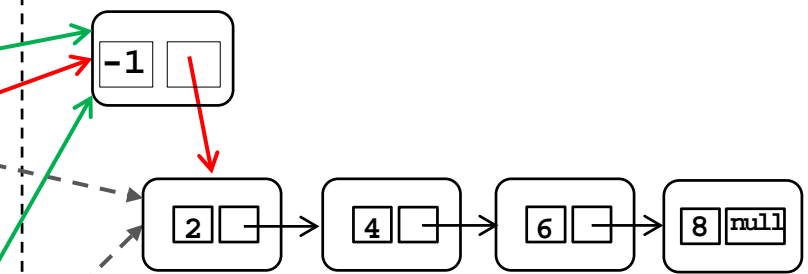
```
public class ProvaInsert {
```

```
    private ProvaInsert() { }  
  
    public static void main(String[] args) {  
        // Crea una seqüència enllaçada amb  
        // 4 primers números parells  
        NodeInt seqIni = null;  
        for (int i = 4; i >= 1; i--) {  
            seqIni = new NodeInt(i * 2, seqIni);  
        }  
        // Mostra la seqüència: 2 4 6 8  
        System.out.println("Inicial: \n" + UtilLinked.toString(seqIni));  
  
        // Insereix la dada -1 en la posició 0  
        seqIni = UtilLinked.insert(seqIni, -1, 0); *  
        // Mostra la seqüència: -1 2 4 6 8  
        System.out.println("Després de insert -1 en posició 0: \n"  
            + UtilLinked.toString(seqIni));  
    }  
    ...
```

pila



monticle (heap)



Seqüències enllaçades. Inserció

- **Exemple.** Suposar una classe amb un atribut, de nom `seq`, que és una seqüència enllaçada d'`int`. Escriure un **mètode d'instància** que, donat un índex $i \geq 0$ i una dada `d`, inserisca la dada `d` en la posició `i` de la seqüència `seq`. Si l'índex sobrepassa la longitud de la seqüència, la inserció no es fa.

```
public void insert(int d, int i) {  
    if (i == 0) { this.seq = new NodeInt(d, this.seq); }  
    else {  
        NodeInt aux = this.seq; int k = 0;  
        while (aux != null && k < i - 1) {  
            aux = aux.next; k++;  
        }  
        if (aux != null) { // Èxit en la cerca  
            aux.next = new NodeInt(d, aux.next);  
        }  
    }  
}
```

En execució, `seq` està en el heap, és un atribut de l'objecte al que s'aplica el mètode

```
public class SeqIntLinked {
    private NodeInt seq;

    /** Crea una SeqIntLinked buida. */
    public SeqIntLinked() { seq = null; }

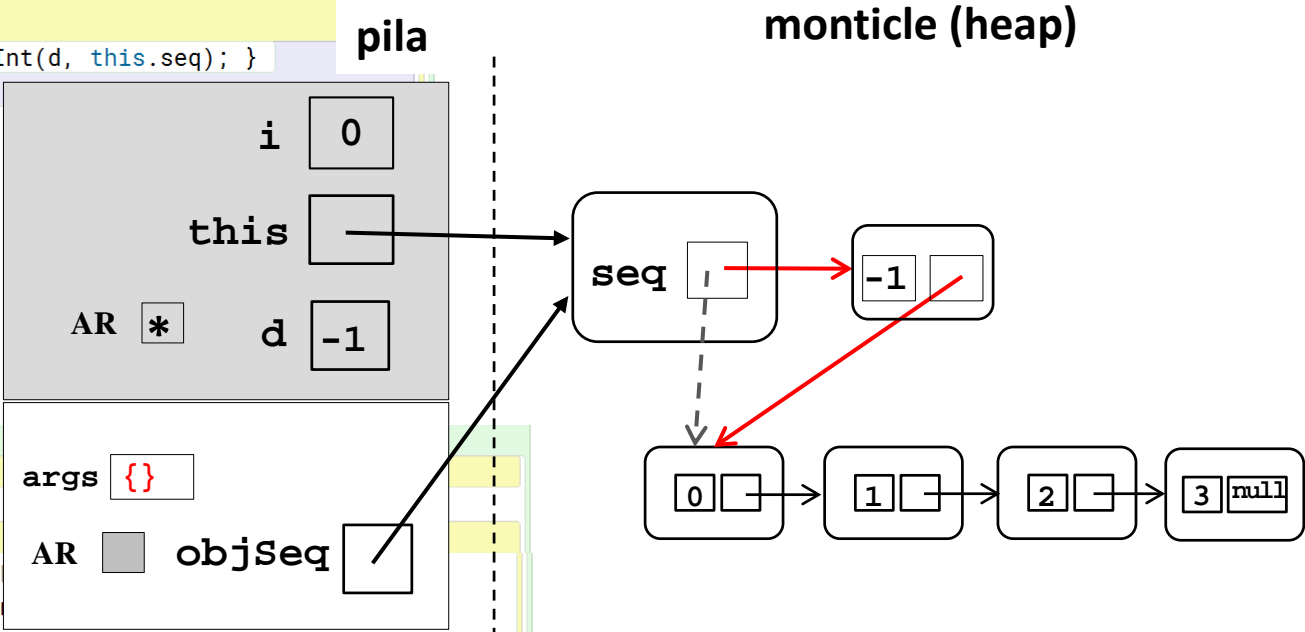
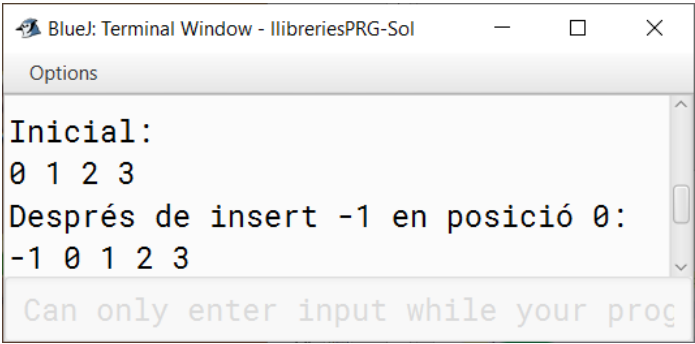
    /** Crea una SeqIntLinked en els enters [0..n-1] */
    public SeqIntLinked(int n) {
        for (int i = n - 1; i >= 0; i--) { seq = new NodeInt(i, seq); }
    }

    /** Insereix la dada d en la posició i (sent i >= 0)
     * de la seqüència seq.
     */
    public void insert(int d, int i) {
        if (i == 0) { this.seq = new NodeInt(d, this.seq); }
        else {
            ...
        }
    }
}
```

```
public class ProvaInsert {
    private ProvaInsert() { }

    public static void main(String[] args) {
        // Crea un objecte de tipus SeqIntLinked
        SeqIntLinked objSeq = new SeqIntLinked(4);
        // Mostra la seqüència: 0 1 2 3
        System.out.println("\nInicial: \n" + objSeq);

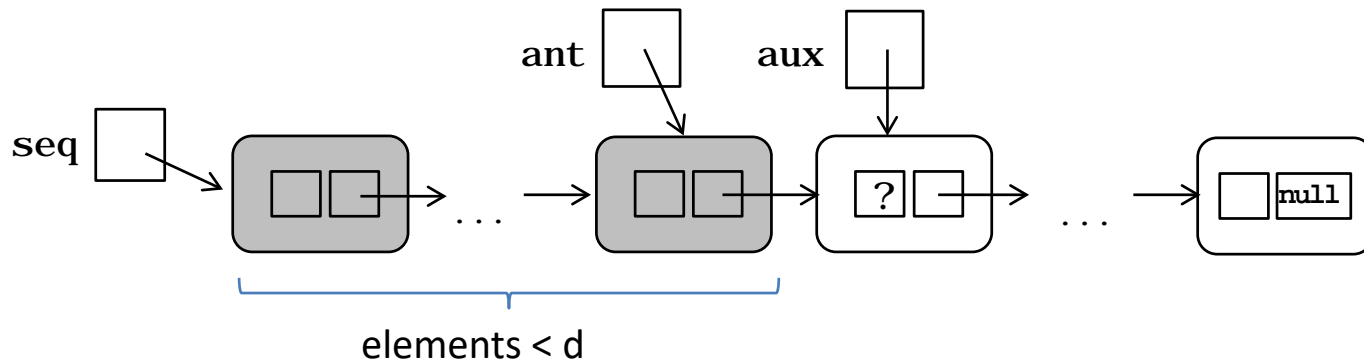
        // Insereix la dada -1 en la posició 0
        objSeq.insert(-1, 0);
        // Mostra la seqüència: -1 0 1 2 3
        System.out.println("Després de insert -1 en posició 0: \n" + objSeq);
    }
}
```



Seqüències enllaçades. Inserció

- **Exemple.** Donada una seqüència enllaçada tal que les seues dades estan **ordenades** de menor a major, inserir una dada **d** mantenint l'ordenació.

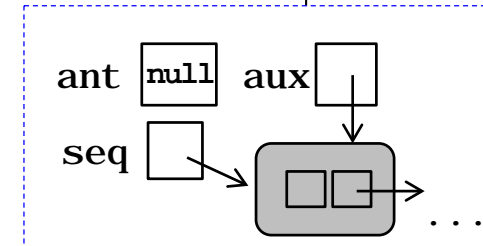
Cal que s'inserisca en el cap de la subseqüència de **seq** que continga als elements $\geq d$. Amb **aux** es busca el primer node $\geq d$. La variable **ant** referència en cada moment el node anterior o previ a **aux**:



Seqüències enllaçades. Inserció

- Exemple.** Donada una seqüència enllaçada tal que les seues dades estan ordenades de menor a major, inserir una dada d mantenint l'ordenació.

```
public static NodeInt insertSort(NodeInt seq, int d) {  
    NodeInt aux = seq,  
            ant = null; // el primer node no té  
                        // anterior definit  
    while (aux != null && aux.data < d) {  
        ant = aux; aux = aux.next;  
    }  
    if (aux == seq) { seq = new NodeInt(d, seq); } // Cas a)  
    else { ant.next = new NodeInt(d, aux); }       // Cas b)  
    return seq;  
}
```



Acabada la cerca:

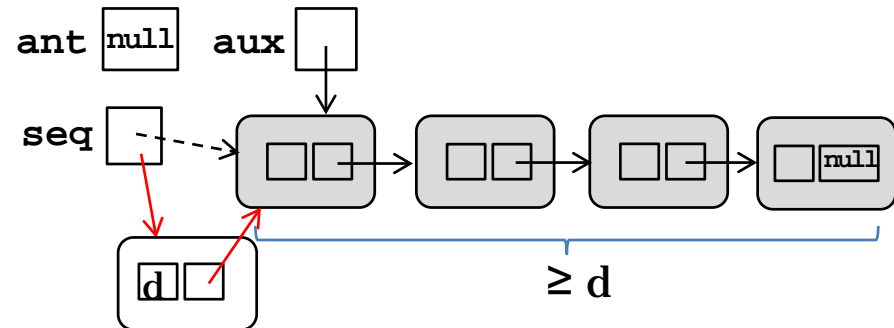
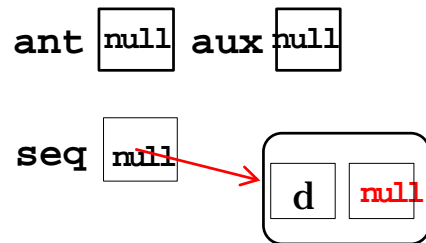
Cas a) $aux == seq$ (o $ant == null$), seq està buida o totes les seues dades són $\geq d$.

Cas b) $aux != seq$ (o $ant != null$), no totes les seues dades són $\geq d$. La inserció darrere de ant situa el nou node a continuació de totes les dades $< d$.

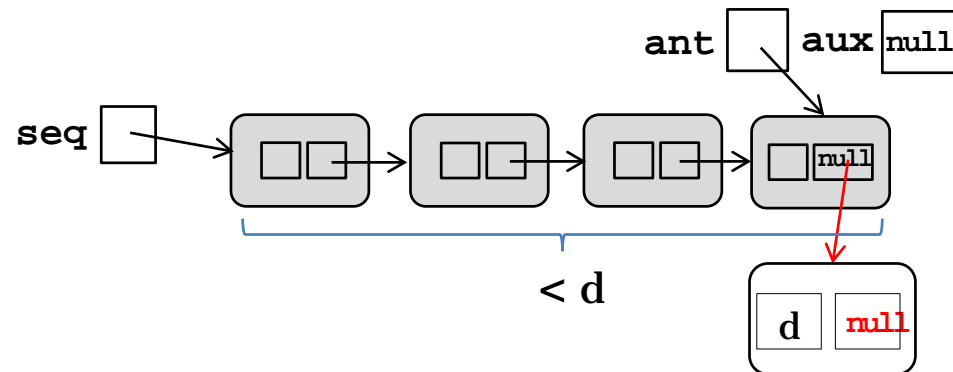
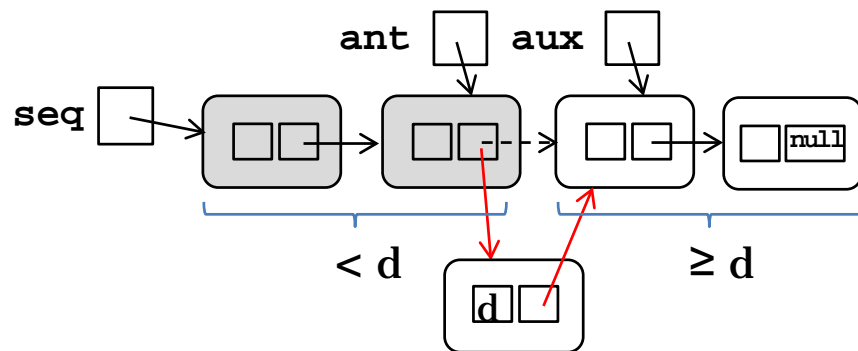
Seqüències enllaçades. Inserció

Acabada la cerca:

Cas a) $aux == seq$ (o $ant == null$), seq està buida o totes les seues dades són $\geq d$.

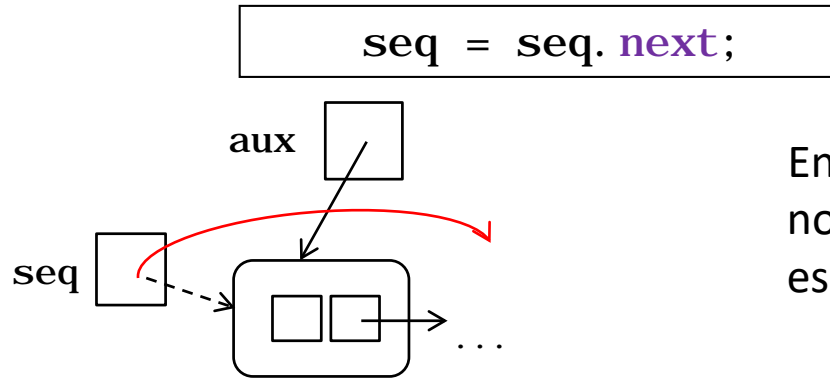


Cas b) $aux \neq seq$ (o $ant \neq null$), no totes les seues dades són $\geq d$. La inserció darrere de ant situa el nou node a continuació de totes les dades $< d$.



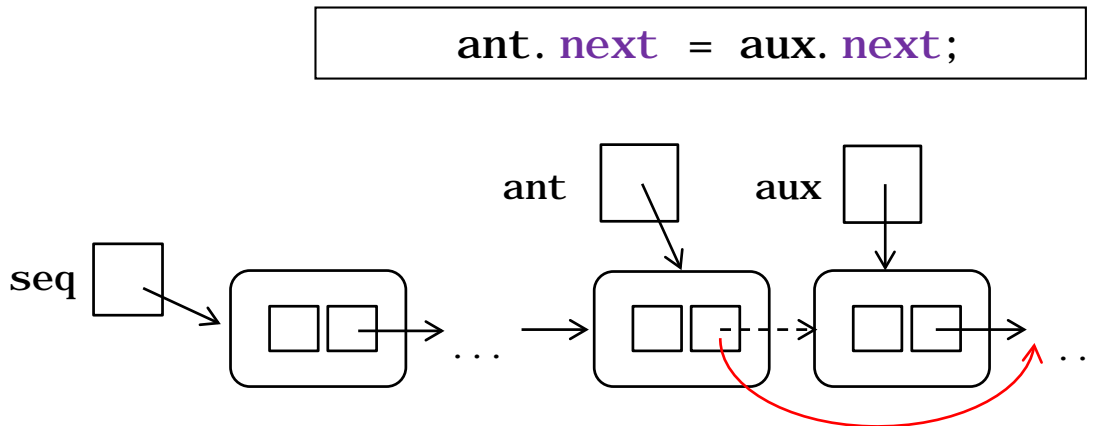
Seqüències enllaçades. Eliminació

- L'eliminació d'una dada es resol igualment sense realitzar cap moviment de les dades existents.
- Siga **aux** la referencia al node a eliminar. Es poden donar dos casos:
 - a) El node a eliminar és el primer de la seqüència.



En el cas particular de que només hi haguera un node, seq es faria null (seqüència buida).

- b) El node a eliminar té un anterior:



En el cas particular de que **aux** fora l'últim, **ant.next** es faria null (ant passaria a ser l'últim).

on **ant** és una referència al node anterior. S'elimina el node en el cap de la subseqüència **ant.next**.

Seqüències enllaçades. Eliminació

- **Exemple.** Eliminar, si existeix, la primera ocurrència d'una dada d. Si no existeix, no es fa res.

seqüència resultant de l'esborrament

```
public static NodeInt delete(NodeInt seq, int d) {  
    NodeInt aux = seq, ant = null;  
    while (aux != null && aux.data != d) {  
        ant = aux;  
        aux = aux.next;  
    }  
    if (aux != null) { // Èxit en la cerca  
        if (aux == seq) { // o ant == null  
            seq = aux.next;  
        }  
        else { ant.next = aux.next; }  
    }  
    // El paràmetre seq pot haver canviat,  
    // cal tornar el seu valor:  
    return seq;  
}
```

Seqüències enllaçades. Eliminació

- **Exemple.** Escriure un mètode que esborre totes les dades que estiguen per baix de cert `limit`.

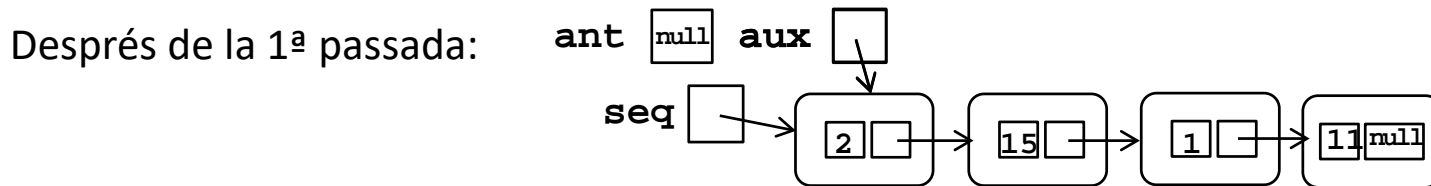
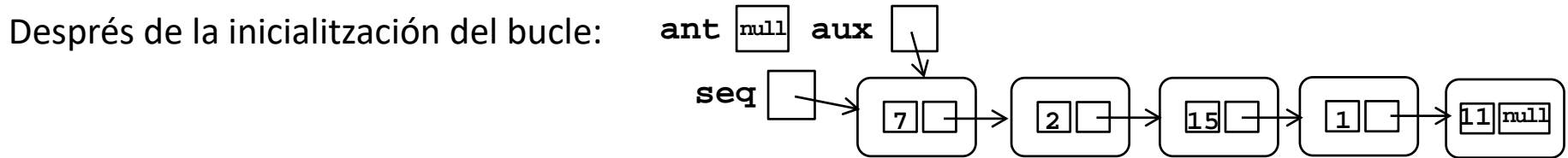
seqüència resultant de l'esborrament

```
public static NodeInt deleteUnder(NodeInt seq, int limit) {
    NodeInt aux = seq, ant = null;
    while (aux != null) {
        if (aux.data < limit) { // node a esborrar
            if (aux == seq) { seq = seq.next; }
            else { ant.next = aux.next; }
            // ant no s'actualitza
        }
        else { ant = aux; }

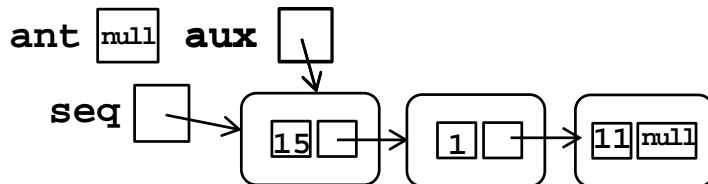
        aux = aux.next;
    }
    return seq;
}
```

Seqüències enllaçades. Eliminació

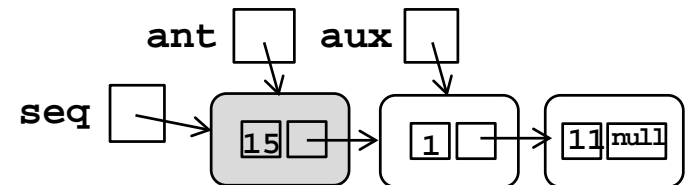
- Traça exemple de **deleteUnder** amb un valor de **limit** igual a 10.



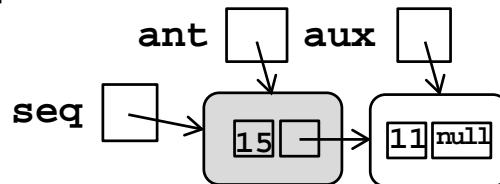
Després de la 2ª passada:



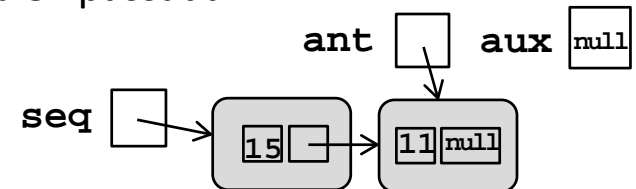
Després de la 3ª passada:



Després de la 4ª passada:



Després de la 5ª passada:

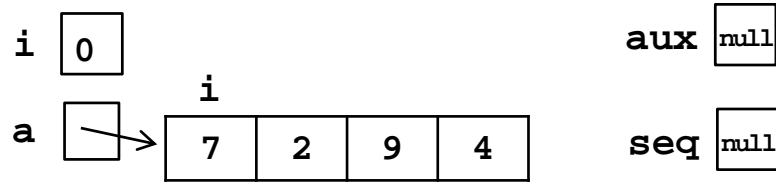


Seqüències enllaçades. Exercicis

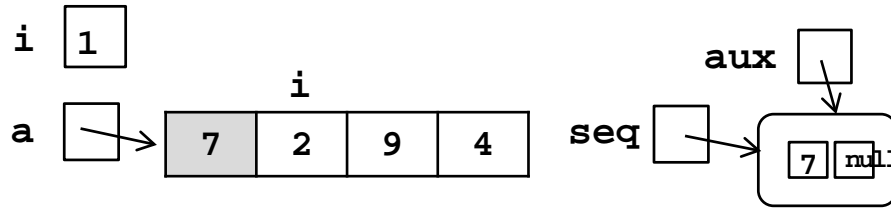
- **Completa** en la classe **UtilLinked**, del paquet **linear** del projecte BlueJ **llibreriesPRG**, els mètodes següents:
 - `public static NodeInt toSeq(int[] a)`
que torna una seqüència enllaçada amb els elements d'un array **a** donat.
 - `public static NodeInt invertir(NodeInt seq)`
que inverteix l'ordre dels elements d'una seqüència donada (amb cost lineal).
 - `public static NodeInt menorsQue(NodeInt seq, int e)`
que torna una seqüència enllaçada amb els elements menors que **e** i en el mateix ordre que apareixen en una seqüència donada **seq**. El cost serà lineal amb la longitud de **seq**.
- **Completa** la classe **TestUtilLinked**, del paquet **linear** del projecte BlueJ **llibreriesPRG**, seguint els comentaris. Et servirà per comprovar la correcció dels mètodes de la classe **UtilLinked**.

- **Traça exemple de toSeq (recorregut **ascendent** del array)**

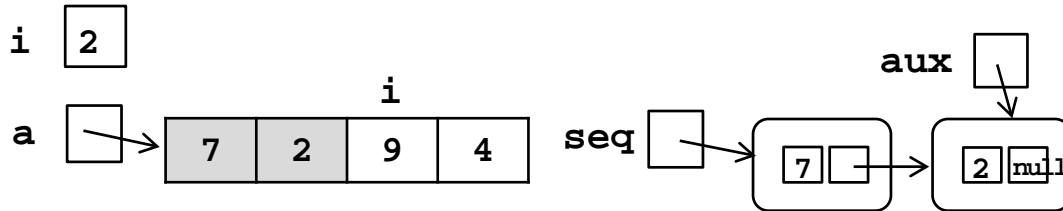
Després inicialització:



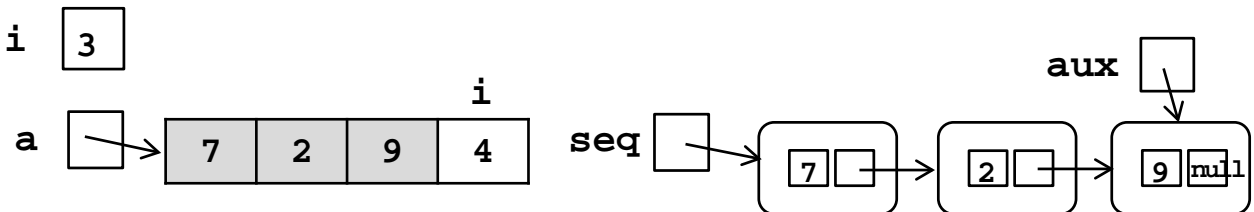
Després 1^a passada:



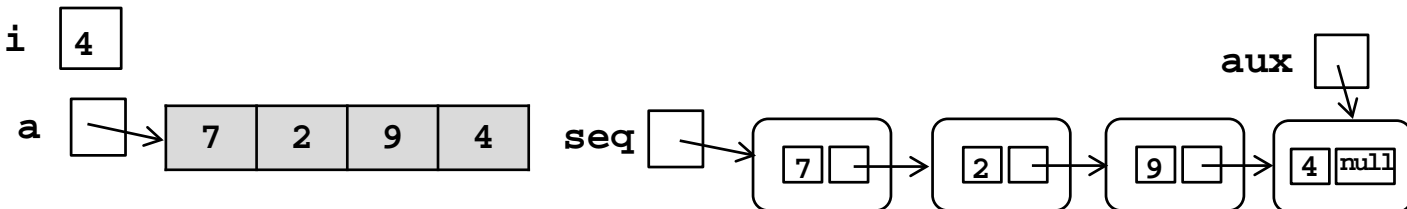
Després 2^a passada:



Després 3^a passada:

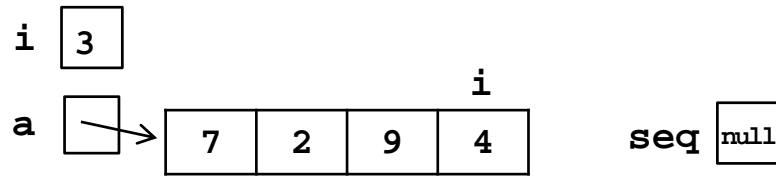


Després darrera passada:

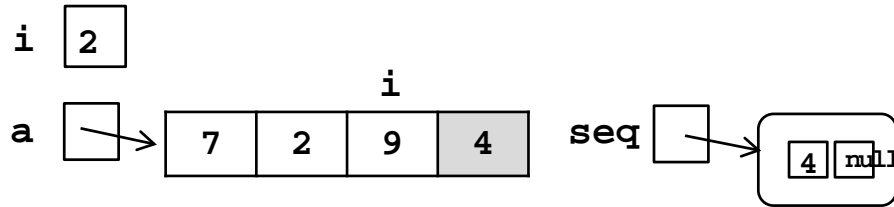


- Traça exemple de **toSeq** (recorregut **descendent** del array)

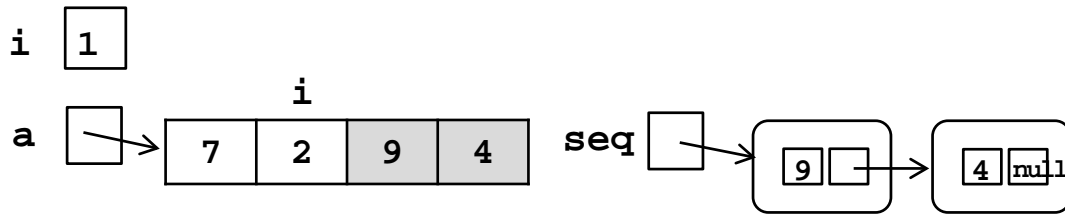
Després inicialització:



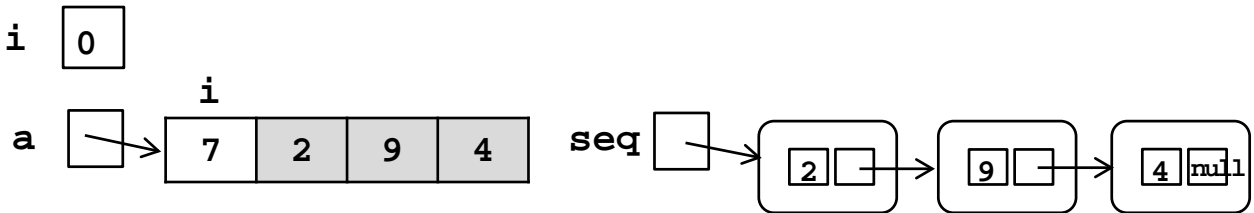
Després 1^a passada:



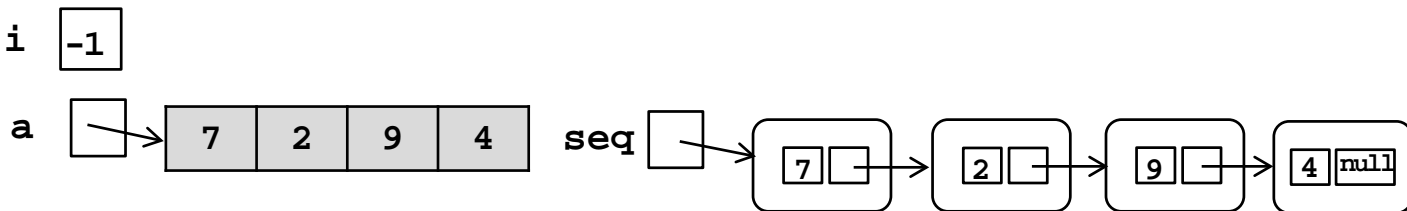
Després 2^a passada:



Després 3^a passada:

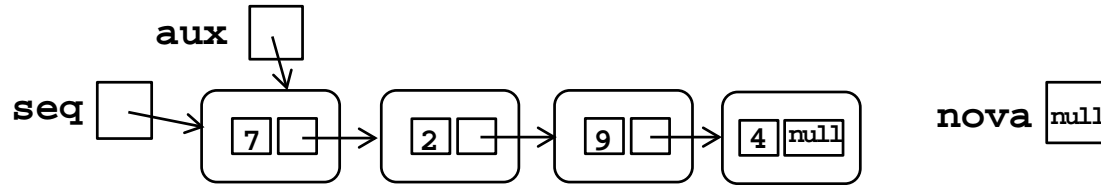


Després darrera passada:

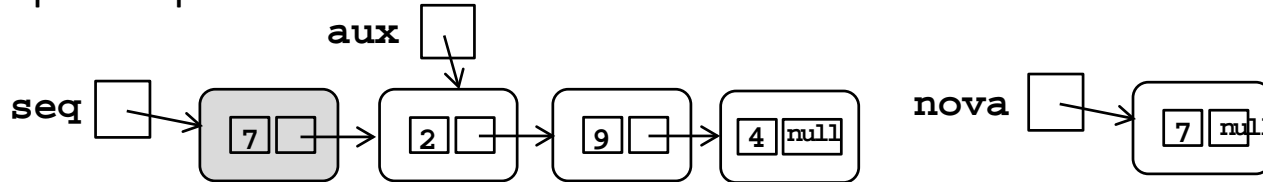


- **Traça exemple d'invertir:**

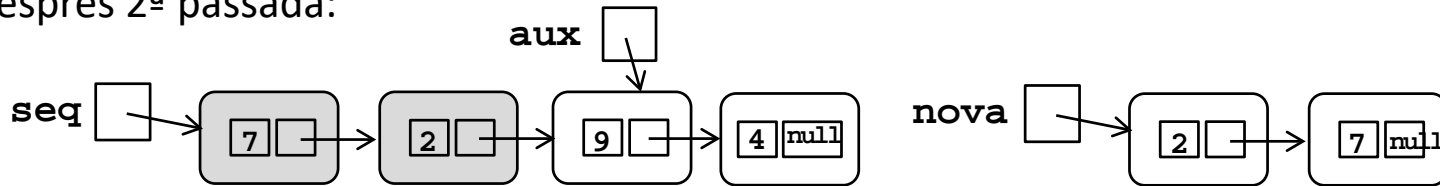
Després inicialització:



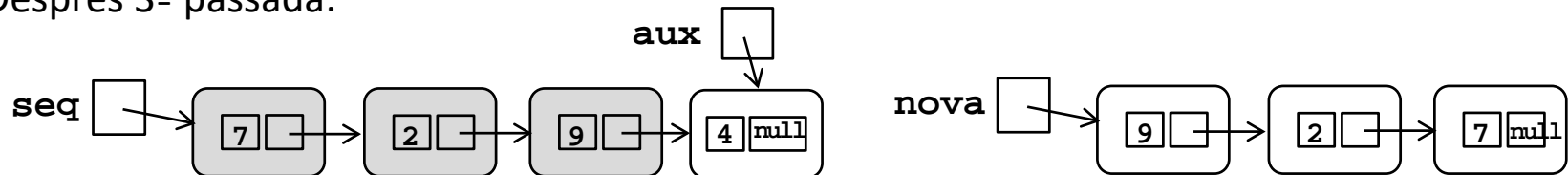
Després 1ª passada:



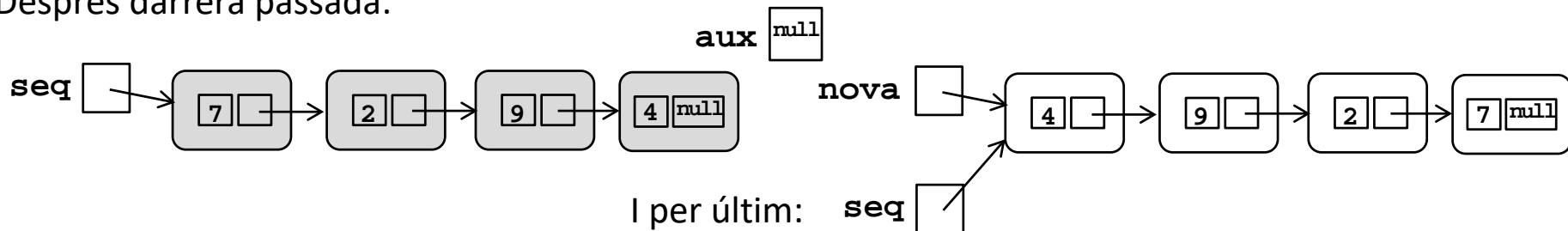
Després 2ª passada:



Després 3ª passada:

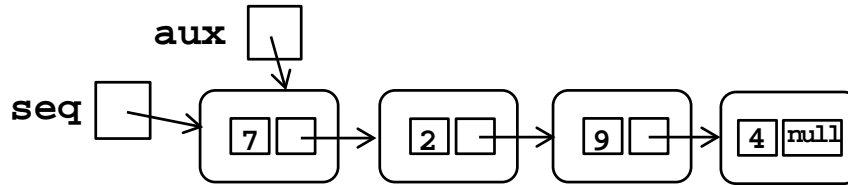


Després darrera passada:

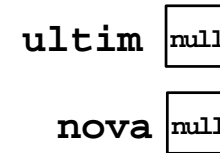
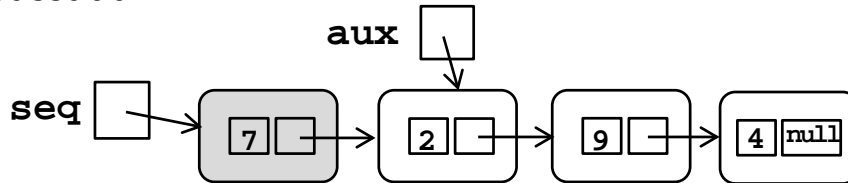


- Traça exemple de `menorsQue` amb `e` igual a 7:

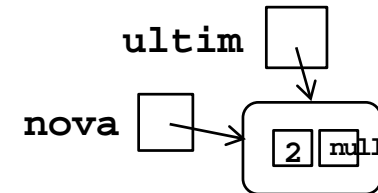
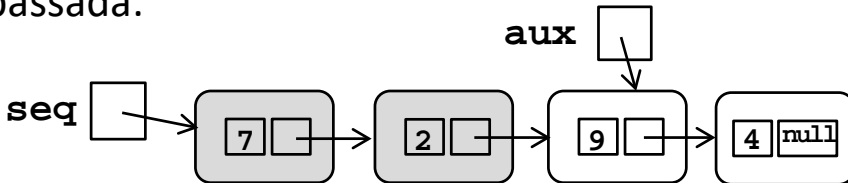
Després inicialització:



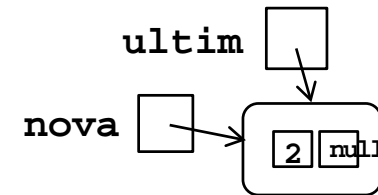
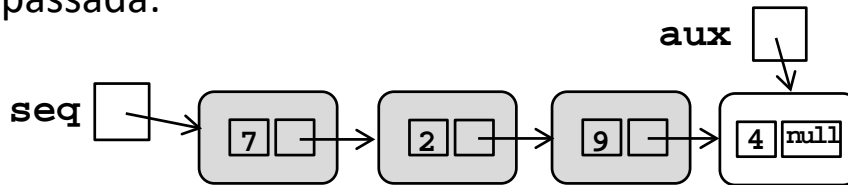
Després 1ª passada:



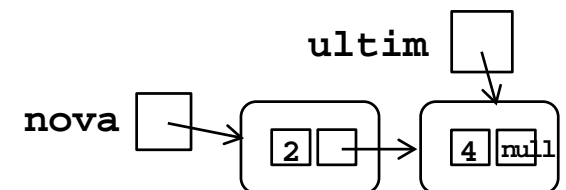
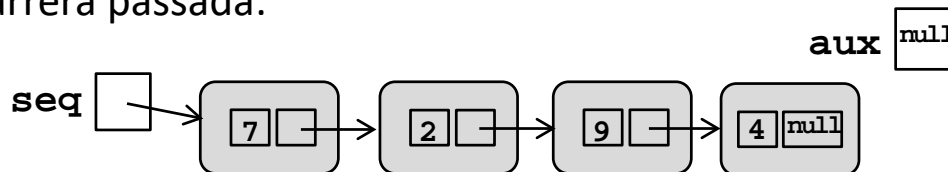
Després 2ª passada:



Després 3ª passada:

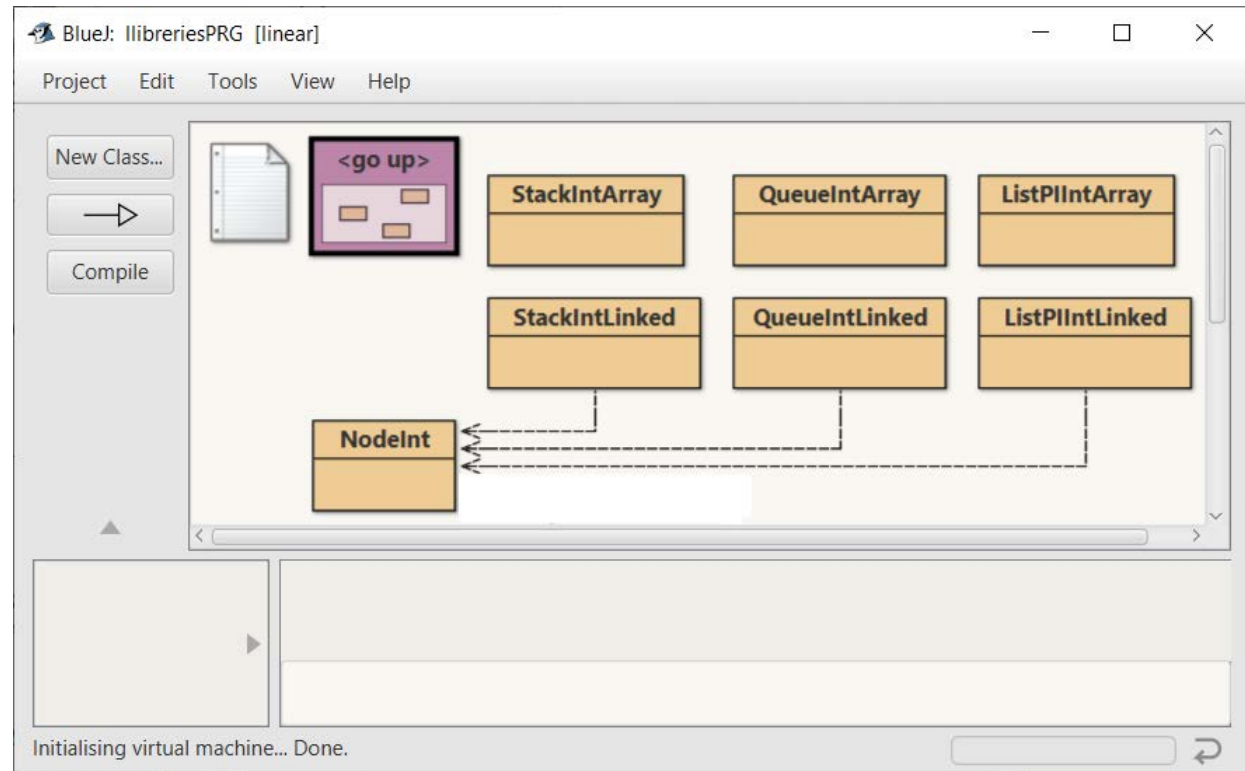
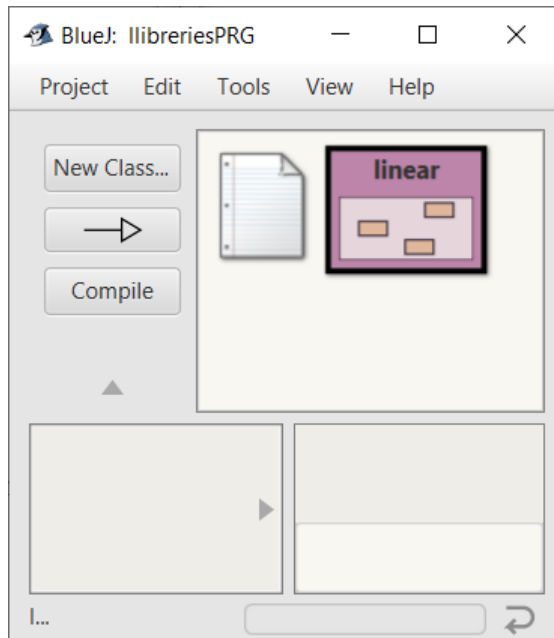


Després darrera passada:



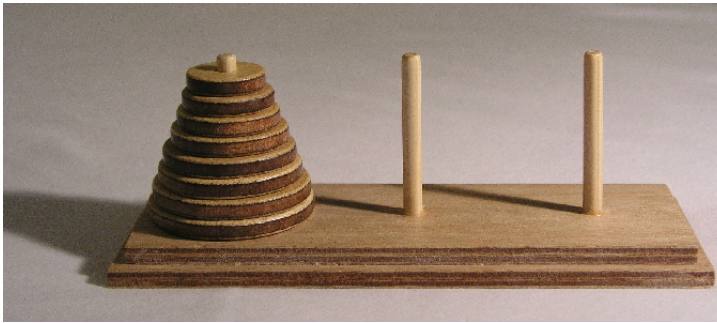
Representació enllaçada de tipus lineals

- La implementació enllaçada de **Stack**, **Queue** i **ListPI** amb dades de tipus enter es basarà en la classe **NodeInt**.
- S'inclouran totes elles en un mateix paquet **linear**.
- La classe **NodeInt** i les seues components, atributs i mètodes, seran *friendly*: seran accessibles per a les classes del paquet, `private` per a la resta.
- El paquet **linear** es completarà amb la implementació de **Stack**, **Queue** i **ListPI** d'`int`, basada en arrays.



Piles

- Una *pila* (*stack*) és una seqüència en la que l'accés al primer element es realitza seguint un criteri *LIFO* (*Last In First Out*).
 - Els elements d'una pila sempre s'eliminen d'ella en ordre invers al que varen ser col.locats, de manera que l'últim element en entrar és el primer en sortir, i a l'inrevés, el primer element en entrar és l'últim en sortir.
- Exemples de piles:



- Pila de registres d'activació

n = 0 r = 1	fact
n = 1 r = ?	fact
n = 2 r = ?	fact
n = 3 r = ?	fact
n = 4 r = ?	fact
f = ?	main

- Els compiladors utilitzen piles per tal d'avaluar expressions aritmètiques.
- Un algorisme per trobar un camí a través d'un laberint. Quan l'algorisme troba una intersecció, empila la ubicació en la pila i, a continuació, explora la primera branca. Si aquesta branca és un punt mort, torna a la ubicació del cim de la pila i explora la següent branca no provada. Si totes les branques són llocs sense eixida, desempila la ubicació de la pila, i en el cim queda una intersecció trobada prèviament.

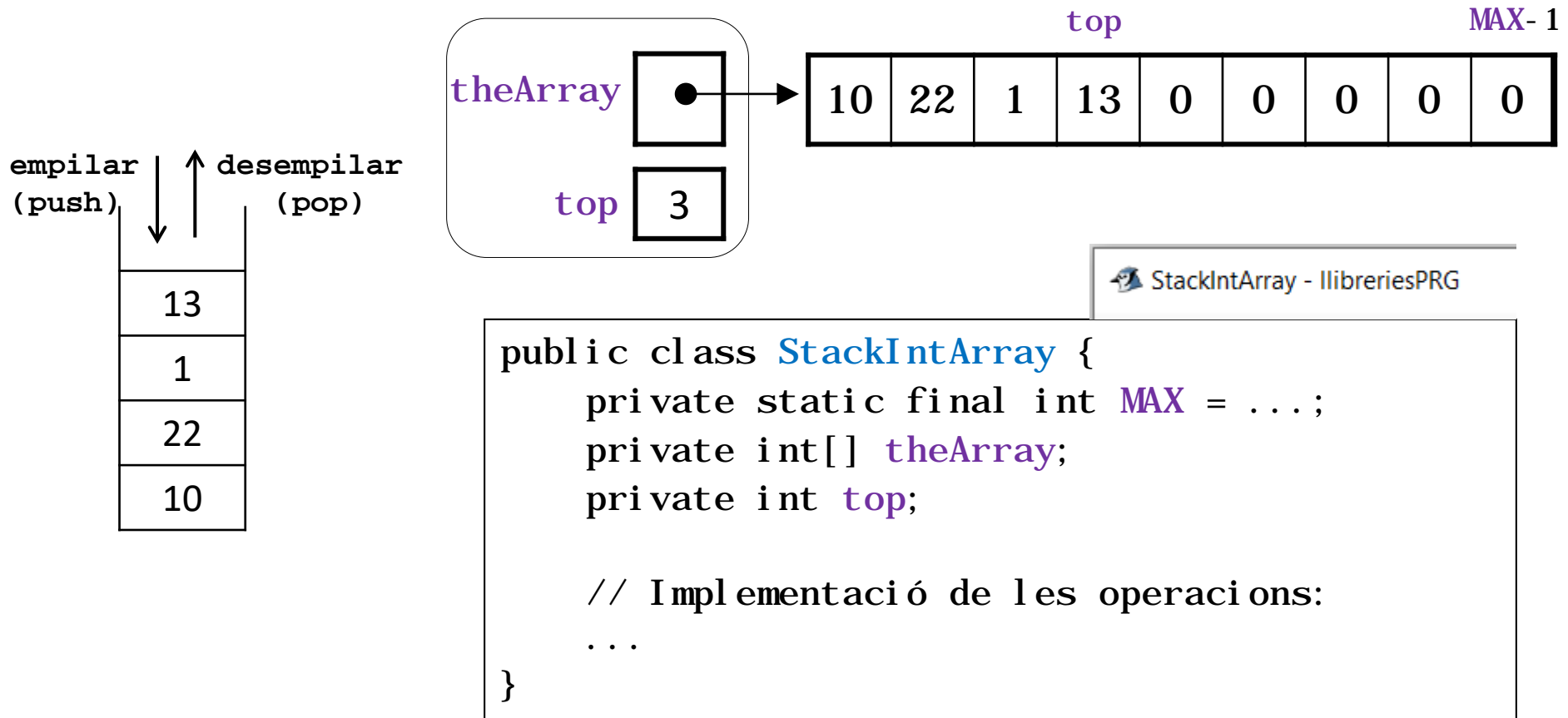
Piles - Operacions

Operació	Descripció
<code>public Stack()</code>	Crea una nova Stack buida.
<code>public void push(Ti pus element)</code>	Empila (<i>push</i>) l'element sobre la Stack.
<code>public Ti pus pop()</code>	Desempila (<i>pop</i>) l'element del cim de la Stack i el torna. Llança <code>NoSuchElementException</code> si la Stack està buida.
<code>public Ti pus peek()</code>	Torna (sense desempilar-lo) l'element del cim (<i>top</i>) de la Stack. Llança <code>NoSuchElementException</code> si la Stack està buida.
<code>public boolean empty()</code>	Torna <code>true</code> si la Stack està buida i <code>false</code> en cas contrari.
<code>public int size()</code>	Torna el nombre d'elements de la Stack.

Es pot aconseguir implementar aquestes operacions amb cost constant.

Piles – Implementació amb arrays

- Una pila (**Stack**) pot implementar-se usant un array (**theArray**) que emmagatzeme les dades, un índex (**top**) que marque el punt d'accés a la pila i una constant que definisca la dimensió màxima de l'array (**MAX**).
- Stack amb dades de tipus **int**



Piles – Implementació amb arrays

- Operació constructora `StackIntArray`: Crea l'array i inicialitza el punt d'accés a -1, indicant que la pila està buida.

```
public StackIntArray() {  
    theArray = new int[MAX];  
    top = -1;  
}
```

- Operació `push` (empilar):

```
public void push(int x) {  
    if (top + 1 == theArray.length) { duplicateArray(); }  
    top++;  
    theArray[top] = x;  
}
```

```
private void duplicateArray() {  
    int[] aux = new int[2 * theArray.length];  
    for (int i = 0; i < theArray.length; i++) {  
        aux[i] = theArray[i];  
    }  
    theArray = aux;
```


Piles – Implementació amb arrays

- Operacions **pop** (desempilar) i **peek** (consultar quin és el cim): llancen l'excepció *unchecked NoSuchElementException* si la pila està buida.

```
public int pop() {  
    if (top < 0) {  
        throw new NoSuchElementException("Empty stack");  
    }  
    int x = theArray[top];  
    top--;  
    return x;  
}  
  
public int peek() {  
    if (top < 0) {  
        throw new NoSuchElementException("Empty stack");  
    }  
    return theArray[top];  
}
```

- Operació consultora **empty**:

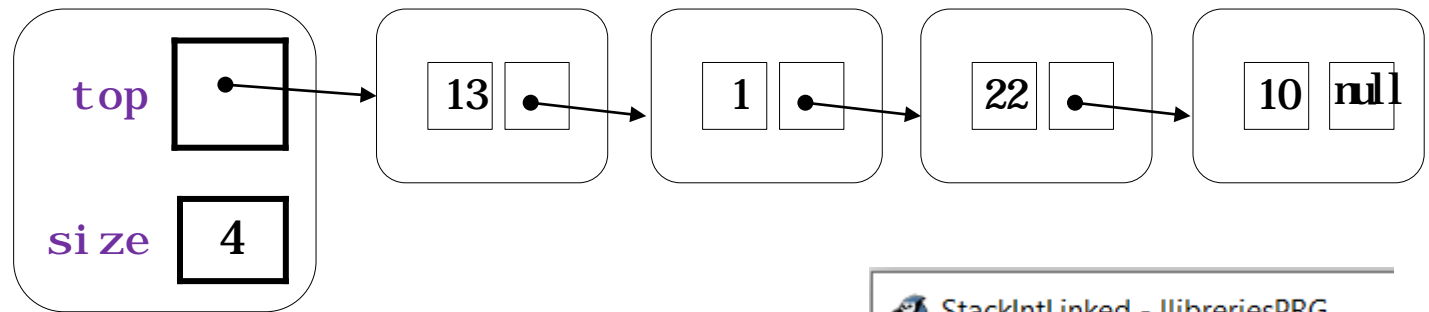
```
public boolean empty() { return top == -1; }
```


- Operació consultora **size**:

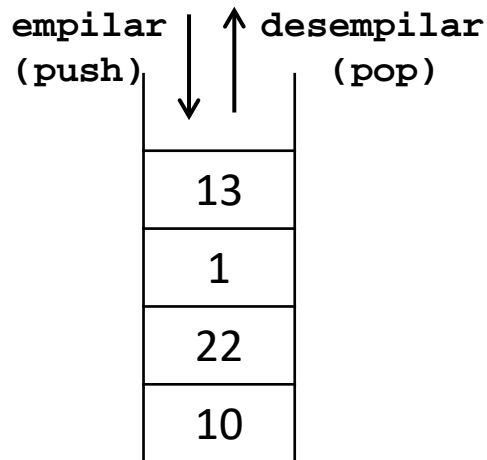
```
public int size() { return top + 1; }
```

Piles – Implementació enllaçada

- En una **implementació enllaçada** d'una pila, el cim és el primer element de la mateixa (la seua referència) i es representa mitjançant un atribut anomenat **top**. A més, s'afegeix un atribut **size** que representa el nombre d'elements de la pila.



 StackIntLinked - llibreriesPRG

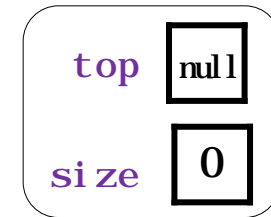


```
public class StackIntLinked {  
    private NodeInt top;  
    private int size;  
  
    // Implementació de les operacions:  
    ...  
}
```

Piles – Implementació enllaçada

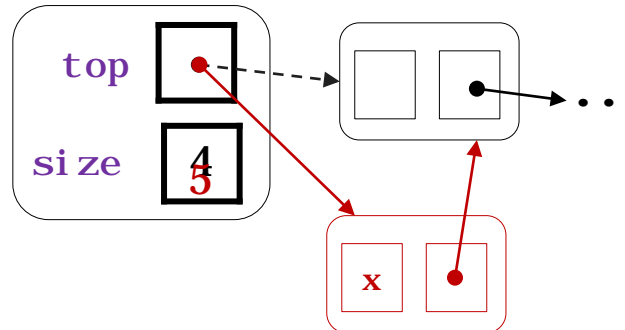
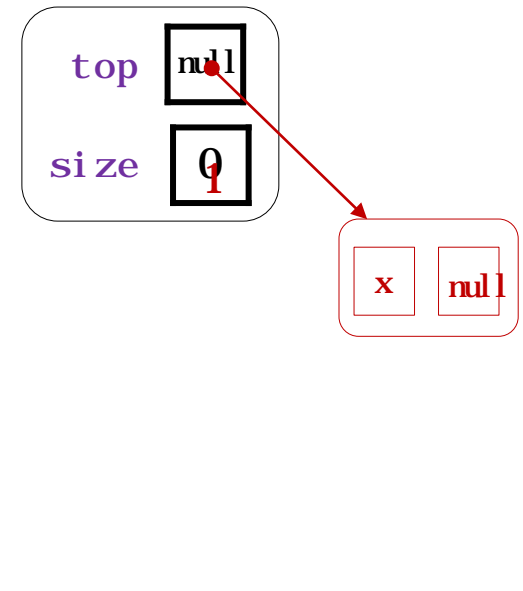
- Operació constructora **StackIntLinked**: Assigna **null** a la referència **top** i inicialitza **size** a 0.

```
public StackIntLinked() {  
    top = null;  
    size = 0;  
}
```



- Operació **push** (empilar):

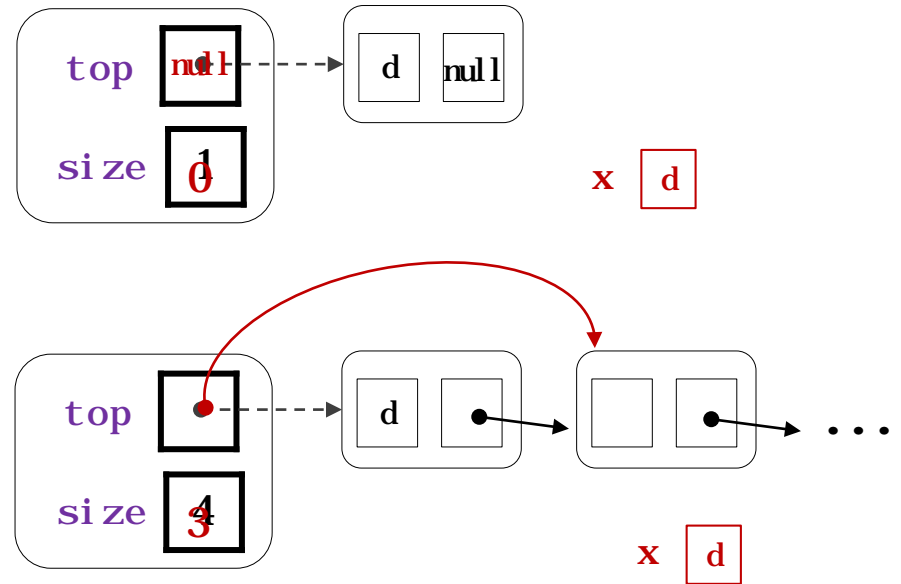
```
public void push(int x) {  
    top = new NodeInt(x, top);  
    size++;  
}
```



Piles – Implementació enllaçada

- Operacions **pop** (desempilar) i **peek** (consultar quin és el cim): llancen l'excepció *unchecked NoSuchElementException* si la pila està buida.

```
public int pop() {  
    if (top == null) {  
        throw new NoSuchElementException("Empty stack");  
    }  
    int x = top.data;  
    top = top.next;  
    size--;  
    return x;  
}
```



```
public int peek() {  
    if (top == null) {  
        throw new NoSuchElementException("Empty stack");  
    }  
    return top.data;  
}
```


Piles – Implementació enllaçada

- Operació consultora **empty**:

```
public boolean empty() {  
    return top == null;  
    // o return size == 0;  
}
```

- Operació consultora **size**:

```
public int size() { return size; }
```

 StackIntArray - llibreriesPRG

 StackIntLinked - llibreriesPRG

- Revisa** els mètodes **equals(Object)** i **toString()** de les classes **StackIntLinked** i **StackIntArray**
- Completa** la classe **TestStackInt** del paquet **linear** seguint els comentaris. Et servirà per provar els mètodes de les classes **StackIntLinked** i **StackIntArray**.

Piles – Comparació d'implementacions

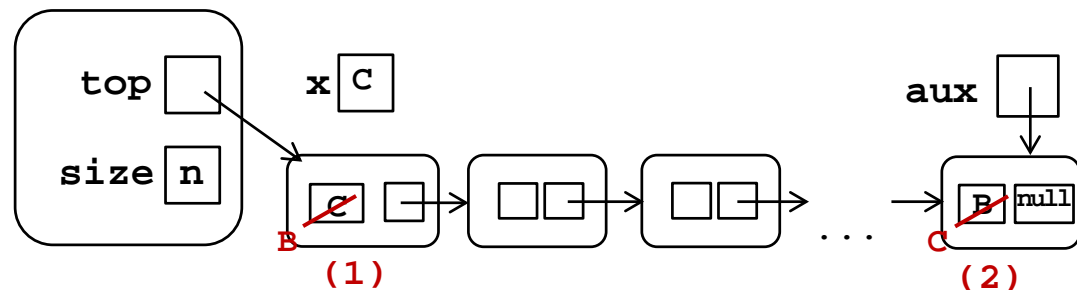
- La **complexitat temporal** de totes les operacions en les dues representacions estudiades és independent de la talla del problema: $\Theta(1)$.
- Pel que fa a la **complexitat espacial**, la implementació amb arrays presenta l'inconvenient de l'estimació del tamany màxim de l'array i la reserva d'espai que en molts casos no s'utilitzarà.
- Per altra banda, la representació enllaçada requereix un espai de memòria addicional per als enllaços.
- Aquest consum addicional d'espai no tindrà massa importància si el tipus de les components de la pila és relativament menut, com és el cas d'una pila d'ints, o de piles d'objectes (en eixe cas les components són referències).

Piles – Exercici

- Afegir a la classe **StackIntLinked** un mètode **topBase** que intercanvie els elements que apareixen en el cim (*top*) i la base de la pila. Si la pila té com a molt un element, ha de quedar igual.

```
/** Intercanvia els elements del cim i de la base de la pila.
 * Si la pila està buida o té un element, no canvia.
 */
public void topBase() {
    if (this.size > 1) {
        NodeInt aux = /* COMPLETAR */
        int x = aux.data;
        // Cerca del darrer element (node on aux.next és null):
        while (/* COMPLETAR */) { aux = aux.next; }
        // en acabar el bucle, aux != null
        // Actualitzar la dada de top (1)
        /* COMPLETAR */
        // Actualitzar la dada de aux (2)
        /* COMPLETAR */
    }
}
```

- Quan hi ha dos o més elements:

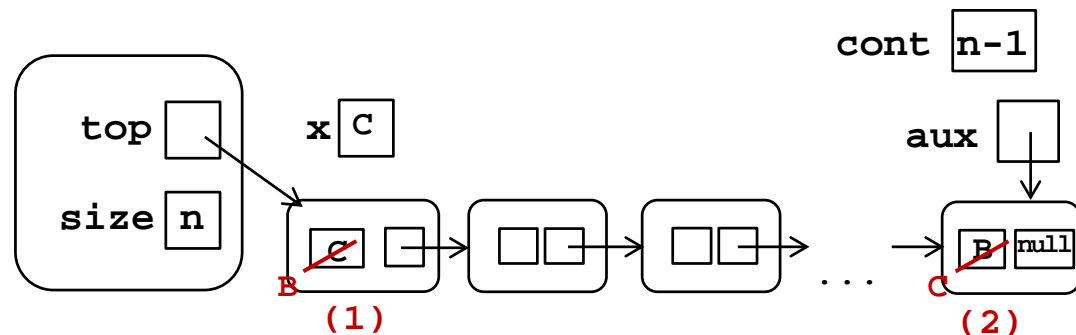


Piles – Exercici

- En el mètode anterior, la cerca del darrer element es pot basar en la talla de la pila

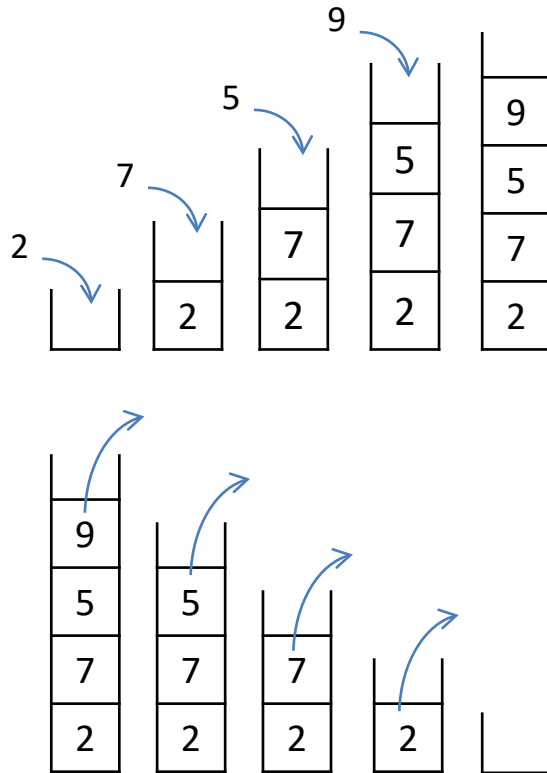
```
/** Intercanvia els elements del cim i de la base de la pila.
 * Si la pila està buida o té un element, no canvia.
 */
public void topBase() {
    if (this.size > 1) {
        NodeInt aux = /* COMPLETAR */
        int x = aux.data; int cont = 0;
        // Cerca del darrer element:
        while (/* COMPLETAR */) { aux = aux.next; cont++; }
        // en acabar el bucle, aux != null
        // Actualitzar la dada de top (1)
        /* COMPLETAR */
        // Actualitzar la dada de aux (2)
        /* COMPLETAR */
    }
}
```

- Quan hi ha dos o més elements:



Piles – Exercici

- En una pila, la política de gestió de les dades (mètodes *push* i *pop* en el cim com úniques operacions per a afegir o eliminar elements) implica que necessàriament els elements són accessibles en ordre invers al de la seua inserció en la pila. El següent programa ho aprofita per a escriure per pantalla una seqüència d'enters inversament a com s'han introduït per teclat.



```
package usaLinear;
```

```
import java.util.Scanner;  
import linear.StackIntLinked;
```

```
public class InvertirOrdre {
```

```
    private InvertirOrdre() { }
```

```
    public static void main(String[] args) {
```

```
        Scanner teclat = new Scanner(System.in);
```

```
        System.out.println("Introdueix una seqüència d'enters >=0."  
                            + " Escriu un valor negatiu per acabar.");
```

```
        // Crear un StackIntLinked buida
```

```
        int n = teclat.nextInt();
```

```
        while (n >= 0) {  
            // Empilar la dada n  
            n = teclat.nextInt();  
        }
```

```
        System.out.println("Seqüència en sentit invers:");
```

```
        while (/* La pila no estiga buida */) {  
            // Mostrar per pantalla el cim i desempilar  
        }
```

```
    }
```

BlueJ: Terminal Window - llibreriaPRG-Sol

Options

Introdueix una seqüència d'enters >=0. Escriu un valor negatiu per acabar.

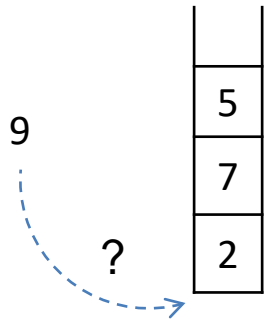
2 7 5 9 -1

Seqüència en sentit invers:

9 5 7 2

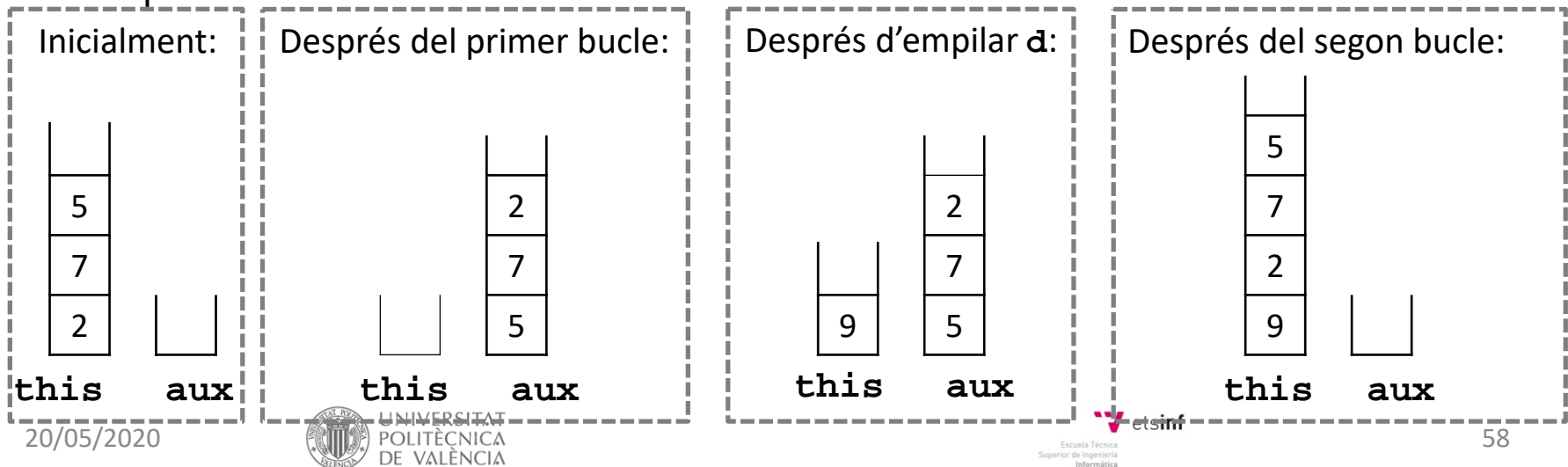
Piles – Example

- Escriu en la classe **StackIntLinkedPlus** (derivada de **StackIntLinked**) un mètode que, donat un enter **d**, l'afegisca en la base de la pila, deixant la resta d'elements amb l'ordenació relativa inicial.
- El problema es pot resoldre usant una pila auxiliar:



```
public void insertBaseIter(int d) {  
    StackIntLinked aux = new StackIntLinked();  
    while (!this.empty()) { aux.push(this.pop()); }  
    this.push(d);  
    while (!aux.empty()) { this.push(aux.pop()); }  
}
```

- Exemple:



```

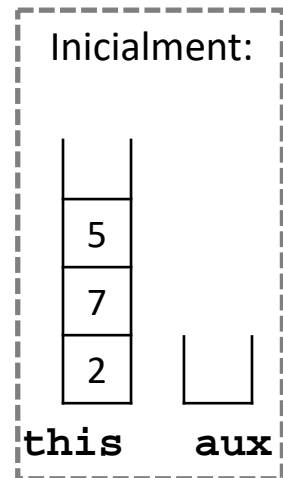
public class TestStackIntLinkedPlus {
    private TestStackIntLinkedPlus() { }

    public static void main(String[] args) {
        //1. Crea una pila plus com una llista enllaçada
        // amb els valors 2, 7 i 5
        int[] dades = {2, 7, 5};
        StackIntLinkedPlus p = creaPila(dades);

        //2. Mostra per pantalla l'estat de la pila
        // i el seu cim
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());

        //3. Empila el 9 en la base de la pila
        System.out.println("\nAfegeix el 9 en la base de la pila");
        p.insertBaseIter(9);
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());
    }
}

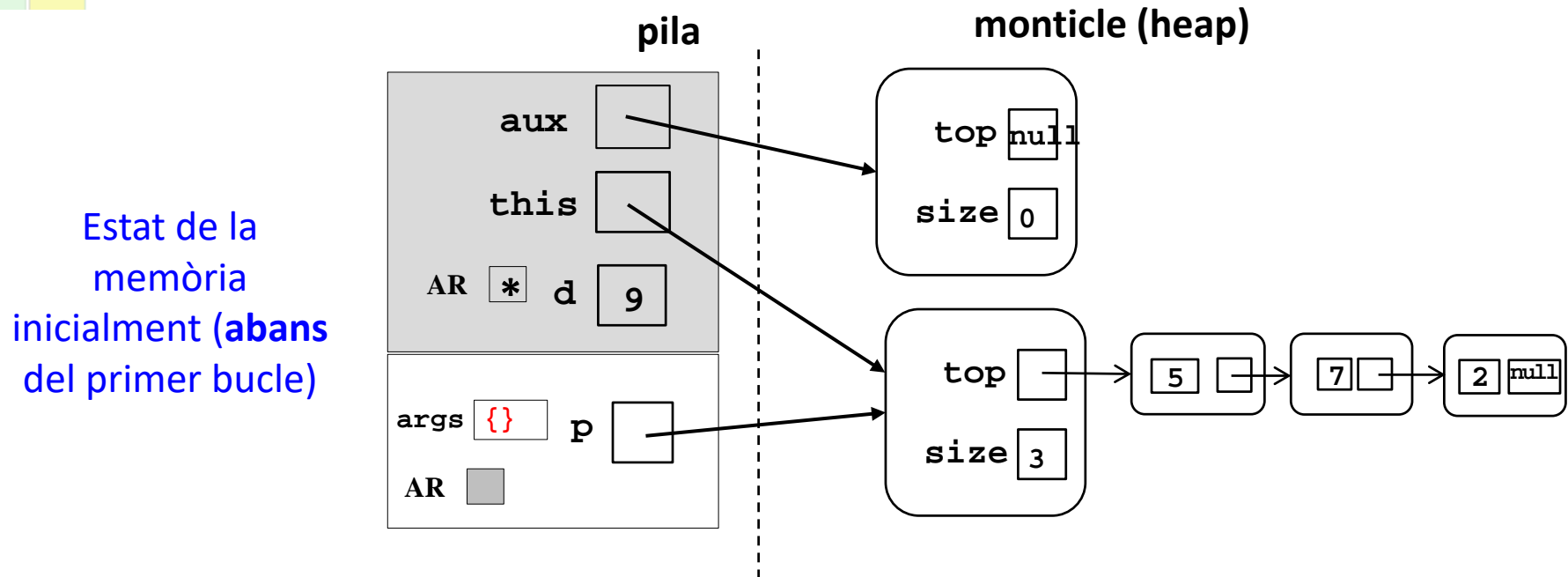
```



```

public void insertBaseIter(int d) {
    StackIntLinked aux = new StackIntLinked();
    while (!this.empty()) { aux.push(this.pop()); }
    this.push(d);
    while (!aux.empty()) { this.push(aux.pop()); }
}

```



```

public class TestStackIntLinkedPlus {
    private TestStackIntLinkedPlus() { }

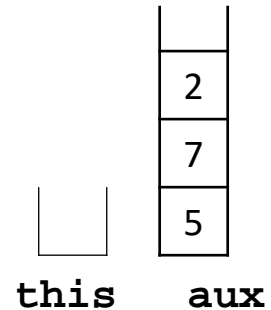
    public static void main(String[] args) {
        //1. Crea una pila plus com una llista enllaçada
        // amb els valors 2, 7 i 5
        int[] dades = {2, 7, 5};
        StackIntLinkedPlus p = creaPila(dades);

        //2. Mostra per pantalla l'estat de la pila
        // i el seu cim
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());

        //3. Empila el 9 en la base de la pila
        System.out.println("\nAfegeix el 9 en la base de la
        p.insertBaseIter(9);
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());
    }
}

```

Després del primer bucle:

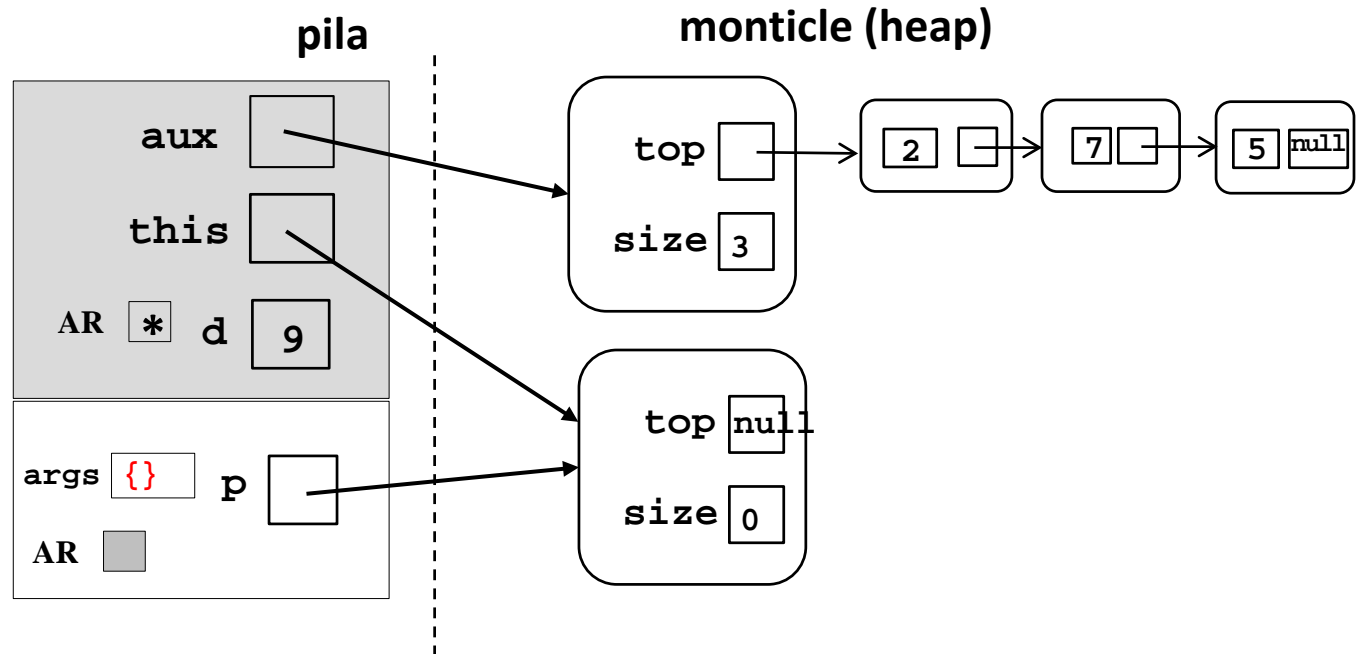


```

public void insertBaseIter(int d) {
    StackIntLinked aux = new StackIntLinked();
    while (!this.empty()) { aux.push(this.pop()); }
    this.push(d);
    while (!aux.empty()) { this.push(aux.pop()); }
}

```

Estat de la
memòria **després**
del primer bucle



```

public class TestStackIntLinkedPlus {
    private TestStackIntLinkedPlus() { }

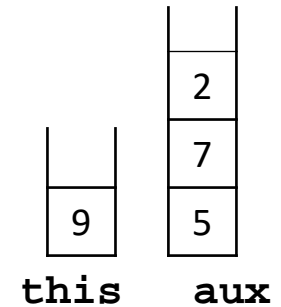
    public static void main(String[] args) {
        //1. Crea una pila plus com una llista enllaçada
        // amb els valors 2, 7 i 5
        int[] dades = {2, 7, 5};
        StackIntLinkedPlus p = creaPila(dades);

        //2. Mostra per pantalla l'estat de la pila
        // i el seu cim
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());

        //3. Empila el 9 en la base de la pila
        System.out.println("\nAfegeix el 9 en la base de la
        p.insertBaseIter(9);
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());
    }
}

```

Després d'empilar d:

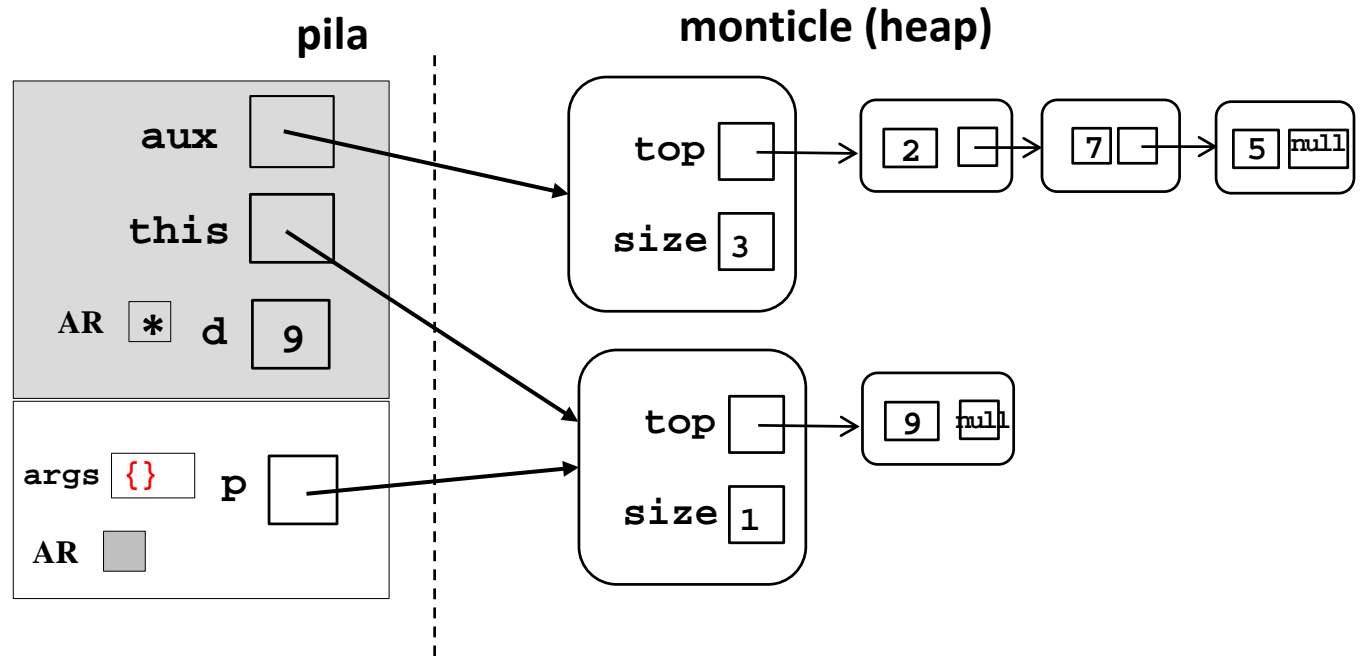


```

public void insertBaseIter(int d) {
    StackIntLinked aux = new StackIntLinked();
    while (!this.empty()) { aux.push(this.pop()); }
    this.push(d);
    while (!aux.empty()) { this.push(aux.pop()); }
}

```

Estat de la
memòria **després**
de empilar `d`



```

public class TestStackIntLinkedPlus {
    private TestStackIntLinkedPlus() { }

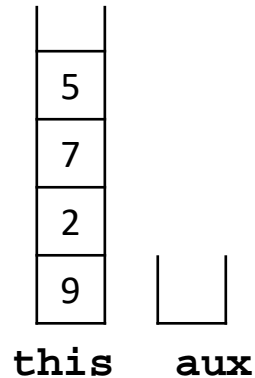
    public static void main(String[] args) {
        //1. Crea una pila plus com una llista enllaçada
        // amb els valors 2, 7 i 5
        int[] dades = {2, 7, 5};
        StackIntLinkedPlus p = creaPila(dades);

        //2. Mostra per pantalla l'estat de la pila
        // i el seu cim
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());

        //3. Empila el 9 en la base de la pila
        System.out.println("\nAfegeix el 9 en la base de la pila");
        p.insertBaseIter(9);
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());
    }
}

```

Després del segon bucle:

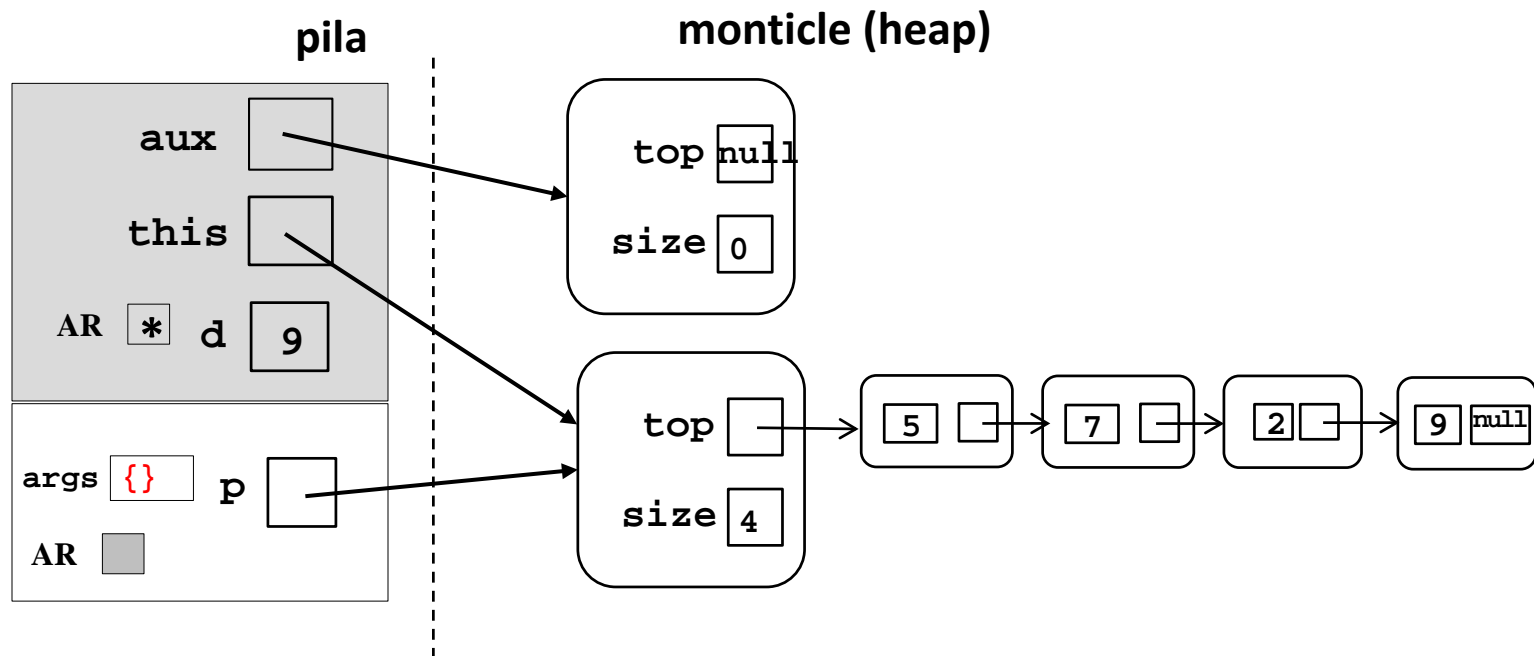


```

public void insertBaseIter(int d) {
    StackIntLinked aux = new StackIntLinked();
    while (!this.empty()) { aux.push(this.pop()); }
    this.push(d);
    while (!aux.empty()) { this.push(aux.pop()); }
}

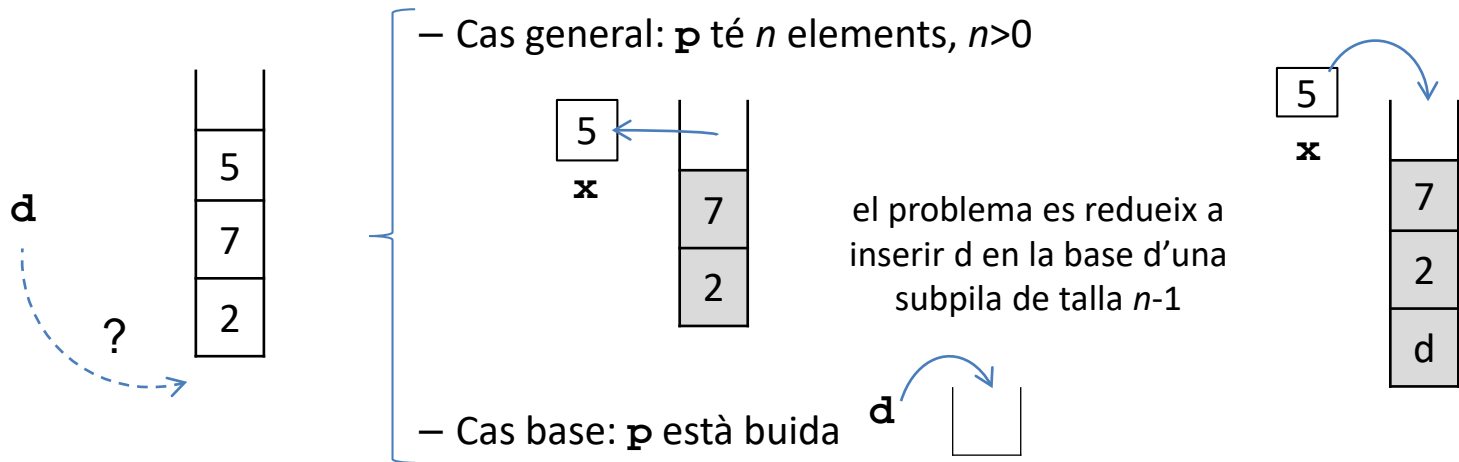
```

Estat de la memòria després del segon bucle



Piles – Example

- En el problema anterior podem notar que la resolució per a una pila p de talla n passa per:
 - desempilar el cim de p , (p es queda amb $n-1$ elements)
 - fer les operacions precises per a que d aplegue a quedar en la base de p , i els $n-1$ elements tornen a empilar-se adequadament,
 - empilar l'element que inicialment estava en el cim.
- Admiteix la següent estructuració recursiva:



```
public void insertBaseRec(int d) {  
    if (this.empty()) { this.push(d); }  
    else {  
        int x = this.pop();  
        insertBaseRec(d);  
        this.push(x);  
    }  
}
```

```

public class TestStackIntLinkedPlus {
    private TestStackIntLinkedPlus() { }

    public static void main(String[] args) {
        //1. Crea una pila plus com una llista enllaçada
        // amb els valors 2, 7 i 5
        int[] dades = {2, 7, 5};
        StackIntLinkedPlus p = creaPila(dades);
        ...

        //3. Empila el 9 en la base de la pila
        System.out.println("\nAfegeix el 9 en la base de la pila. ");
        p.insertBaseRec(9);
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());
    }
}

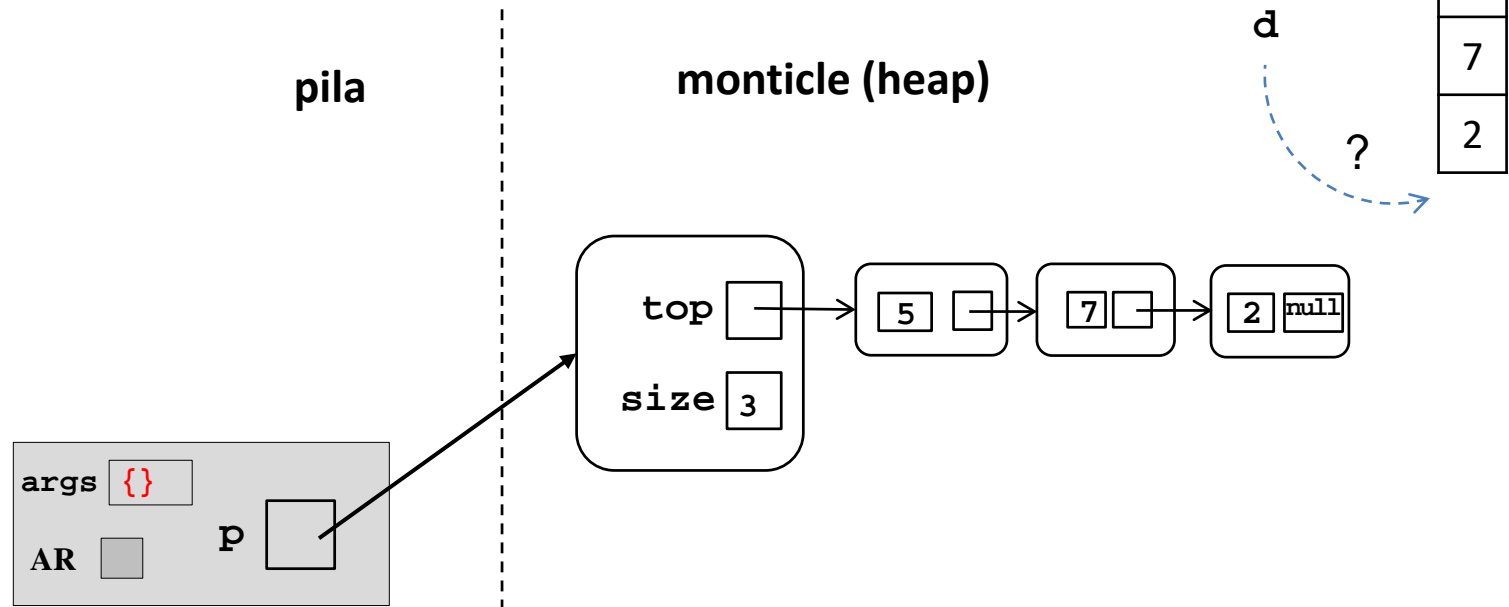
```

```

public void insertBaseRec(int d) {
    if (this.empty()) { this.push(d); }
    else {
        int x = this.pop();
        this.insertBaseRec(d);
        this.push(x);
    }
}

```

Estat de la memòria en el **main**, abans de la crida **p.insertBaseRec(9)**




```

public class TestStackIntLinkedPlus {
    private TestStackIntLinkedPlus() { }

    public static void main(String[] args) {
        //1. Crea una pila plus com una llista enllaçada
        // amb els valors 2, 7 i 5
        int[] dades = {2, 7, 5};
        StackIntLinkedPlus p = creaPila(dades);
        ...

        //3. Empila el 9 en la base de la pila
        System.out.println("\nAfegeix el 9 en la base de la pila. ");
        p.insertBaseRec(9);
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());
    }
}

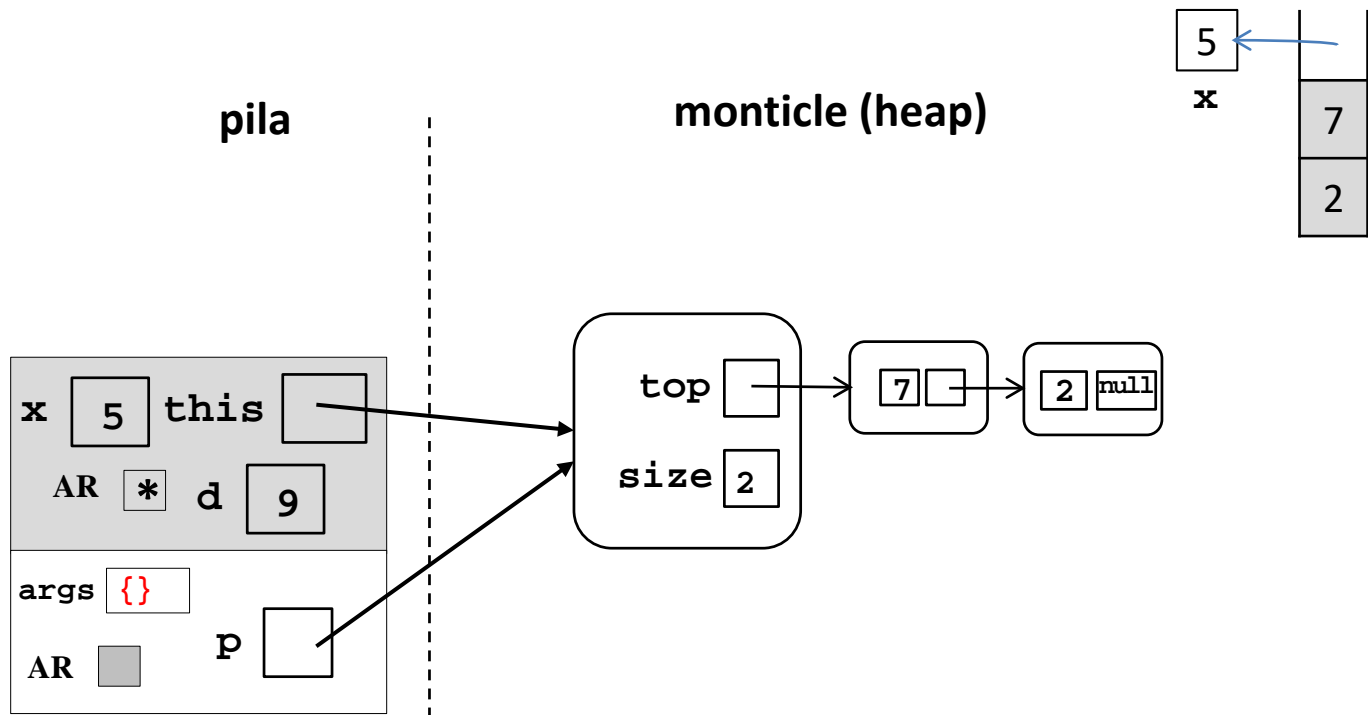
```

```

public void insertBaseRec(int d) {
    if (this.empty()) { this.push(d); }
    else {
        int x = this.pop();
        this.insertBaseRec(d);
        this.push(x);
    }
}

```

Estat de la
memòria
en el punt
de ruptura



```

public class TestStackIntLinkedPlus {
    private TestStackIntLinkedPlus() { }

    public static void main(String[] args) {
        //1. Crea una pila plus com una llista enllaçada
        // amb els valors 2, 7 i 5
        int[] dades = {2, 7, 5};
        StackIntLinkedPlus p = creaPila(dades);
        ...

        //3. Empila el 9 en la base de la pila
        System.out.println("\nAfegeix el 9 en la base de la pila. ");
        p.insertBaseRec(9);
        System.out.println(p.toString());
        System.out.println("Cim = " + p.peek());
    }
}

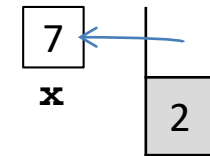
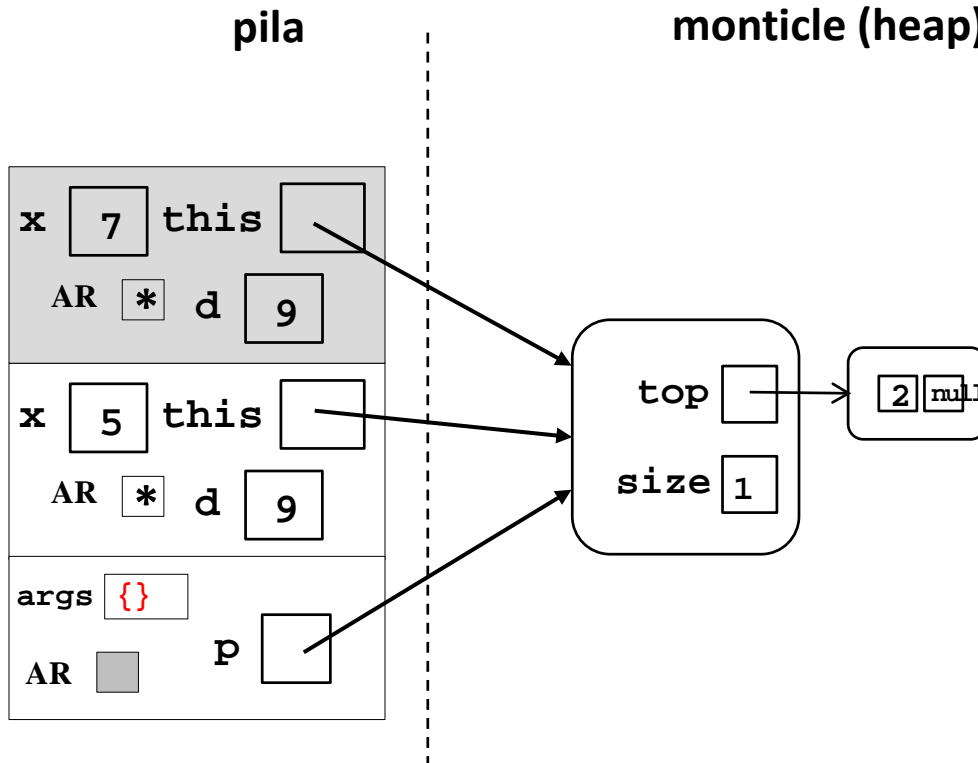
```

```

public void insertBaseRec(int d) {
    if (this.empty()) { this.push(d); }
    else {
        int x = this.pop();
        this.insertBaseRec(d);
        this.push(x);
    }
}

```

Estat de la
memòria
en el punt
de ruptura



```
public class TestStackIntLinkedPlus {
```

```
...
```

```
//3. Empila el 9 en la base de la pila
```

```
System.out.println("\nAfegeix el 9 en la base de la pila. ");
```

```
p.insertBaseRec(9);
```

```
System.out.println(p.toString());
```

```
System.out.println("Cim = " + p.peek());
```

```
public void insertBaseRec(int d) {
```

```
if (this.empty()) { this.push(d); }
```

```
else {
```

```
int x = this.pop();
```

```
→ this.insertBaseRec(d);
```

```
this.push(x);
```

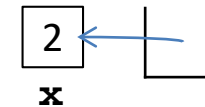
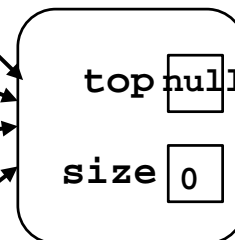
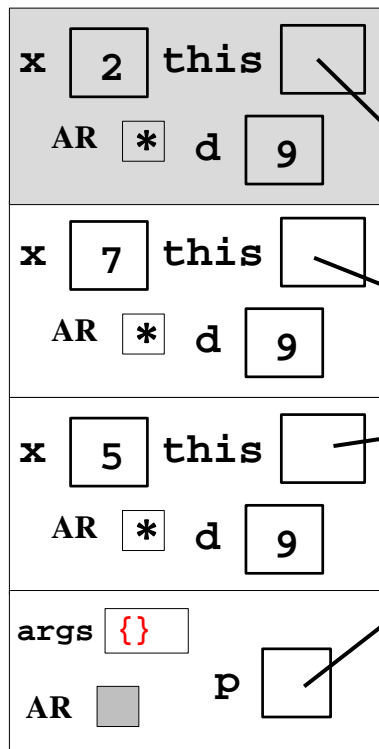
```
}
```

```
}
```

Estat de la
memòria
en el punt
de ruptura

pila

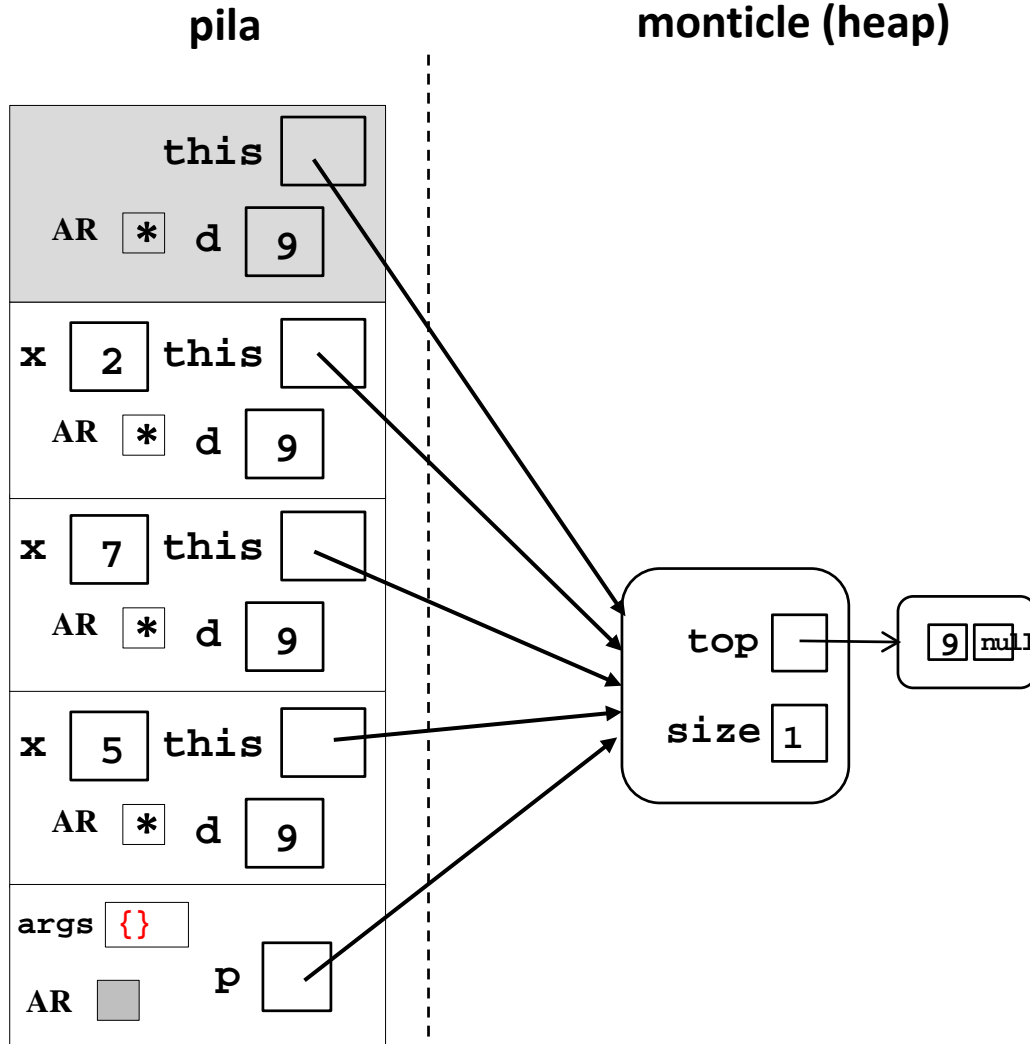
monticle (heap)



```
public class TestStackIntLinkedListPlus {
    ...
    //3. Empila el 9 en la base de la pila
    System.out.println("\nAfegeix el 9 en la base de la pila. ");
    p.insertBaseRec(9);
    System.out.println(p.toString());
    System.out.println("Cim = " + p.peek());
}
```

```
public void insertBaseRec(int d) {
    if (this.empty()) { this.push(d); }
    else {
        int x = this.pop();
        this.insertBaseRec(d);
        this.push(x);
    }
}
```

Estat de la memòria just en acabar el cas base

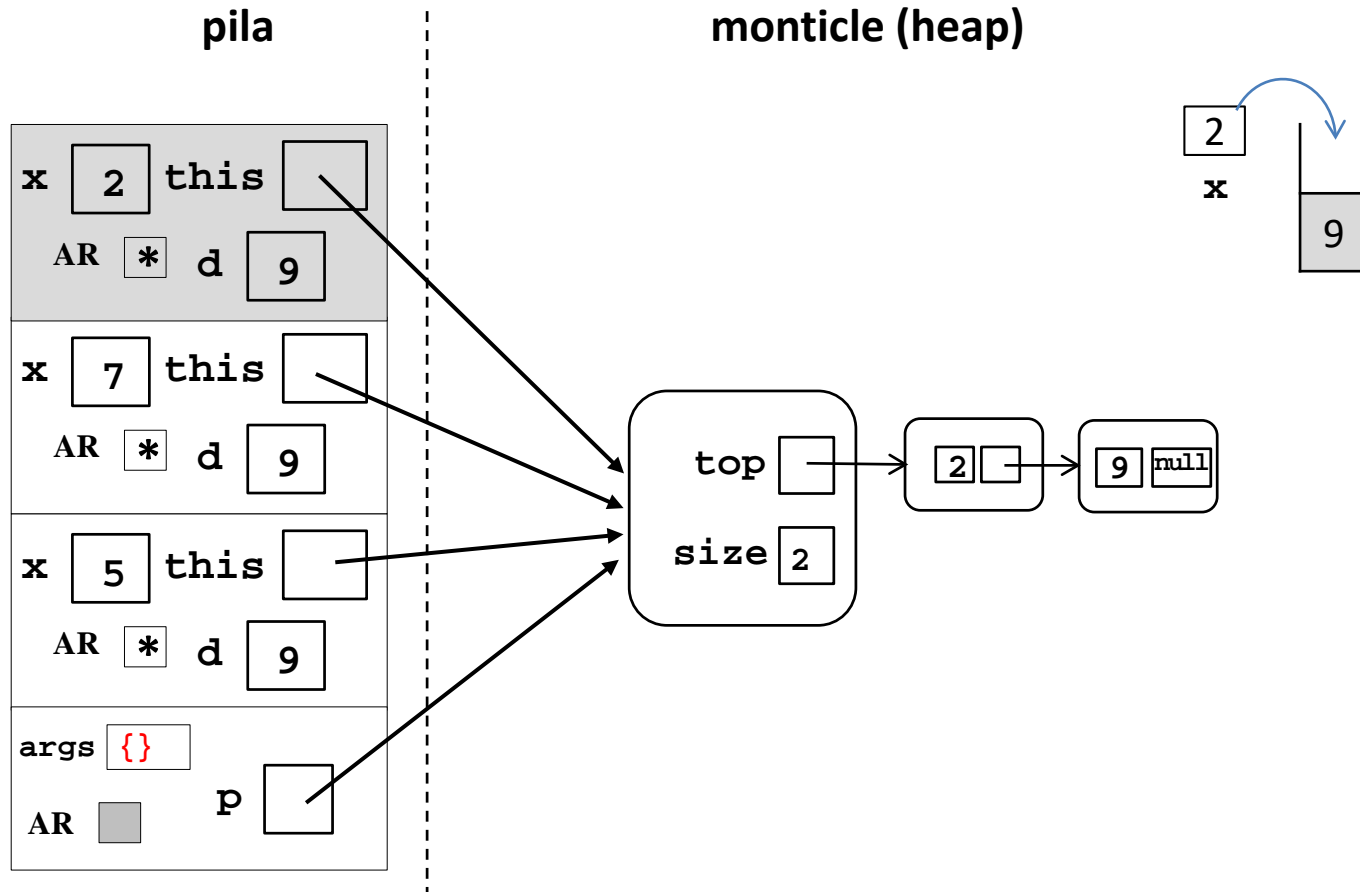


```
public class TestStackIntLinkedPlus {
    ...

    //3. Empila el 9 en la base de la pila
    System.out.println("\nAfegeix el 9 en la base de la pila. ");
    p.insertBaseRec(9);
    System.out.println(p.toString());
    System.out.println("Cim = " + p.peek());
}
```

```
public void insertBaseRec(int d) {
    if (this.empty()) { this.push(d); }
    else {
        int x = this.pop();
        this.insertBaseRec(d);
        this.push(x);
    }
}
```

Estat de la
memòria
just abans
d'acabar



```

public class TestStackIntLinkedPlus {
    ...

    //3. Empila el 9 en la base de la pila
    System.out.println("\nAfegeix el 9 en la base de la pila. ");
    p.insertBaseRec(9);
    System.out.println(p.toString());
    System.out.println("Cim = " + p.peek());
}

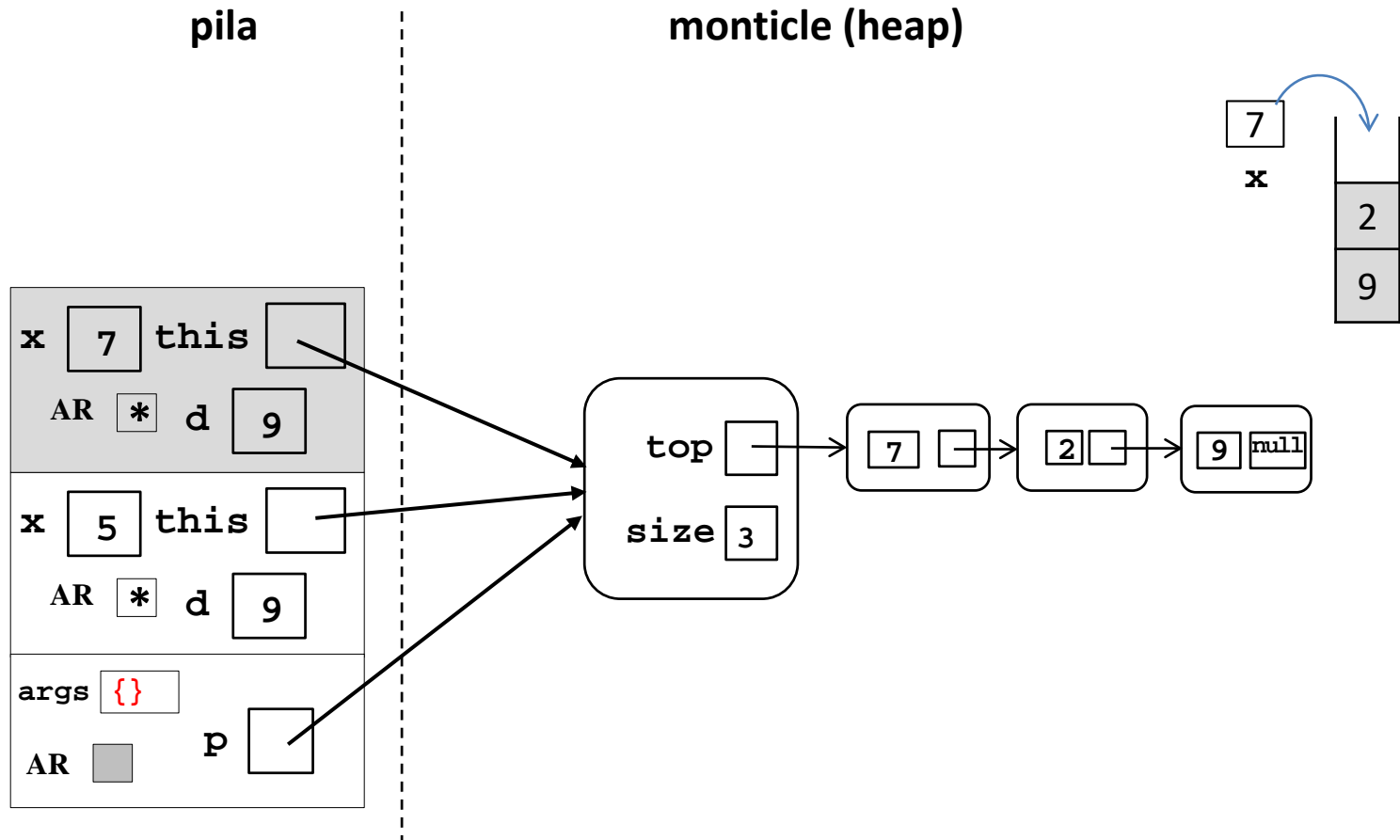
```

```

public void insertBaseRec(int d) {
    if (this.empty()) { this.push(d); }
    else {
        int x = this.pop();
        this.insertBaseRec(d);
        this.push(x);
    }
}

```

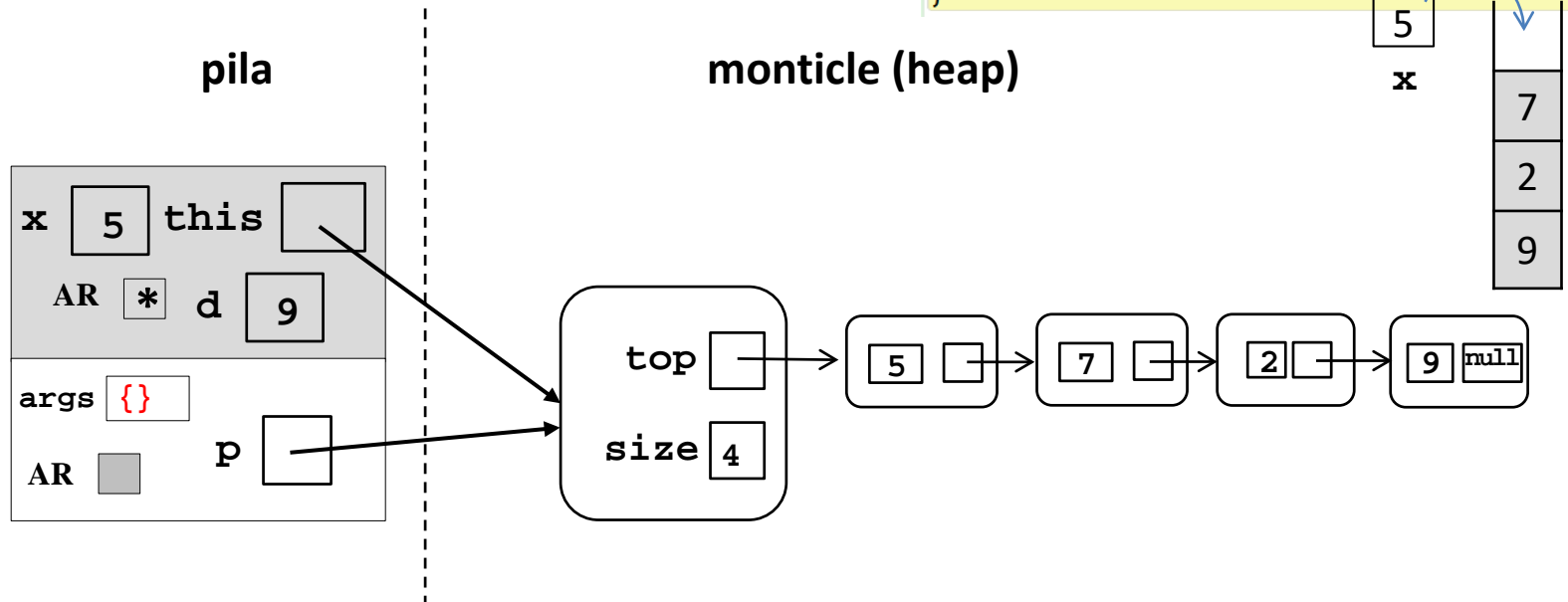
Estat de la memòria just abans d'acabar



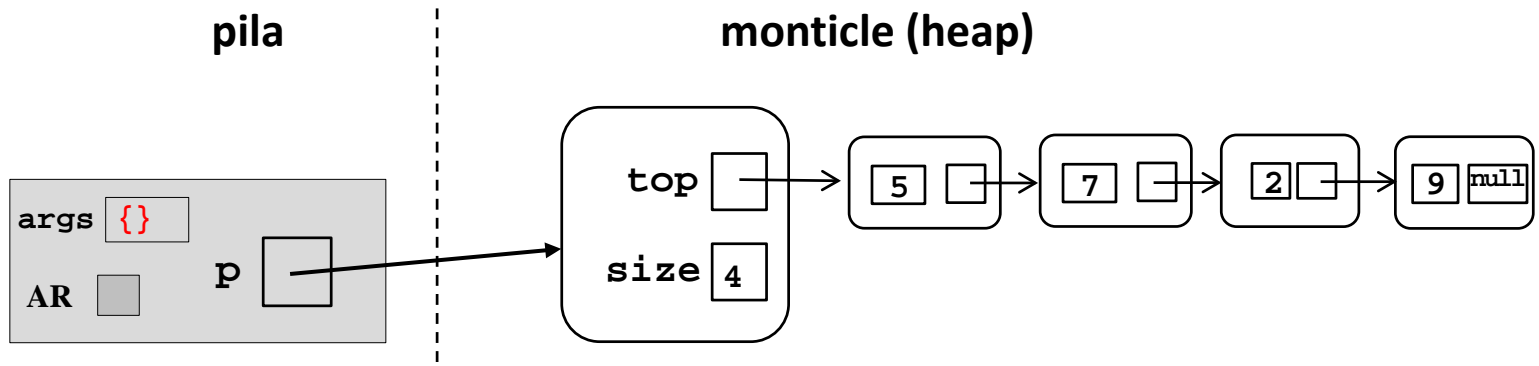
```
public class TestStackIntLinkedPlus {
    ...
    //3. Empila el 9 en la base de la pila
    System.out.println("\nAfegeix el 9 en la base de la pila. ");
    p.insertBaseRec(9);
    System.out.println(p.toString());
    System.out.println("Cim = " + p.peek());
}
```

```
public void insertBaseRec(int d) {
    if (this.empty()) { this.push(d); }
    else {
        int x = this.pop();
        this.insertBaseRec(d);
        this.push(x);
    }
}
```

Estat de la memòria just abans d'acabar

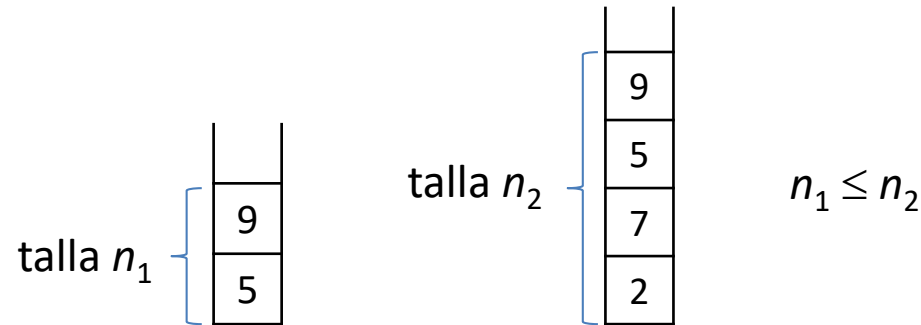


Estat de la memòria en tornar al main

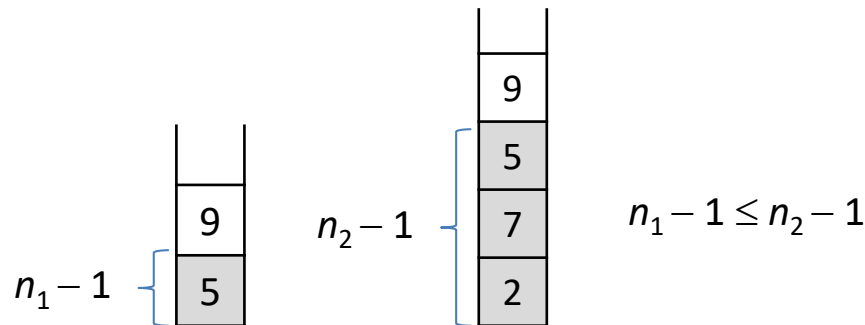


Piles – Exercici

- Donades dues piles **p1** i **p2**, **p1** de menor o igual talla que **p2**, es desitja saber si **p1** és “sombrero” de **p2**, és a dir, els elements de **p1** apareixen en la part de dalt de **p2** i en el mateix ordre. Trivialment, la pila buida és sombrero de qualsevol altra. Exemple:

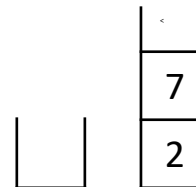


- Completa, en la classe **StackIntLinkedPlus**, un mètode **recursiu esSombrero** que comprova si una pila **p** és sombrero de l'actual (**this**).
 - Cas general. **p** no és buida, (**this** tampoc ho és per ser de talla major o igual).



p és sombrero de **this** sii coincideixen els seus cims, i **p.pop()** és sombrero de **this.pop()**.

- Cas base. **p** és buida, és sombrero de **this**.



Piles – Exercicis

- **Completa** en la classe **StackIntLinkedPlus** (derivada de **StackIntLinked**) els següents mètodes recursius:
 - `public void esborrarBase()` que esborra la dada en la base la pila actual, deixant la resta d'elements amb l'ordenació relativa inicial. Si la pila està buida, no fa res.
 - `public int sumarPila()` que suma les dades de la pila actual.
 - `public void canviarSigne()` que canvia el signe de les dades de la pila.
- **Completa** la classe **TestStackIntLinkedPlus** del paquet **linear** seguint els comentaris. Et servirà per comprovar la correcció dels mètodes de la classe **StackIntLinkedPlus**.

Piles – Exercici

- Donada una String **s** que conté únicament els caràcters '[', ']', ')', '(', es vol implementar un mètode que comprove si **s** està *ben parentitzada* (amb les regles habituals de parentització). Per exemple, per a "[()] []" el mètode ha de tornar **true**, per a "[() []] []" o "[() ()] []" ha de tornar **false**.
- El problema es pot resoldre examinant un per un els successius caràcters de **s** per a veure si tot parèntesi tancat que aparega *cancel·la* el corresponent parèntesi obert d'entre els precedents. Mentre queden caràcters per examinar:

[[~~()~~]

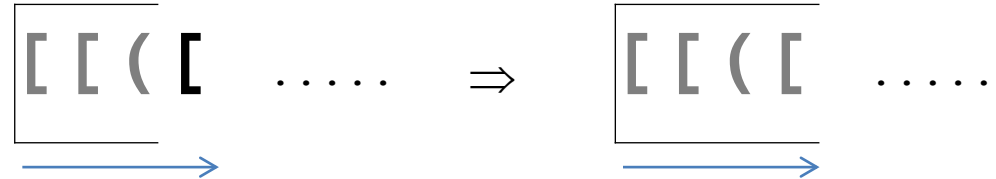


- Si el caràcter examinat és un parèntesi tancat, hi ha que veure si forma parella amb el parèntesi obert pendent de tancar **més recent** (cas de quedar-ne algun), per a cancel·lar-lo.
⇒ Cal recordar la seqüència de parèntesi oberts que van aparegut i quedant pendents de cancel·lar.
 - Si el caràcter examinat és un parèntesi obert, s'ha d'afegir a la seqüència de parèntesi pendents de tancar (aquesta seqüència pot gestionar-se com una **pila**).
- L'expressió estarà mal parentitzada si es troba un parèntesi tancat que no cancel·la a l'obert més recent o, si processats tots els caràcters, queden parèntesi oberts pendents de tancar.

Piles – Exercici

- Iterativament, es recorren un a un seqüencialment els caràcters de l'expressió.
- En cada iteració, per al caràcter c examinat es pot fer el següent anàlisi de casos:

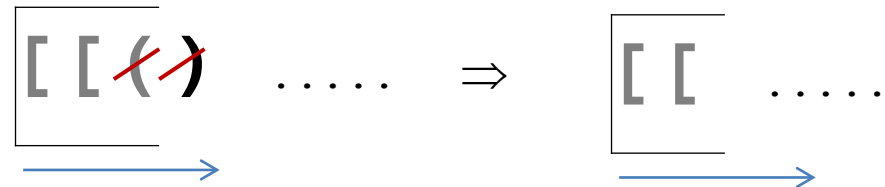
- c és un parèntesi obert:



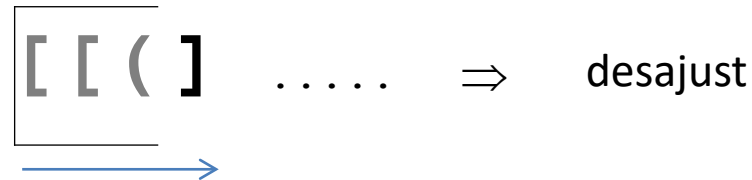
- c és un parèntesi tancat:

- té un caràcter immediatament anterior pendent

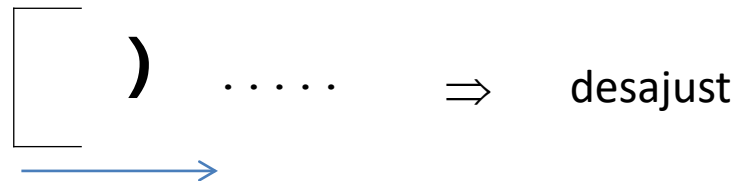
al que cancel·la



al que no cancel·la



- no té caràcter anterior pendent de cancel·lar



- En acabar el bucle, l'expressió ha resultat estar ben parentitzada si no s'ha trobat cap desajust i no queden parèntesi oberts per cancel·lar.

Piles – Exercici

- En la classe BenParentitzada del paquet *usaLinear*, completa el mètode *expressioBenParent* i comprova que és correcte executant el *main*.

```
/** Comprova si els caràcters de s formen una expressió
 * ben parentitzada de '[', ']', '(', ')'.
 */
public static boolean expressioBenParent(String s) {
    // Crear una pila de parèntesi oberts pendents de tancar,
    // inicialment buida

    boolean desajust = false;
    int i = 0;
    while (i < s.length() && !desajust) {
        char c = s.charAt(i);
        if (c == '(' || c == '[') { /* empilar c en la pila */ }
        else { // c és un parèntesi tancat
            if (/* queden parèntesi per tancar */) {
                if (/* té un caràcter immediatament anterior pendent al que cancel·la */) {
                    /* desempilar */
                }
                else { desajust = true; }
            }
            else { desajust = true; }
        }
        i++;
    }
    // torna true sii s'han tancat correctament tots els parèntesi
    return /* completar */;
}

public static void main(String[] args) {
    String[] s = {"[()()]", "[()[]]", "[()()]" };
    for (int i = 0; i < s.length; i++) {
        boolean b = expressioBenParent(s[i]);
        System.out.printf("La cadena %s ", s[i]);
        if (!b) { System.out.printf("NO "); }
        else { System.out.printf("SÍ "); }
        System.out.printf("està ben parentitzada.\n");
    }
}
```

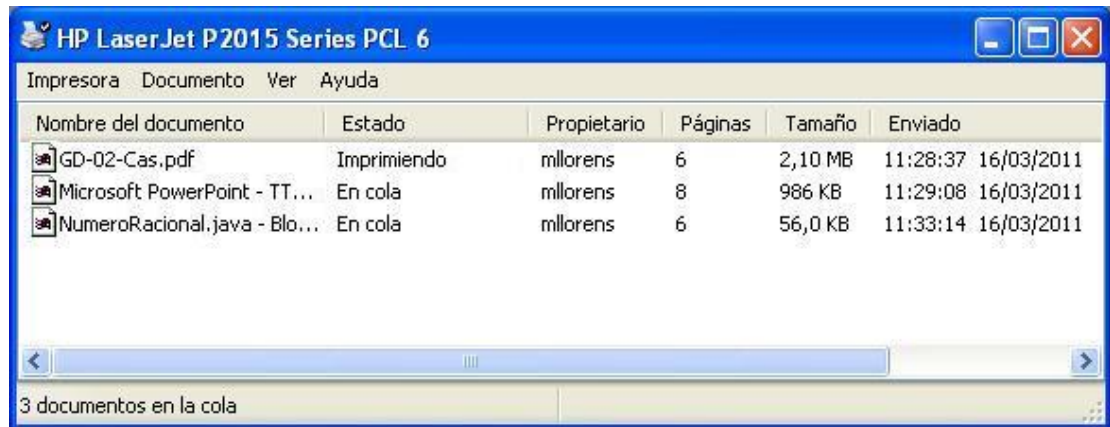
BlueJ: Terminal Window - llibresPRG-Sol

Options

```
La cadena [()()] SÍ està ben parentitzada.
La cadena [()[]] NO està ben parentitzada.
La cadena [()()] NO està ben parentitzada.
```

Cues

- Una *cua* (*queue*) és una col·lecció de dades del mateix tipus en la que l'accés es realitza seguint un criteri *FIFO* (*First In First Out*).
 - El primer element que arriba (que entra en la cua) és el primer en ser atès (en ser eliminat de la cua).
- Exemples de cues:



- Cua d'impressió dels treballs d'una impressora
- Execució d'aplicacions en un sistema amb 1 únic processador
- Paquets d'informació en xarxes computacionals

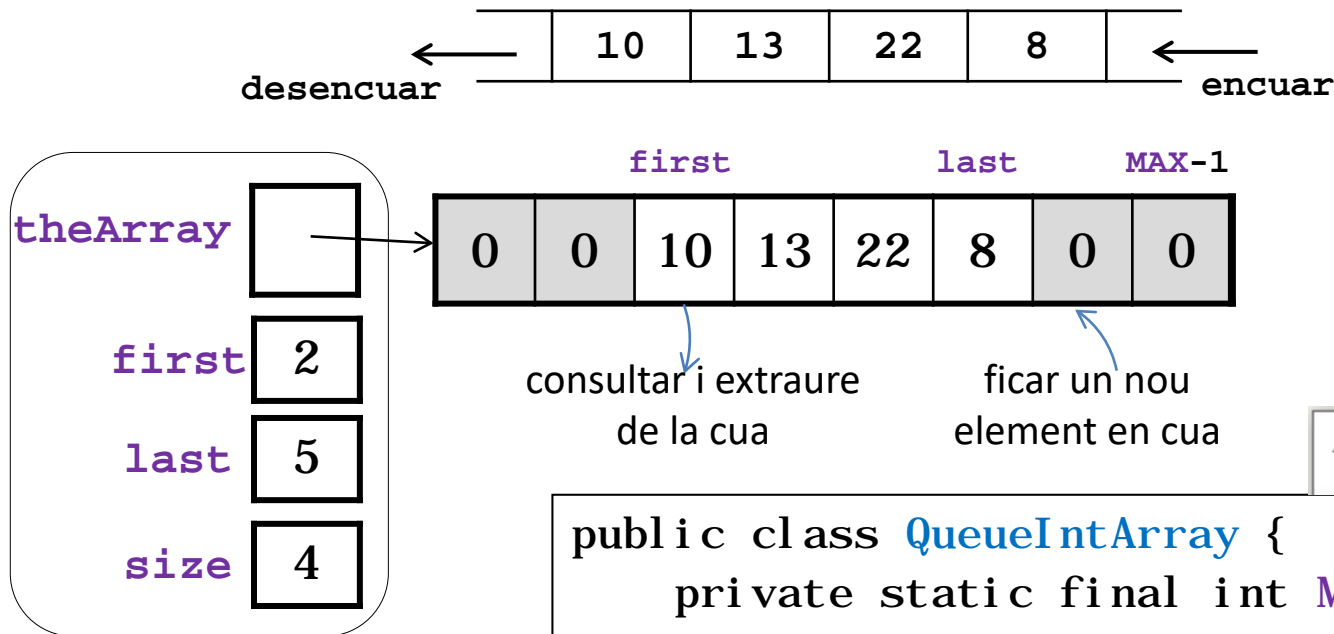
Cues - Operacions


Operació	Descripció
<code>public Queue()</code>	Crea una nova Queue buida.
<code>public void add(Tipus element)</code>	Insereix (<i>encua</i>) l'element al final de la Queue.
<code>public Tipus remove()</code>	Extrau (<i>desencua</i>) l'element en el cap de la Queue i el torna. Llança <code>NoSuchElementException</code> si la Queue està buida.
<code>public Tipus element()</code>	Torna l'element en el cap de la Queue (sense desencuar-lo). Llança <code>NoSuchElementException</code> si la Queue està buida.
<code>public boolean empty()</code>	Torna <code>true</code> si la Queue està buida i <code>false</code> en cas contrari.
<code>public int size()</code>	Torna el nombre d'elements de la Queue.

Es pot aconseguir implementar aquestes operacions amb cost constant.

Cues – Implementació amb arrays

- Una classe **Queue** pot estructurar-se amb els atributs:
 - **theArray**: array de capacitat máxima **MAX**, que emmagatzema les dades en posicions consecutives, no necessàriament començant en l'índex 0.
 - **first** i **last**: índexs que marquen els dos punts d'accés a la cua.
 - **size**: nombre d'elements de la cua.

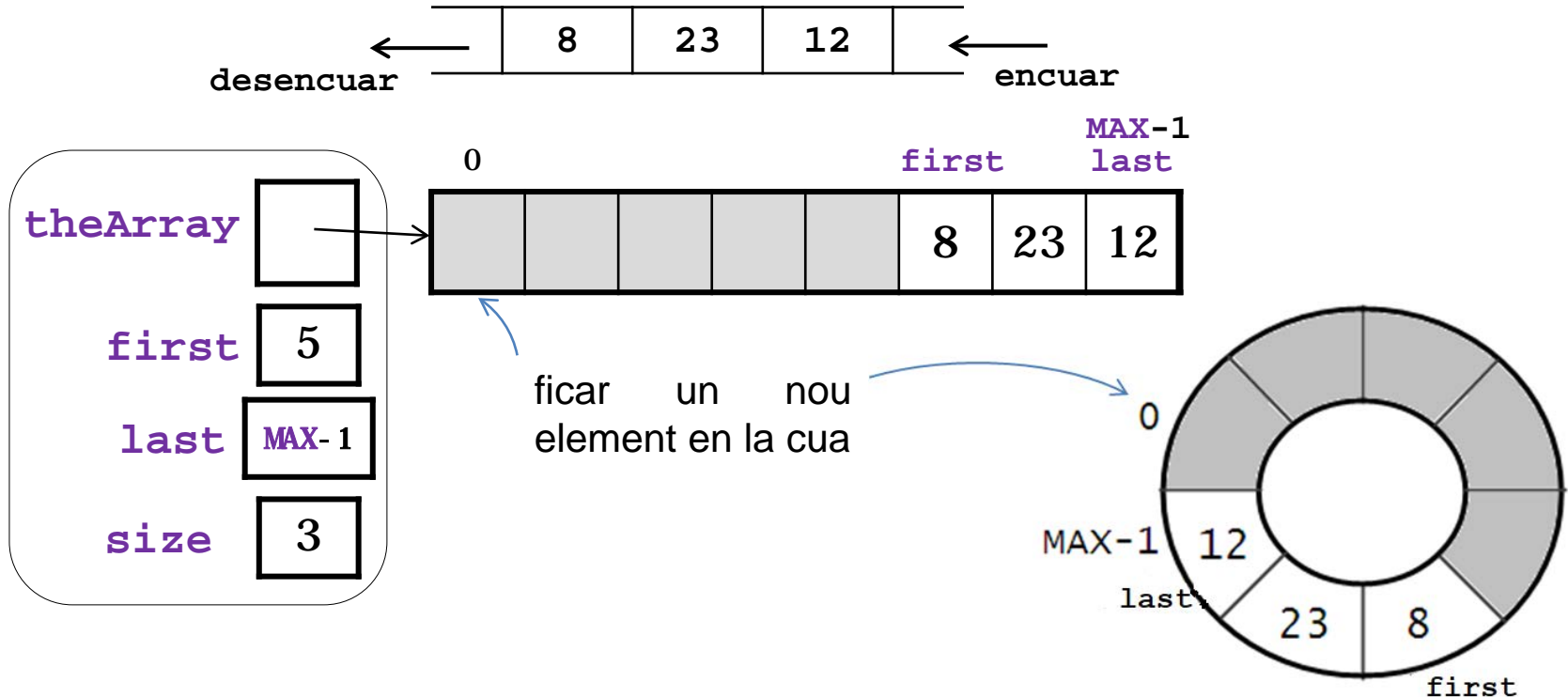


 QueueIntArray - llibreriesPRG

```
public class QueueIntArray {  
    private static final int MAX = ...;  
    private int[] theArray;  
    private int first, last, size;  
  
    // Implementació de les operacions:  
    ...  
}
```

Cues – Implementació amb arrays

- Per a poder reutilitzar totes les posicions de l'array sense haver de desplaçar elements, es considera l'array com una *estructura circular* continua, sense principi ni fi, on la posició següent de $MAX - 1$ és 0.

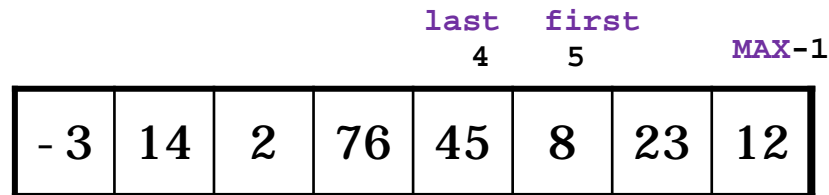
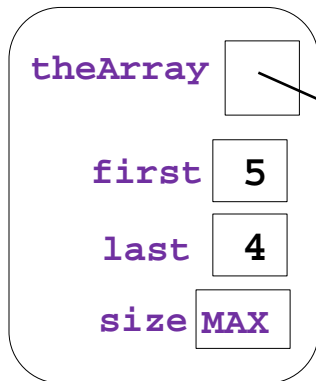
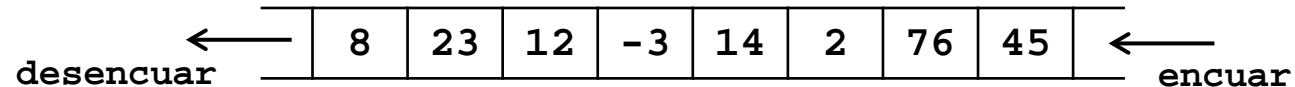


Mètode auxiliar de càlcul de l'índex següent a `i` en aquest array *circular*

```
private static int increment(int i, int length) {  
    i++;  
    if (i == length) { i = 0; }  
    return i;  
}
```

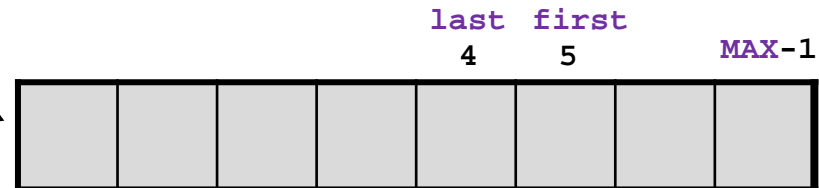
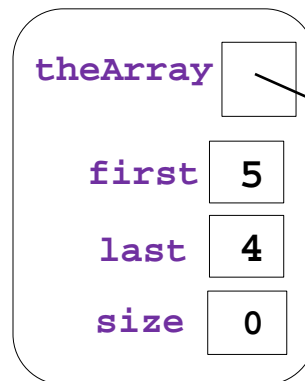
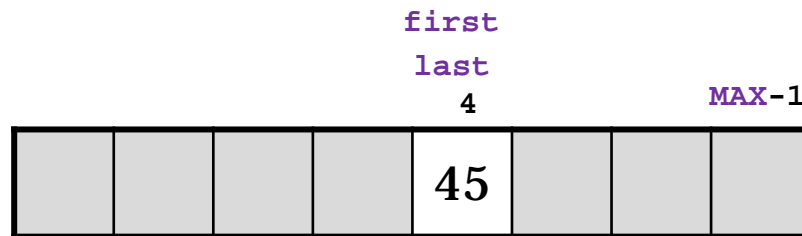
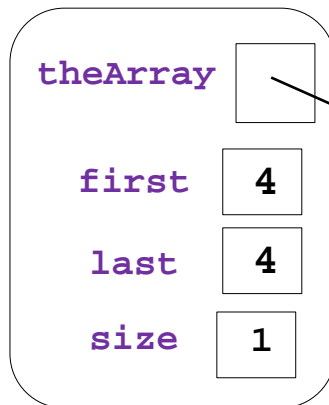

Cues – Implementació amb arrays

- L'atribut **size** permet distingir entre una cua plena i una buida quan **first** és l'índex següent de **last** en l'array circular.
- Exemple:



En successives operacions de desencuar,
first aplegaria a **MAX-1**, 0, 1, 2, ...

...



Cues – Implementació amb arrays

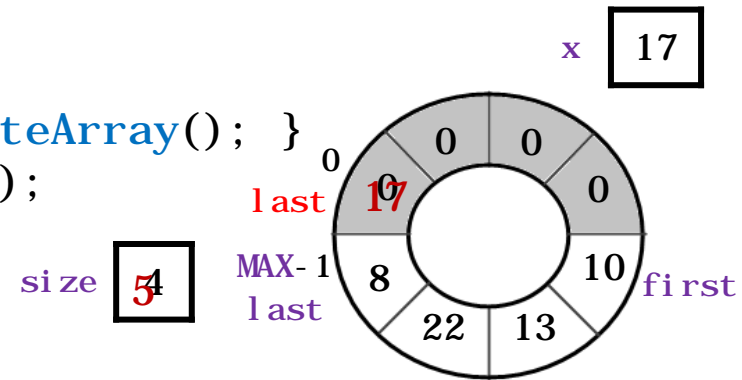
- Operació constructora `QueueIntArray`: Crea l'array e inicialitza els punts d'accés, indicant que la cua està buida.

```
public QueueIntArray() {  
    theArray = new int[MAX];  
    size = 0; first = 0; last = -1;  
}
```

- Operació `add` (encuar):

```
public void add(int x) {  
    if (size == theArray.length) { duplicateArray(); }  
    last = increment(last, theArray.length);  
    theArray[last] = x;  
    size++;  
}
```

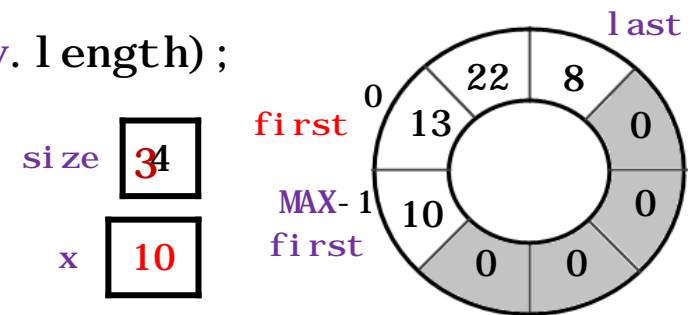
```
private void duplicateArray() {  
    int[] aux = new int[2 * theArray.length];  
    for (int i = 0, pos = first; i < theArray.length; i++) {  
        aux[i] = theArray[pos];  
        pos = increment(pos, theArray.length);  
    }  
    first = 0; last = theArray.length - 1; theArray = aux;  
}
```



Cues – Implementació amb arrays

- Operacions **remove** (desencuar) i **element** (consultar quin és el primer): llancen l'excepció *unchecked NoSuchElementException* si la cua està buida.

```
public int remove() {  
    if (size == 0) {  
        throw new NoSuchElementException("Empty queue");  
    }  
    int x = theArray[first];  
    first = increment(first, theArray.length);  
    size--;  
    return x;  
}  
  
public int element() {  
    if (size == 0) {  
        throw new NoSuchElementException("Empty queue");  
    }  
    return theArray[first];  
}
```



- Operació consultora **empty**:

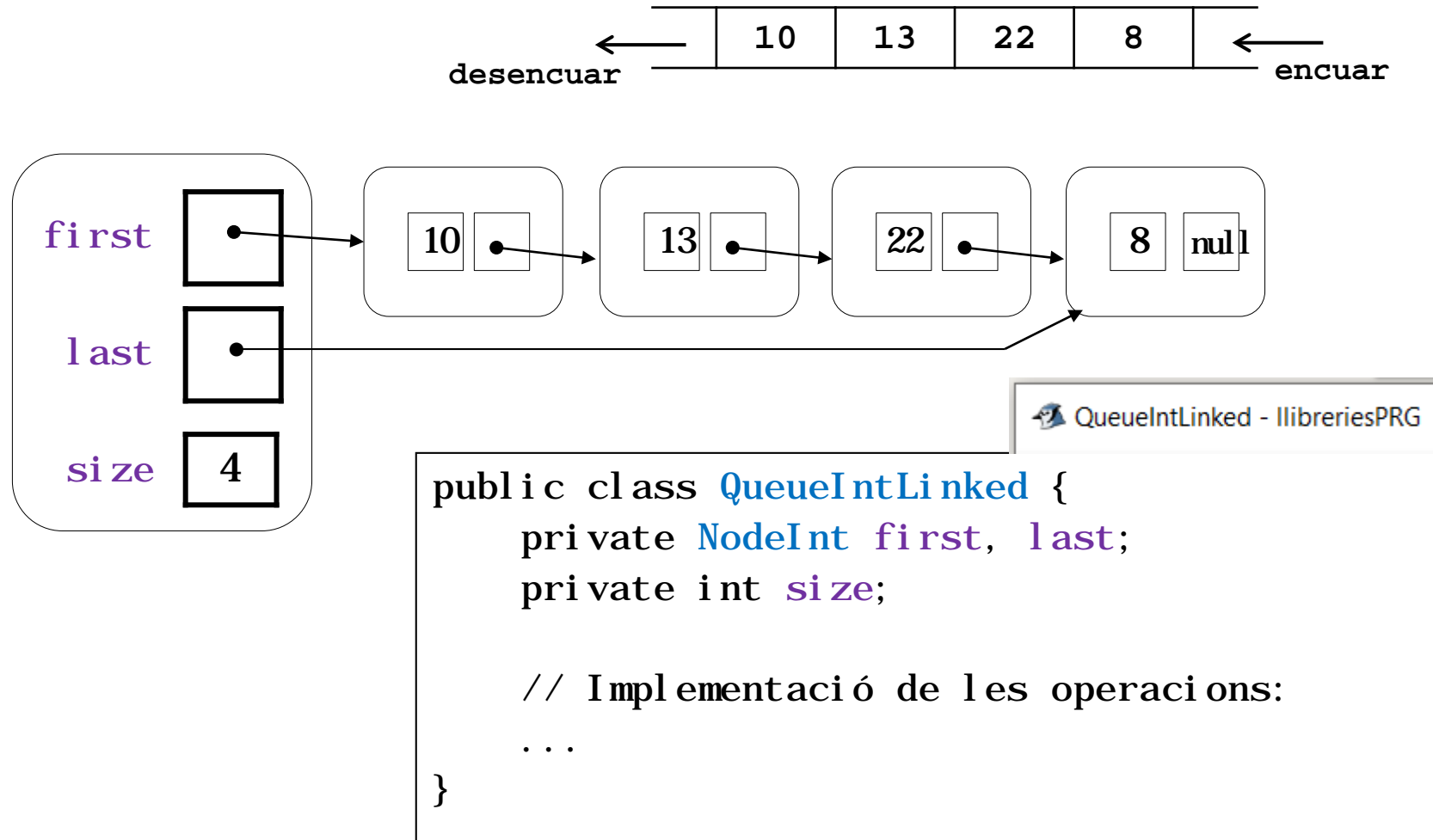
```
public boolean empty() { return size == 0; }
```

- Operació consultora **size**:

```
public int size() { return size; }
```

Cues – Implementació enllaçada

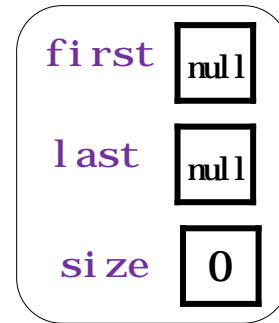
- En una **implementació enllaçada**, la cua tindrà dos atributs per a mantindre les referències al principi i al final, i un atribut que indique el número d'elements de la cua.



Cues – Implementació enllaçada

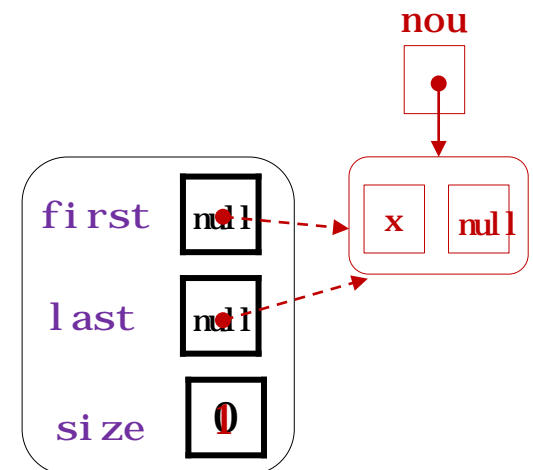
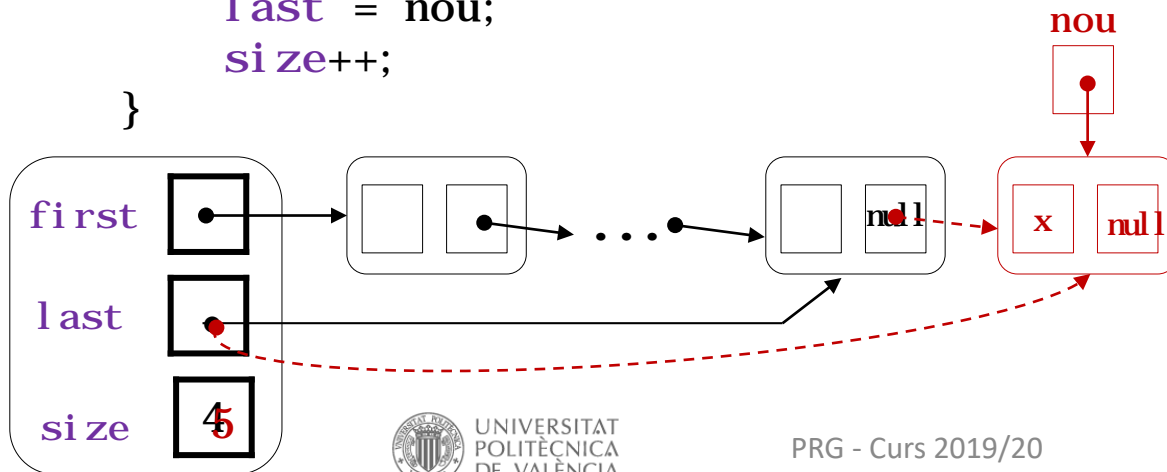
- Operació constructora **QueueIntLinked**: Assigna **null** a les referències **first** i **last** i inicialitza **size** a 0.

```
public QueueIntLinked() {  
    first = null;  
    last = null;  
    size = 0;  
}
```



- Operació **add** (encuar):

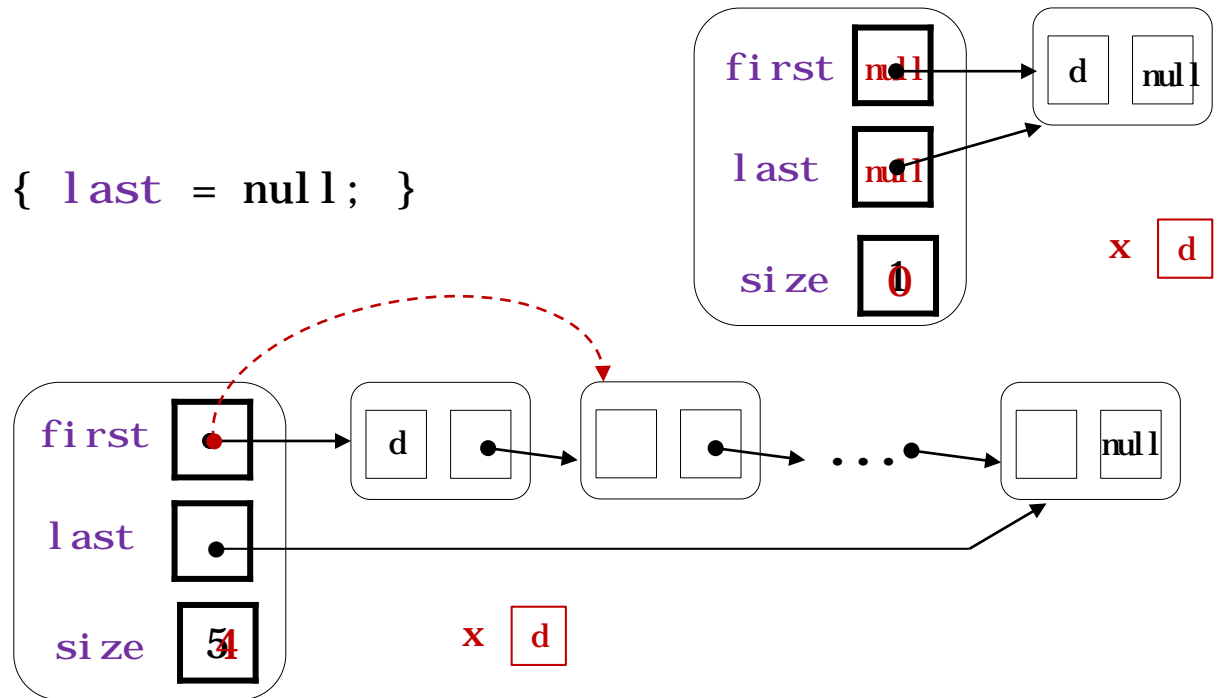
```
public void add(int x) {  
    NodeInt nou = new NodeInt(x);  
    if (last != null) { last.next = nou; }  
    else { first = nou; }  
    last = nou;  
    size++;  
}
```



Cues – Implementació enllaçada

- Operacions **remove** (desencuar) i **element** (consultar quin és el primer): llancen l'excepció *unchecked NoSuchElementException* si la cua està buida.

```
public int remove() {  
    if (size == 0) {  
        throw new NoSuchElementException("Empty queue");  
    }  
    int x = first.data;  
    first = first.next;  
    if (first == null) { last = null; }  
    size--;  
    return x;  
}
```



```
public int element() {  
    if (size == 0) {  
        throw new NoSuchElementException("Empty queue");  
    }  
    return first.data;  
}
```


Cues – Implementació enllaçada


- Operació consultora **empty**:

```
public boolean empty() { return first == null; }
```

- Operació consultora **size**:

```
public int size() { return size; }
```

 QueueIntArray - llibreriesPRG

 QueueIntLinked - llibreriesPRG

- Revisa** els mètodes **toString()** i **equals(Object)** de les classes **QueueIntLinked** i **QueueIntArray**
- Revisa** la classe **TestQueueInt** i executa-la. Et servirà per provar els mètodes de les classes **QueueIntLinked** i **QueueIntArray**

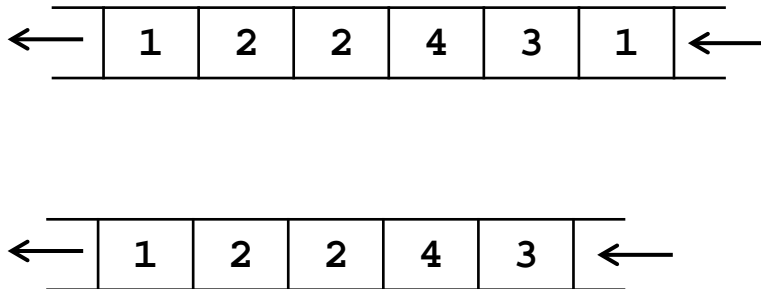
Cues – Comparació d'implementacions

- La **complexitat temporal** de totes les operacions en les dues representacions estudiades és independent de la talla del problema: $\Theta(1)$.
- Pel que fa a la **complexitat espacial**, la implementació amb arrays presenta l'inconvenient de l'estimació del tamany màxim de l'array i la reserva d'espai que en molts casos no s'utilitzarà, igual que succeïa amb les piles.
- També en aquest cas, la representació enllaçada requereix un espai de memòria addicional per als enllaços.
- Si el tipus de les components de la cua és relativament menut, com és el cas d'una cua d'int, o de cues d'objectes (les components són llavors referències), aquest consum addicional d'espai no tindrà massa importància.

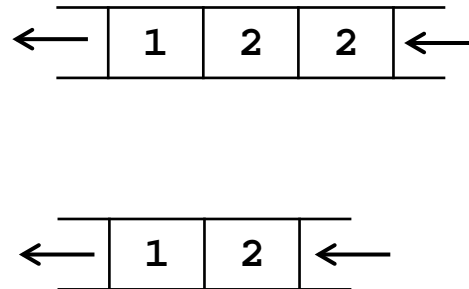
Cues – Exercici

- Afegir a la classe **QueueIntLinked** un mètode d'instància que divideixca la cua en dues meitats amb perfil `public QueueIntLinked divideQueue()`. Tot i tenint en compte que:
 - Precondició: la cua inicial (**this**) té al menys dos elements.
 - La divisió es realitza de forma que la cua inicial es queda amb la primera meitat dels elements, i es retorna una cua amb la resta d'elements.
 - Les cues resultants mantenen l'ordre dels elements en la cua inicial.
 - Si la cua inicial té una quantitat senar d'elements, serà la cua retornada la que tindrà una longitud superior en una unitat.
- Exemples:

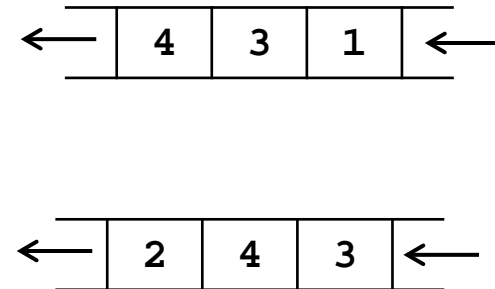
Cua inicial



Cua inicial modificada

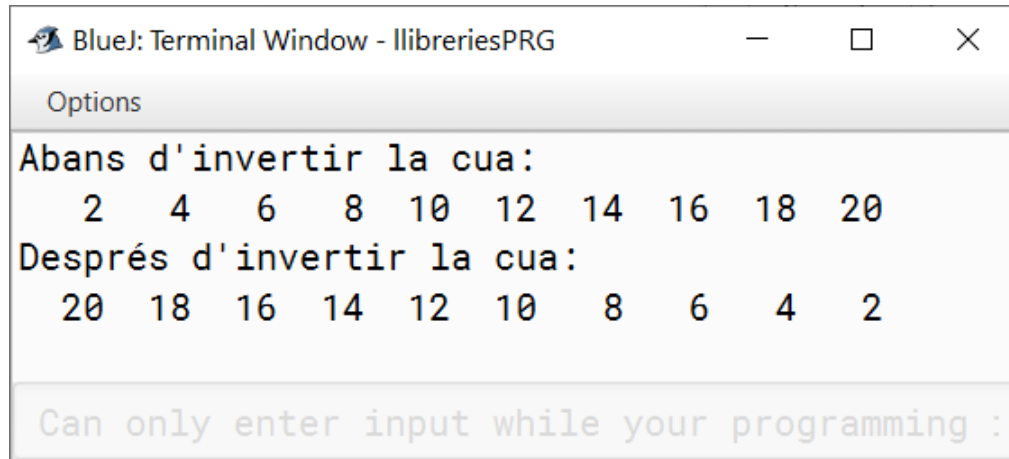


Cua retornada



Cues – Exemple

- En la classe programa *InvertirCua* del paquet *usaLinear*, es crea una cua amb els 10 primers números parells, es mostra per pantalla, s'inverteix l'ordre dels seus elements i es torna a mostrar per pantalla.



```
BlueJ: Terminal Window - lliberiesPRG
Options
Abans d'invertir la cua:
  2  4  6  8 10 12 14 16 18 20
Després d'invertir la cua:
 20 18 16 14 12 10  8  6  4  2

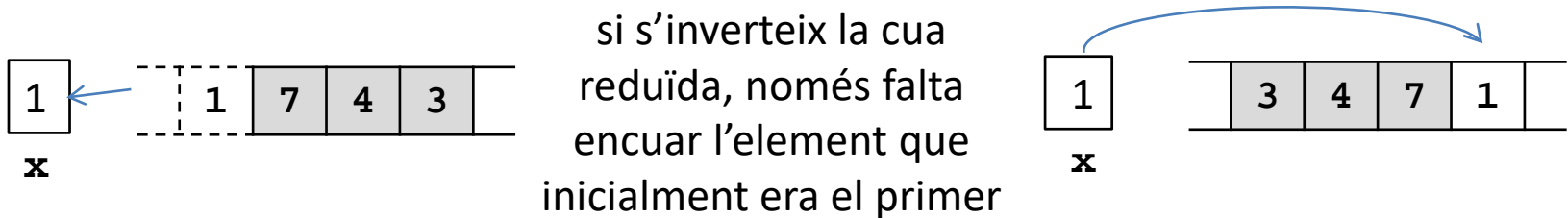
Can only enter input while your programming :
```

- La política **FIFO** de les cues afavoreix que el mètode que inverteix els elements de la cua es resolga **recursivament**.

Cues – Example

```
public static void invertRecur(QueueIntLinked q)
```

- Cas general: q té n elements, $n > 0$, es pot desencuar el primer element, reduint la cua a $n-1$ elements:



- Cas base: q està buida, està trivialment invertida.

```
public static void invertRecur(QueueIntLinked q) {  
    if (!q.empty()) {  
        int x = q.remove();  
        invertRecur(q);  
        q.add(x);  
    }  
}
```

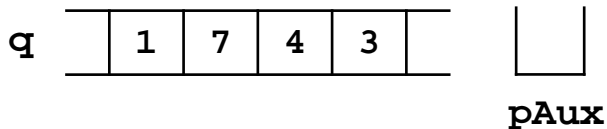
Cues – Example

- Per tal de resoldre el problema anterior iterativament, caldria usar alguna estructura auxiliar que emmagatzemara els elements a mesura que es desencuen de la cua, i que després facilitara recuperar-los en l'ordre invers a l'inicial:

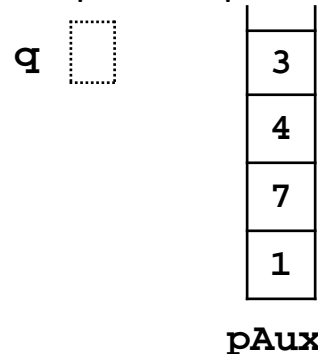
```
public static void invertIter(QueueIntLinked q) {  
    StackIntLinked pAux = new StackIntLinked();  
    while (!q.empty()) {  
        int x = q.remove();  
        pAux.push(x);  
    }  
    while (!pAux.empty()) {  
        int x = pAux.pop();  
        q.add(x);  
    }  
}
```

- Exemple:

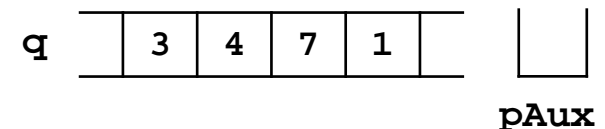
Inicialment:



Després del primer bucle:



Després del segon bucle:



Cues – Example

- La classe programa *InvertirPila*, del paquet *usaLinear*, inverteix una pila utilitzant com estructura de dades auxiliar una cua:

```
public static void invertIter(StackIntLinked p) {  
    QueueIntLinked qAux = new QueueIntLinked();  
    while (!p.empty()) {  
        int x = p.pop();  
        qAux.add(x);  
    }  
    while (!qAux.empty()) {  
        int x = qAux.remove();  
        p.push(x);  
    }  
}
```

BlueJ: Terminal Wind... Options

Abans d'invertir la pila:

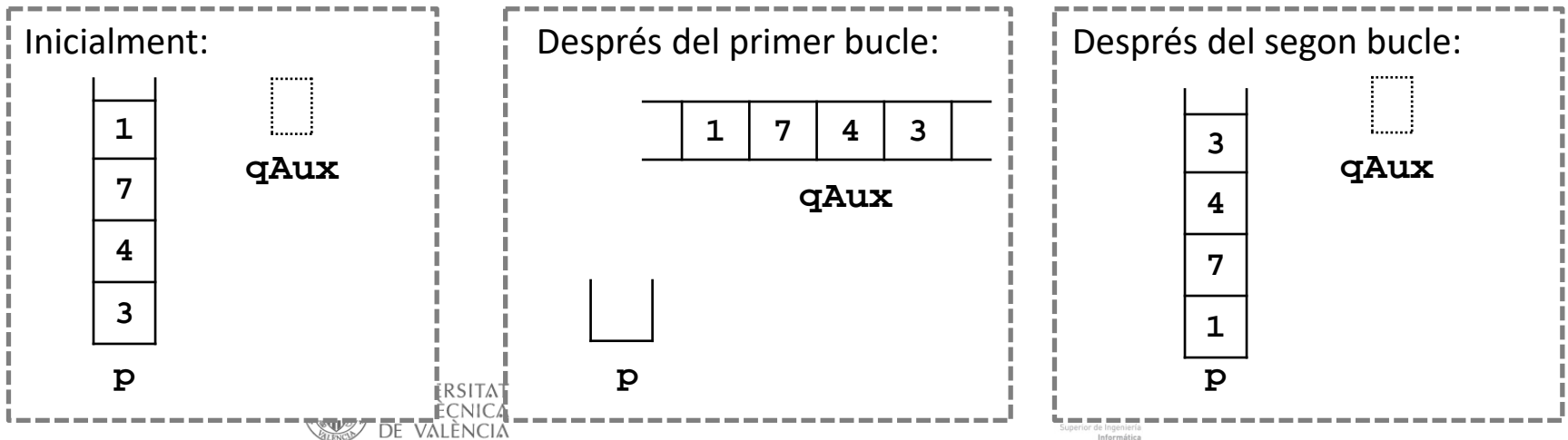
20
18
16
14
12
10
8
6
4
2

Després d'invertir la pila:

2
4
6
8
10
12
14
16
18
20

Can only enter input while yc

- Exemple:



Comparació Stack i Queue

MÈTODE RECURSIU

Stack

```
if (!p.empty()) {  
    int x = p.pop();  
    p.crida_rekursiva();  
    p.push(x);  
}  
else {  
    // tractar stack buida  
}
```

En finalitzar, la **Stack** p **no canvia**

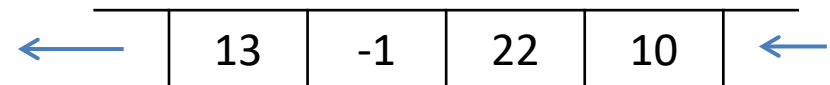
13
-1
22
10

Queue

```
if (!q.empty()) {  
    int x = q.remove();  
    q.crida_rekursiva();  
    q.add(x);  
}  
else {  
    // tractar queue buida  
}
```

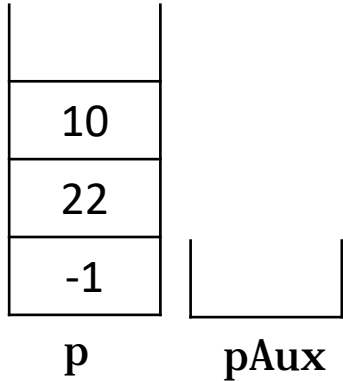


En finalitzar, la **Queue** q **s'ha invertit**



Comparació Stack i Queue

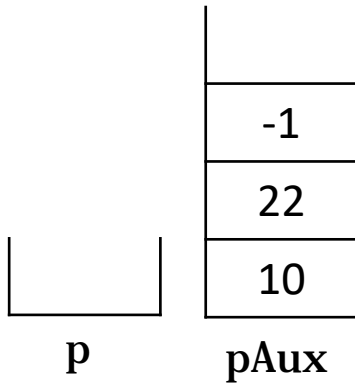
Inicialment



MÈTODE **ITERATIU**: **Stack**

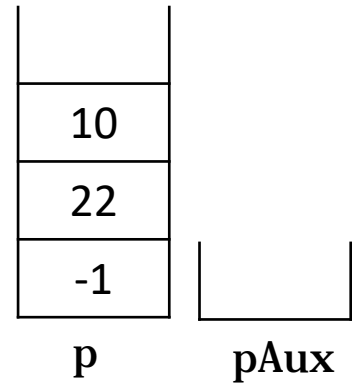
```
/* usant una pila auxiliar */  
Stack pAux = new Stack();  
while (!p.empty()) {  
    int x = p.pop();  
    // tractar p  
    pAux.push(x);  
}
```

Després del primer bucle



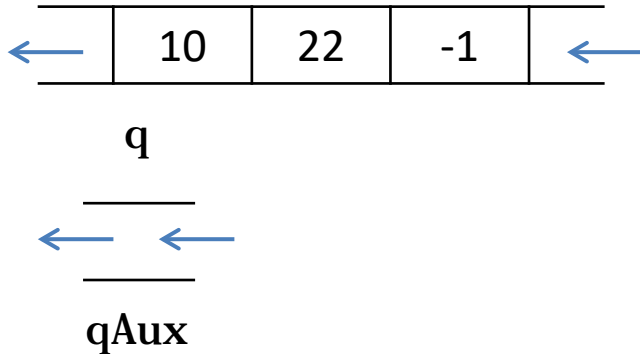
```
while (!pAux.empty()) {  
    int x = pAux.pop();  
    p.push(x);  
}
```

Després del segon bucle



Comparació Stack i Queue

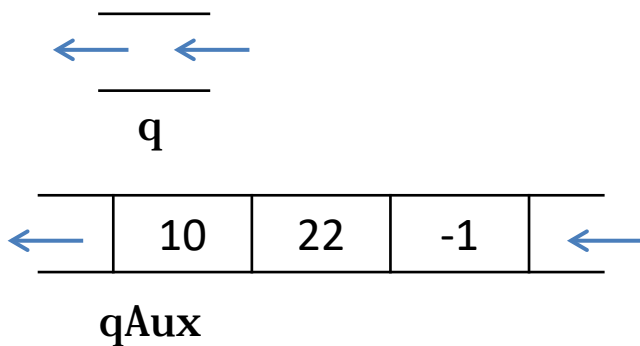
Inicialment



MÈTODE **ITERATIU**: Queue

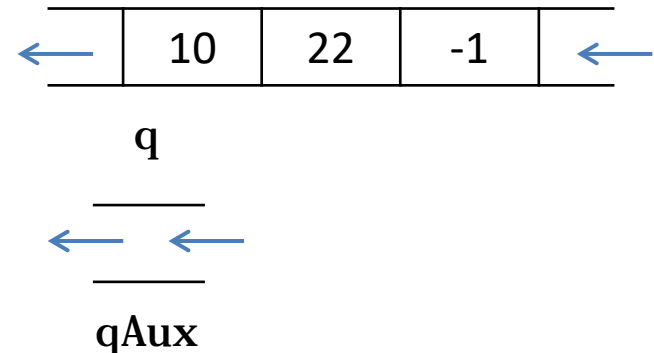
```
/* usant una cua auxiliar */  
Queue qAux = new Queue();  
while (!q.empty()) {  
    int x = q.remove();  
    // tractar q  
    qAux.add(x);  
}
```

Després del primer bucle



```
while (!qAux.empty()) {  
    int x = qAux.remove();  
    q.add(x);  
}
```

Després del segon bucle



Comparació Stack i Queue

MÈTODE ITERATIU: Queue

```
/* tenint en compte size */  
int n = q.size();  
while (n > 0) {  
    int x = q.remove();  
    // tractar q  
    q.add(x);  
    n--;  
}
```

Inicialment



q

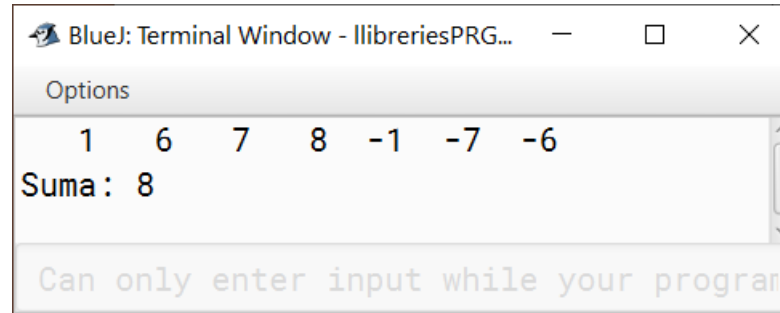
En finalitzar



q

Cues – Exercicis

- La classe **SumarCua** del paquet **usaLinear** crea una cua d'enters i suma les seues dades.



```
BlueJ: Terminal Window - lliberiesPRG...
Options
1 6 7 8 -1 -7 -6
Suma: 8
Can only enter input while your program is running
```

- Completa**, en aquesta classe, els següents mètodes :
- `private static int sumar(QueueIntLinked q)` que suma els elements de `q` fent ús d'una cua auxiliar.
- `private static int sumar2(QueueIntLinked q)` que suma els elements de `q` (sense usar cap cua auxiliar) tenint en compte el seu nombre d'elements (**size**).

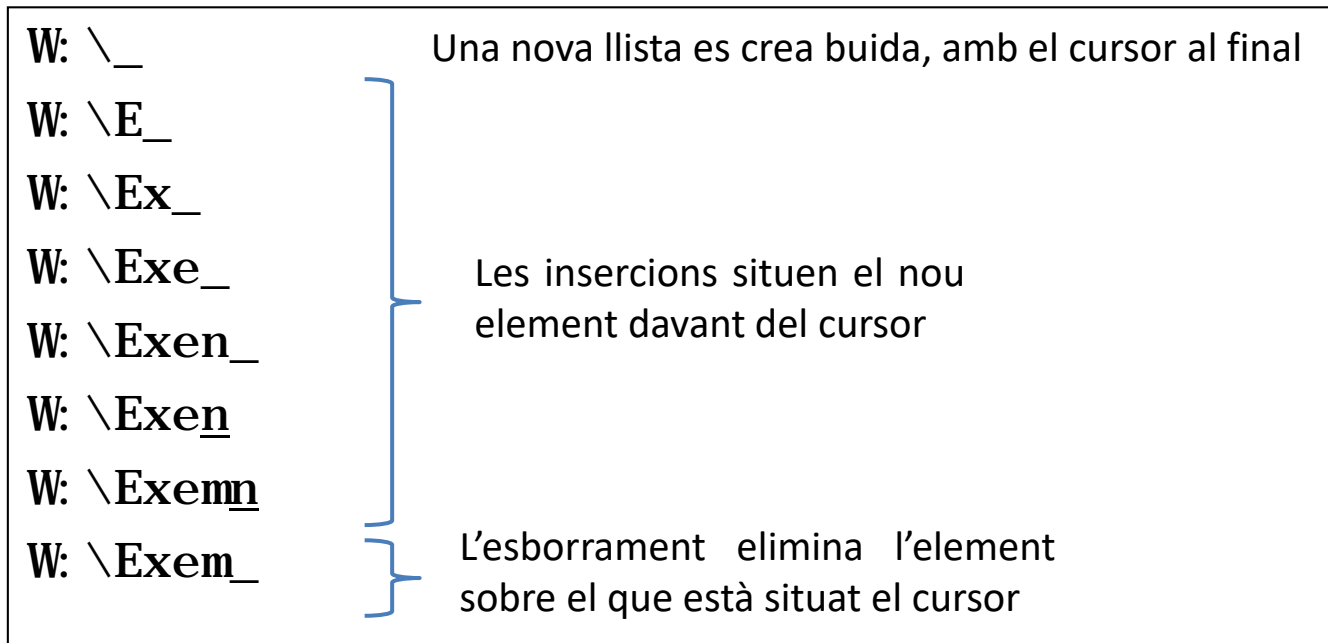
Llistes amb punt d'interès

- Una *llista* (*list*) és una seqüència en la que l'accés pot realitzar-se en qualsevol punt.
- Una *llista amb punt d'interès* és una llista en la que es suposa que hi ha una posició destacada o punt d'accés, el punt d'interès o cursor.
- Si la llista té n dades, $n \geq 0$, numerades de 0 a $n-1$, el punt d'interès pot estar:
 - en una posició $0 \leq i \leq n-1$: *sobre l'element i-èsim*
 - en la posició $i = n$: *a la dreta del tot o final de la llista*i es pot moure per la llista posició a posició.
- Exemple: Llista de caràcters en la línia de comandaments del sistema. El cursor és el punt d'interès.



Llistes amb punt d'interès

- En una *llista amb punt d'interès* (**Li stPI**) les operacions d'inserir una dada, esborrar una dada, etc ... es refereixen al punt d'interès (abreviadament, PI) o cursor:



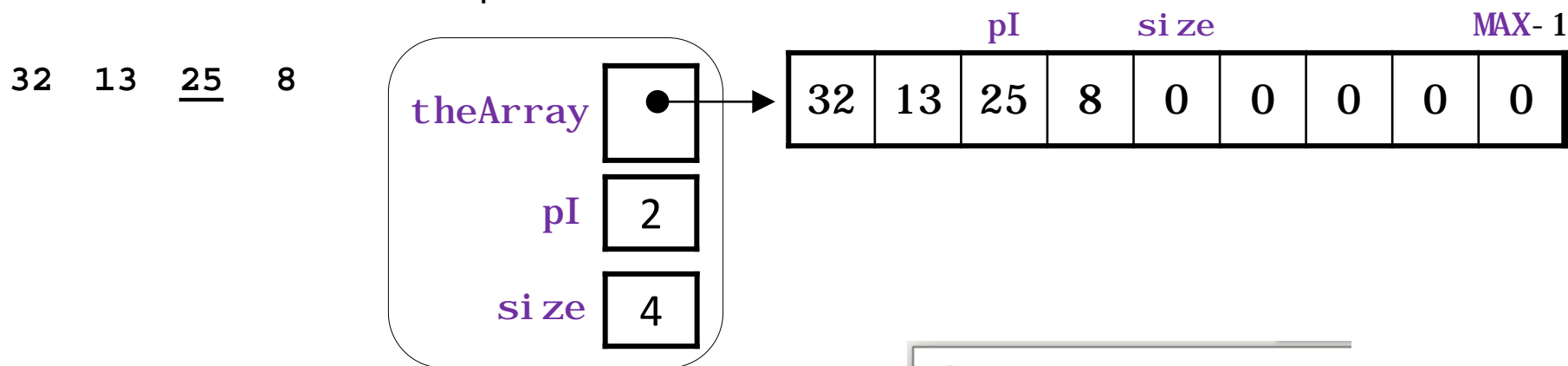
- Habitualment, el cursor pot moure's a dreta i esquerra.
- En el model de llista amb punt d'interès que es va a presentar, el cursor només es pot moure cap a la dreta, o situar-lo a l'inici.


Llista amb PI - Operacions

Operació	Descripció
<code>public ListPI ()</code>	Crea una nova <code>ListPI</code> buida.
<code>public void begin()</code>	Situa el cursor al principi de la <code>ListPI</code> .
<code>public void next()</code>	Desplaça el cursor una posició a la dreta de la seua posició actual. Llança <code>NoSuchElementException</code> si el cursor està al final de la <code>ListPI</code> .
<code>public void insert(Tipus e)</code>	Insereix <code>e</code> davant del punt d'interès de la <code>ListPI</code> (el cursor queda a la dreta de <code>e</code>).
<code>public Tipus remove()</code>	Torna i elimina l'element del punt d'interès de la <code>ListPI</code> , sense moure el cursor. Llança <code>NoSuchElementException</code> si el cursor està al final de la <code>ListPI</code> .
<code>public Tipus get()</code>	Torna (sense eliminar-lo) l'element que està en el cursor. Llança <code>NoSuchElementException</code> si el cursor està al final de la <code>ListPI</code> .
<code>public boolean empty()</code>	Torna <code>true</code> si la <code>ListPI</code> està buida i <code>false</code> en cas contrari.
<code>public boolean isEnd()</code>	Torna <code>true</code> si el cursor està al final de la <code>ListPI</code> i <code>false</code> en cas contrari.
<code>public int size()</code>	Torna el nombre d'elements de la <code>ListPI</code> .

Llista amb PI – Implementació amb arrays

- Una llista amb punt d'interès (**ListPI**) pot implementar-se usant un array (**theArray**) que emmagatzeme les dades, un índex (**pI**) que marca el punt d'interès o cursor i una constant que defineixca la dimensió màxima de l'array (**MAX**).
- Els elements de la llista ocupen les primeres posicions consecutives a l'array. Es presenta una llista amb dades de tipus `int`:



 ListPIIntArray - llibreriesPRG

```
public class ListPIIntArray {  
    private static final int MAX = ...;  
    private int[] theArray;  
    private int pI, size;  
  
    // Implementació de les operacions:  
    ...  
}
```

Llista amb PI – Implementació amb arrays

- Operació constructora `ListPIIntArray`: Crea la llista buida.

```
public ListPIIntArray() {  
    theArray = new int[MAX];  
    size = 0;  
    pI = 0;  
}
```

- Operació consultora `empty`:

```
public boolean empty() { return size == 0; }
```

- Operació consultora `isEnd`:

```
public boolean isEnd() { return pI == size; }
```

- Operació consultora `size`:

```
public int size() { return size; }
```

Llista amb PI – Implementació amb arrays

- Operació **begin**: Situa el PI en la primera posició de la llista.

```
public void begin() { pI = 0; }
```

- Operació **next**: Desplaça el PI una posició a la dreta. Requereix que el PI no es trobe al final de la llista. En aquest cas, llança l'excepció *unchecked NoSuchElementException*.

```
public void next() {  
    if (pI == size) {  
        throw new NoSuchElementException("PI at the end");  
    }  
    pI++;  
}
```

- Operació consultora **get**: Torna l'element en el PI. Requereix que el PI no es trobe al final de la llista. En aquest cas, llança l'excepció *unchecked NoSuchElementException*.

```
public int get() {  
    if (pI == size) {  
        throw new NoSuchElementException("PI at the end");  
    }  
    return theArray[pI];  
}
```


Llista amb PI – Implementació amb arrays

- Operació **insert**: Insereix l'element *x* en el PI. Hi ha que desplaçar a la dreta totes les dades des del PI fins el final de la llista. El PI queda a la dreta de *x*.

```
public void insert(int x) {  
    if (size == theArray.length) { duplicateArray(); }  
    for (int k = size - 1; k >= pI; k--) {  
        theArray[k + 1] = theArray[k];  
    }  
    theArray[pI] = x;  
    pI++;  
    size++;  
}
```

```
private void duplicateArray() {  
    int[] aux = new int[2 * theArray.length];  
    for (int i = 0; i < theArray.length; i++) {  
        aux[i] = theArray[i];  
    }  
    theArray = aux;  
}
```

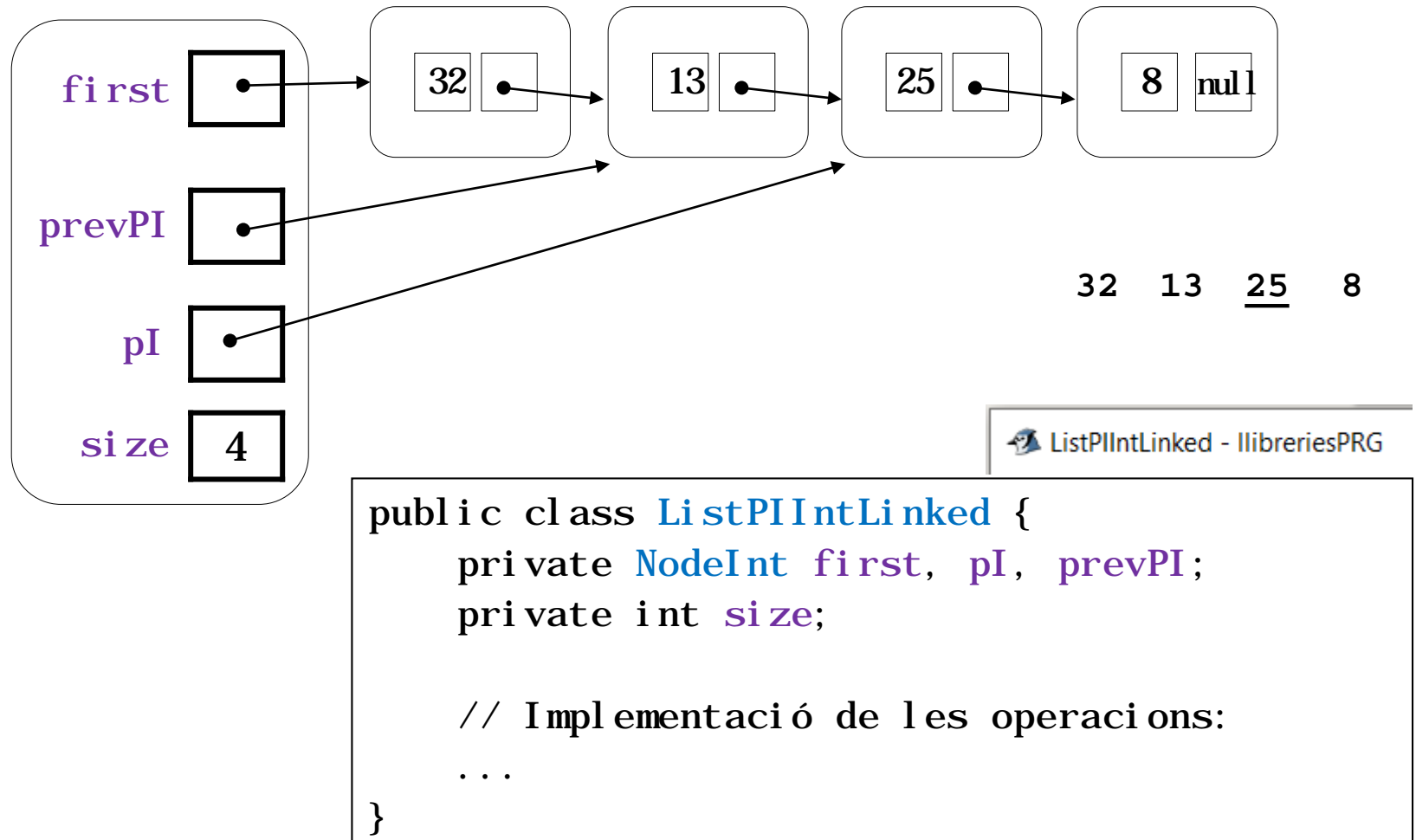
Llista amb PI – Implementació amb arrays

- Operació **remove**: Torna i elimina de la llista l'element en el PI, sense moure'l. Hi ha que desplaçar a l'esquerra totes les dades des del PI fins el final de la llista. Requereix que el PI no es trobe al final de la llista. En aquest cas, llança l'excepció *unchecked NoSuchElementException*.

```
public int remove() {  
    if (pI == size) {  
        throw new NoSuchElementException("PI at the end");  
    }  
    int x = theArray[pI];  
    for (int k = pI + 1; k < size; k++) {  
        theArray[k - 1] = theArray[k];  
    }  
    size--;  
    return x;  
}
```

Llista amb PI – Implementació enllaçada

- En una **implementació enllaçada**, la llista amb punt d'interès tindrà tres atributs per a mantindre les referències al primer element, al punt d'interès i al seu antecessor.



Llista amb PI – Implementació enllaçada

- Operació constructora `ListPIIntLinked`: Crea la llista buida.

```
public ListPIIntLinked() {  
    first = null;  
    pI = null;  
    prevPI = null;  
    size = 0;  
}
```

- Operació consultora `empty`:

```
public boolean empty() { return size == 0; }
```

- Operació consultora `isEnd`:

```
public boolean isEnd() { return pI == null; }
```

- Operació consultora `size`:

```
public int size() { return size; }
```

Llista amb PI – Implementació enllaçada

- Operació **begin**: Situa el PI en la primera posició de la llista.

```
public void begin() { pI = first; prevPI = null; }
```

- Operació **next**: Desplaça el PI al següent node. Requereix que el PI no es trobe al final de la llista. En aquest cas, llança l'excepció *unchecked NoSuchElementException*.

```
public void next() {  
    if (pI == null) {  
        throw new NoSuchElementException("PI at the end");  
    }  
    prevPI = pI;  
    pI = pI.next;  
}
```

- Operació consultora **get**: Torna l'element en el PI. Requereix que el PI no es trobe al final de la llista. En aquest cas, llança l'excepció *unchecked NoSuchElementException*.

```
public int get() {  
    if (pI == null) {  
        throw new NoSuchElementException("PI at the end");  
    }  
    return pI.data;  
}
```

Llista amb PI – Implementació enllaçada

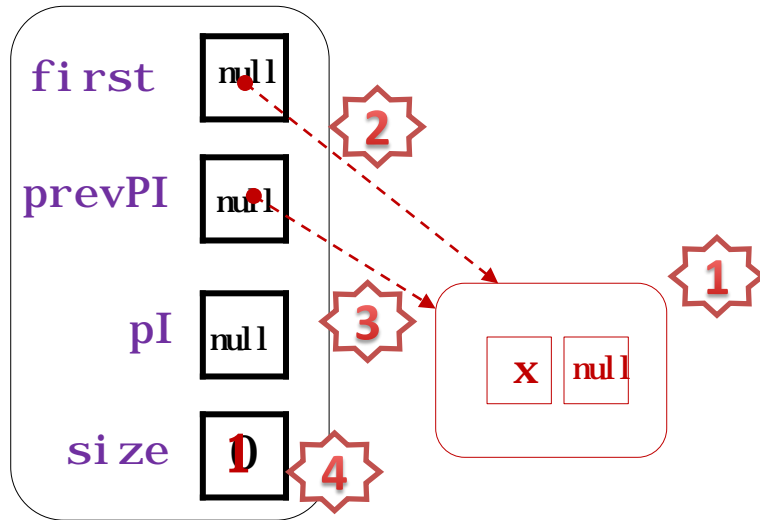
- Operació **insert**: Insereix un nou node amb l'element **x** en la posició anterior al PI. Si el cursor està a l'inici, el nou node serà el primer de la llista. En qualsevol altre cas, el nou node s'insereix entre el node cursor i el seu node predecessor.

```
public void insert(int x) {  
    if (pI == first) {  
        first = new NodeInt(x, pI);  
        prevPI = first;  
    }  
    else {  
        prevPI.next = new NodeInt(x, pI);  
        prevPI = prevPI.next;  
    }  
    size++;  
}
```

Llista amb PI – Implementació enllaçada

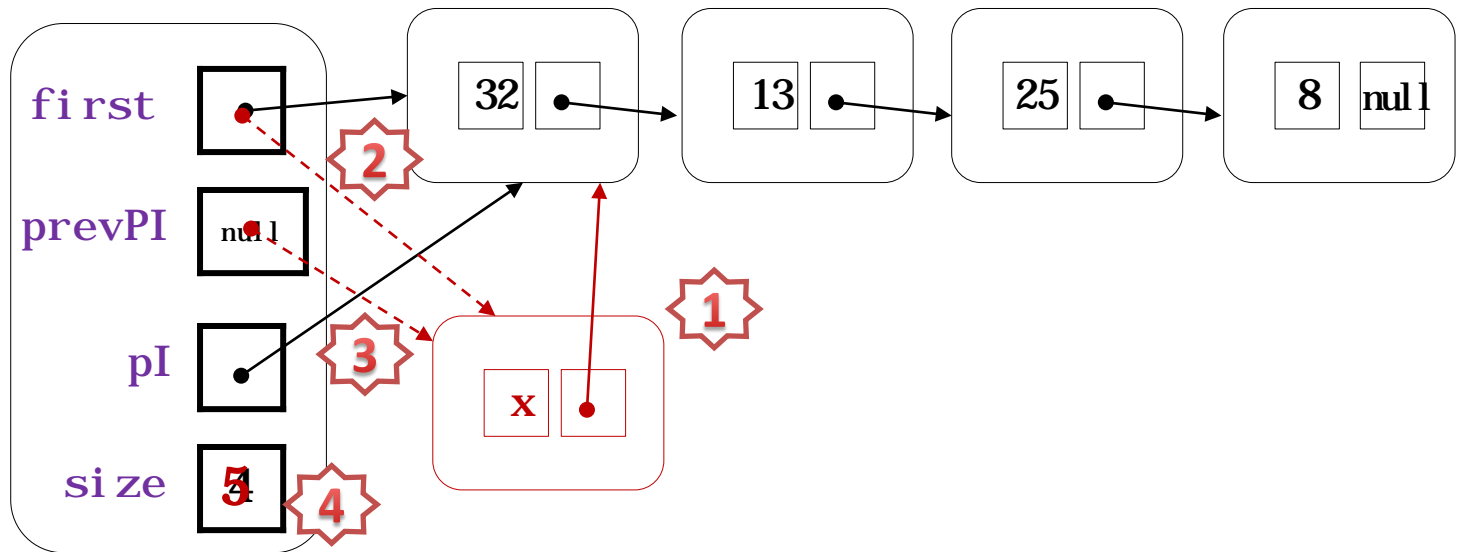
Cas 1: $pI == first$

A



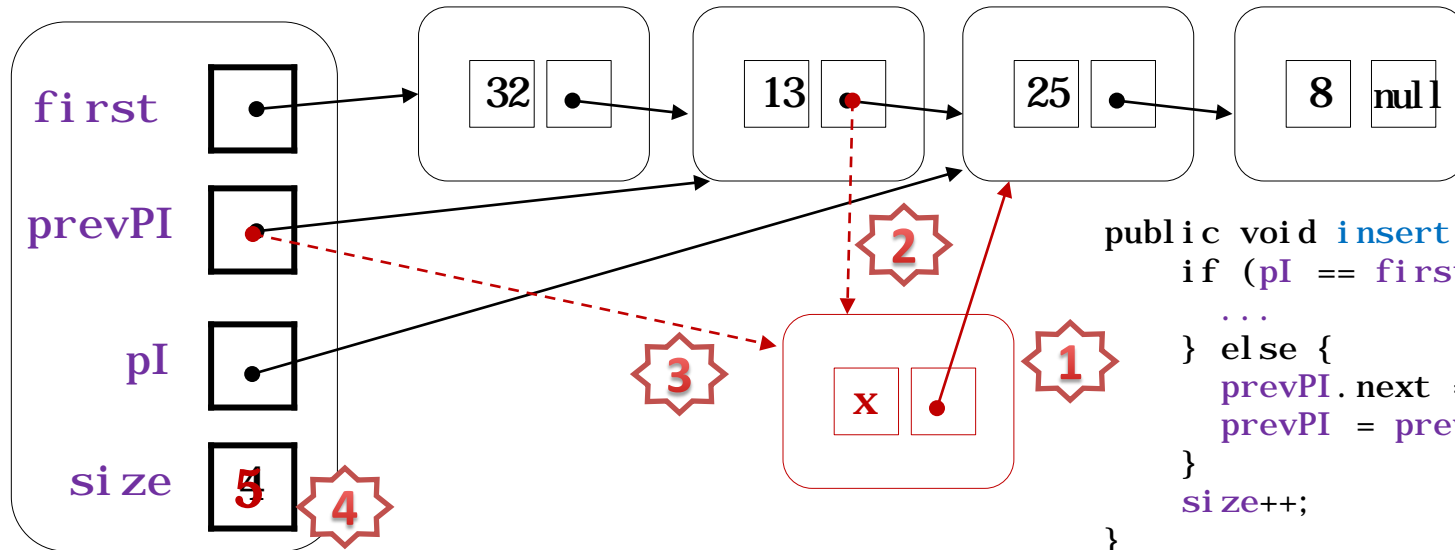
```
public void insert(int x) {  
    if (pI == first) {  
        first = new NodeInt(x, pI);  
        prevPI = first;  
    } else {  
        ...  
    }  
    size++;  
}
```

B

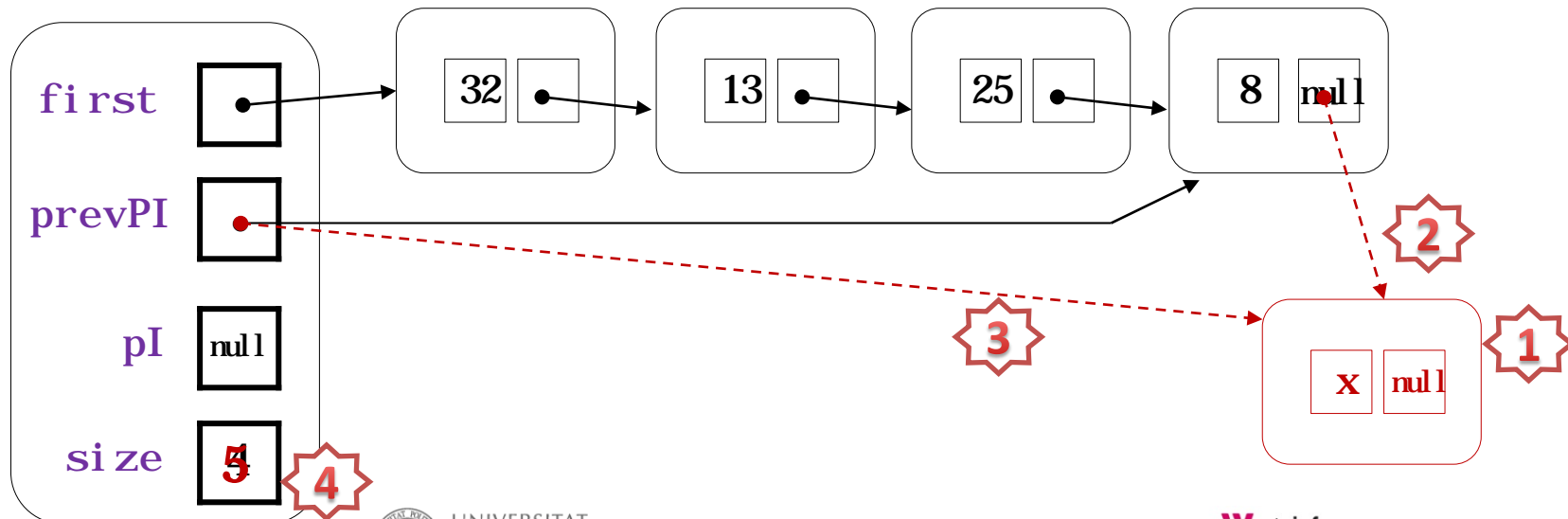


Llista amb PI – Implementació enllaçada

Cas 2: $pI \neq first$



A



B

Llista amb PI – Implementació enllaçada

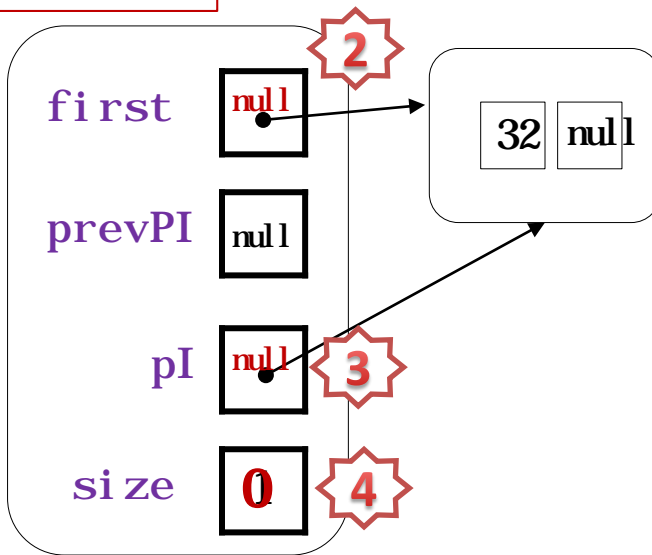
- Operació **remove**: Torna i elimina de la llista l'element en el PI. Si el cursor està a l'inici, **first** s'actualitza al seu node següent. En qualsevol altre cas, els nodes anterior i posterior al node cursor queden enllaçats. Requereix que el PI no es trobe al final. En aquest cas, llança l'excepció *unchecked NoSuchElementException*.

```
public int remove() {  
    if (pI == null) {  
        throw new NoSuchElementException("PI at the end");  
    }  
    int x = pI.data;  
    if (pI == first) { first = first.next; }  
    else { prevPI.next = pI.next; }  
    pI = pI.next;  
    size--;  
    return x;  
}
```

Llista amb PI – Implementació enllaçada

Cas 1: $pI == first$

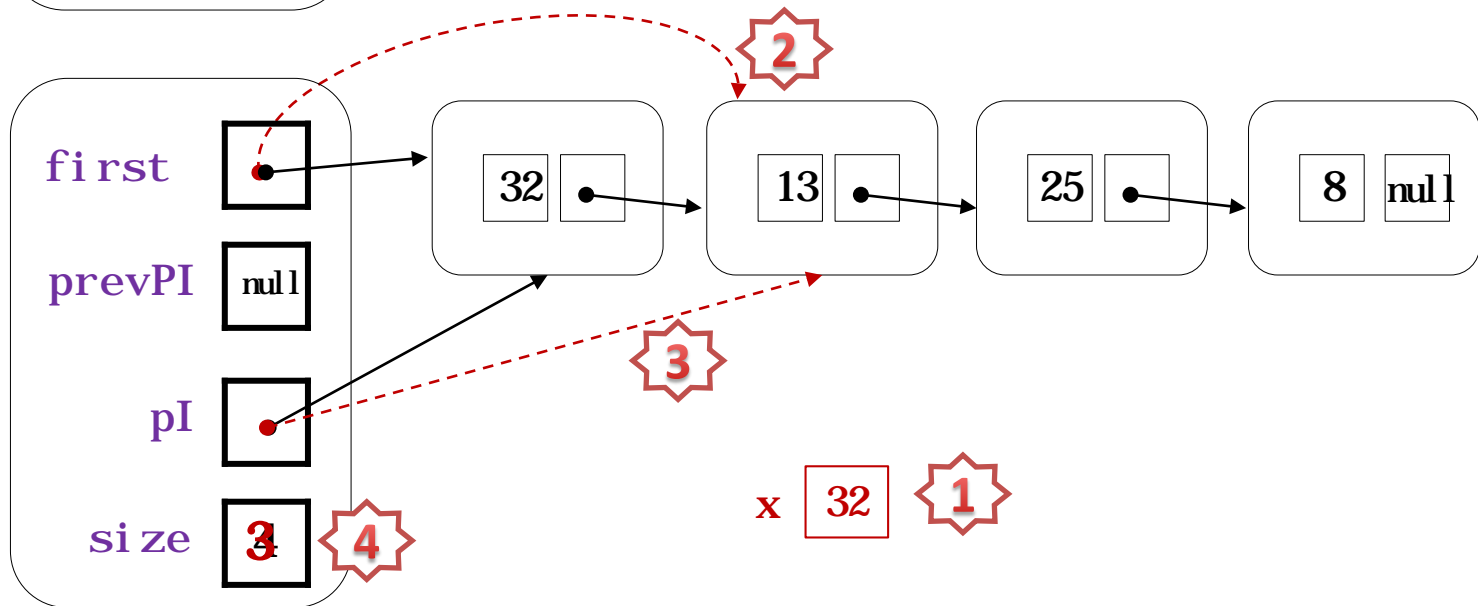
A



```
public int remove() {
    if (pI == null) throw ...
    int x = pI.data;
    if (pI == first) {
        first = first.next;
    }
    else ...
    pI = pI.next;
    size--;
    return x;
}
```

x 32 1

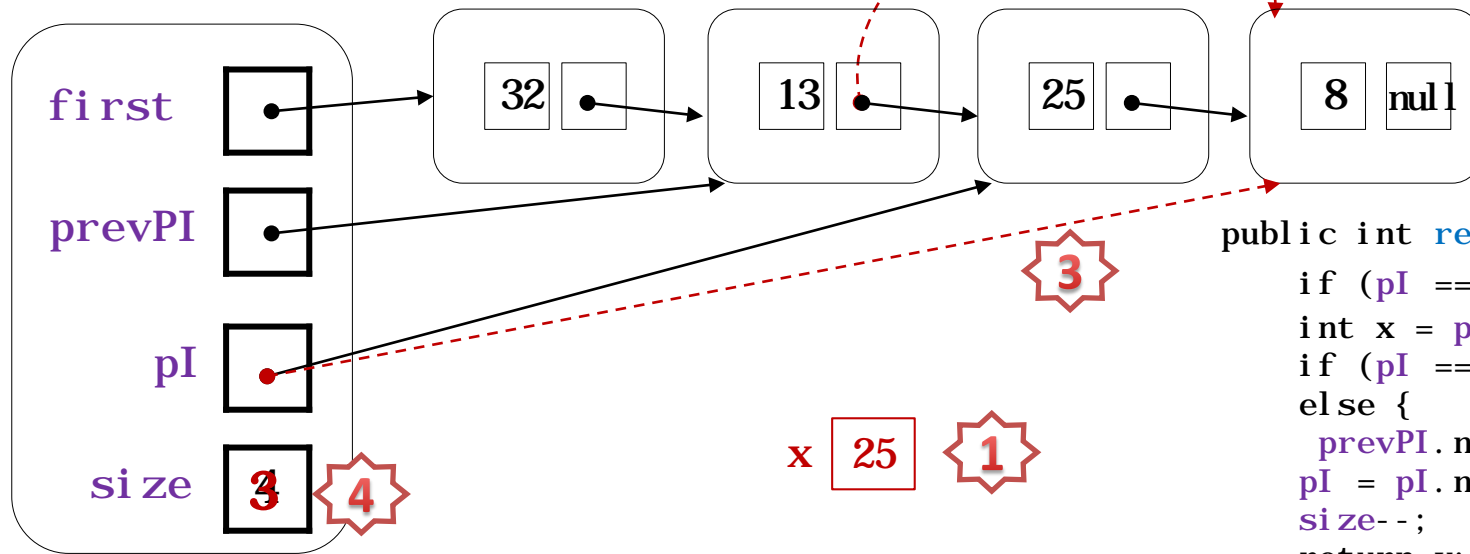
B



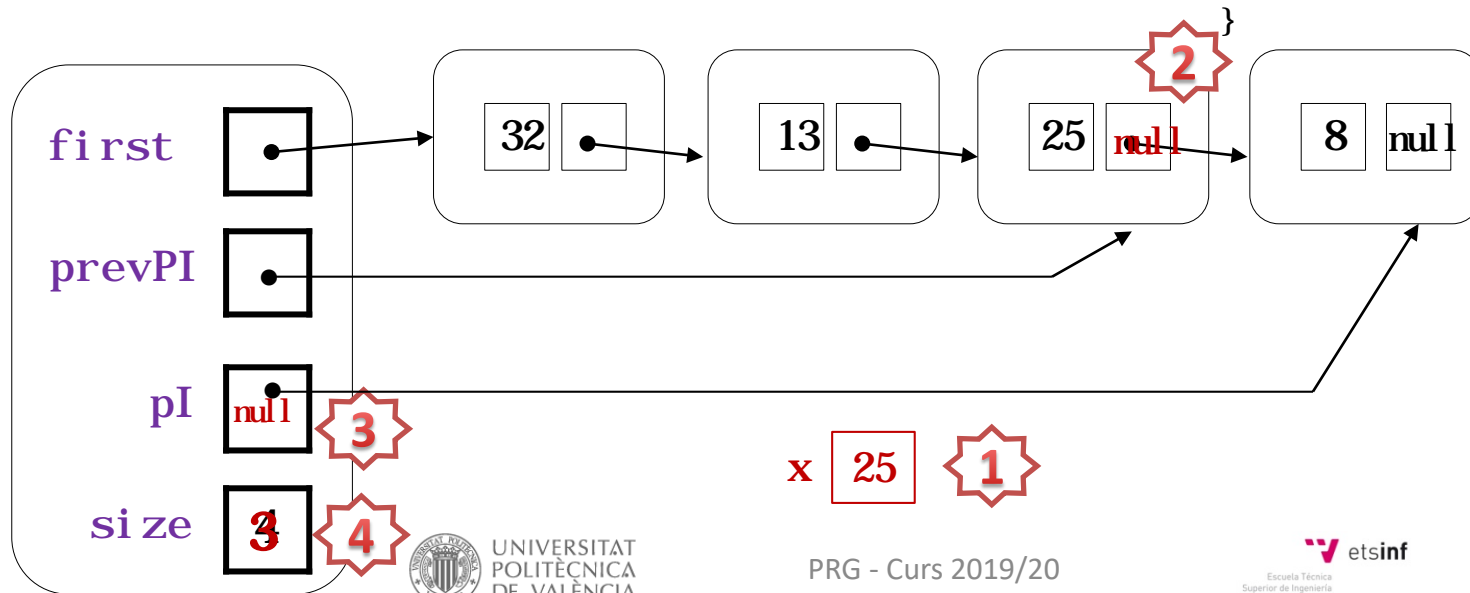
x 32 1

Llista amb PI – Implementació enllaçada

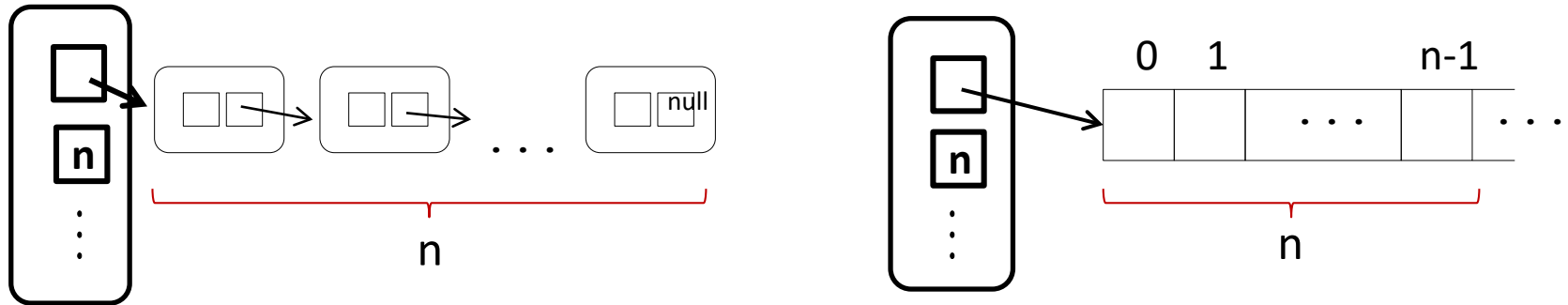
Cas 2: $pI \neq first$



```
public int remove() {  
    if (pI == null) { throw ... }  
    int x = pI.data;  
    if (pI == first) ...  
    else {  
        prevPI.next = pI.next; }  
    pI = pI.next;  
    size--;  
    return x;  
}
```




Llista amb PI – Comparació d'implementacions




Operació	Representació enllaçada	Representació amb arrays
remove()	$\Theta(1)$	$O(n), \Omega(1)$
insert(int x)	$\Theta(1)$	$O(n), \Omega(1)$

- La resta d'operacions definides per a llistes amb punt d'interès tenen el mateix cost constant, $\Theta(1)$, en les dues representacions.


Llista amb PI – Examples


 ListPIIntArray - llibreriesPRG

 ListPIIntLinked - llibreriesPRG

- **Revisa** els mètodes `toString()` i `equals(Object)` de les classes *ListPIIntLinked* i *ListPIIntArray*
- **Revisa** la classe *TestListPIInt* i executa-la. Et servirà per provar els mètodes de les classes *ListPIIntLinked* i *ListPIIntArray*

Llista amb PI – Exemples

 ListPIIntArray - llibreriesPRG

 ListPIIntLinked - llibreriesPRG

- Es desitja que la llista amb punt d'interès tinga els següents mètodes públics:
 - a) Mètode **search**(int) que indique si x apareix en la llista. Si x es troba, situa el PI en la primera ocurrència de x. Si no apareix, el PI no es mou.
 - b) Mètode **searchFromPI**(int) que indique si x apareix en la llista des de la posició del PI inclusivament en endavant. Si x apareix, avança el PI a x. Si no apareix, el PI no es mou.
- **Revisa** la seua implementació en les classes **ListPIIntLinked** i **ListPIIntArray**

```
public boolean search(int x) {  
    return search(first, x); // cerca a partir de first  
}  
  
public boolean searchFromPI(int x) {  
    return search(PI, x); // cerca a partir de pI  
}
```

Llista amb PI – Examples

- El mètodes `search(int)` i `searchFromPI(int)` criden al següent mètode auxiliar que busca la primera ocurrència de `x` des del node `aux` endavant; si el troba, mou el PI al node que conté a `x`. Si no apareix, el PI no es mou.

```
private boolean search(NodeInt aux, int x) {  
    NodeInt ant = null;  
    if (aux == this.pI) { ant = this.prevPI; }  
    while (aux != null && aux.data != x) {  
        ant = aux;  
        aux = aux.next;  
    }  
    boolean res = false;  
    if (aux != null) {  
        this.pI = aux;  
        this.prevPI = ant;  
        res = true;  
    }  
    return res;  
}
```

Llista amb PI – Exercicis

- La classe **ListPIIntLinkedPlus** és una classe derivada de **ListPIIntLinked**. Els seus mètodes s'implementen **fent ús exclusivament de les operacions** de la classe **ListPIIntLinked** (sense accedir a la seua representació interna).
- A mesura que vages completant-los, pots provar-los amb la classe programa **TestListPIIntLinkedPlus**.
- **Completa**, en la classe **ListPIIntLinkedPlus**, el mètode **search(int)** (similar al mètode **search(int)** de **ListPIIntLinked**) que torne true si l'element x es troba en la llista. En aquest cas, situa el punt d'interès en la primera ocurrència de x en la llista. En cas contrari, el punt d'interès de la llista es queda al final i torna false.

```
public boolean search(int x) {  
    // COMPLETAR: situar el PI a l'inici  
  
    // COMPLETAR: bucle de cerca de la primera ocurrència de x en la llista  
  
    // COMPLETAR: si cerca amb èxit, tornar true, sino tornar false  
    return true;  
}
```


Llista amb PI – Exercicis

- **Completa**, en la classe *ListPIIntLinkedPlus*, un mètode `delete(int)` que esborre l'element `x` de la llista. Si `x` apareix més d'una vegada, esborra la primera ocurrència de `x` en la llista. Si `x` no es troba, no fa res. Aquest mètode ha d'implementar-se **fent ús exclusivament de les operacions** de la classe *ListPIIntLinked* (sense accedir a la seua representació interna).

```
public void delete(int x) {  
    // COMPLETAR: situar el PI a l'inici  
  
    // COMPLETAR: bucle de cerca de la primera ocurrència de x en la llista  
  
    // COMPLETAR: si cerca amb èxit, eliminar x, sino no fer res  
  
}
```

- També es pot resoldre, de manera equivalent, fent ús del mètode `search(int)` implementat anteriorment.

Llista amb PI – Exercicis

- **Completa** en la classe *ListPIIntLinked* un mètode amb el següent perfil:

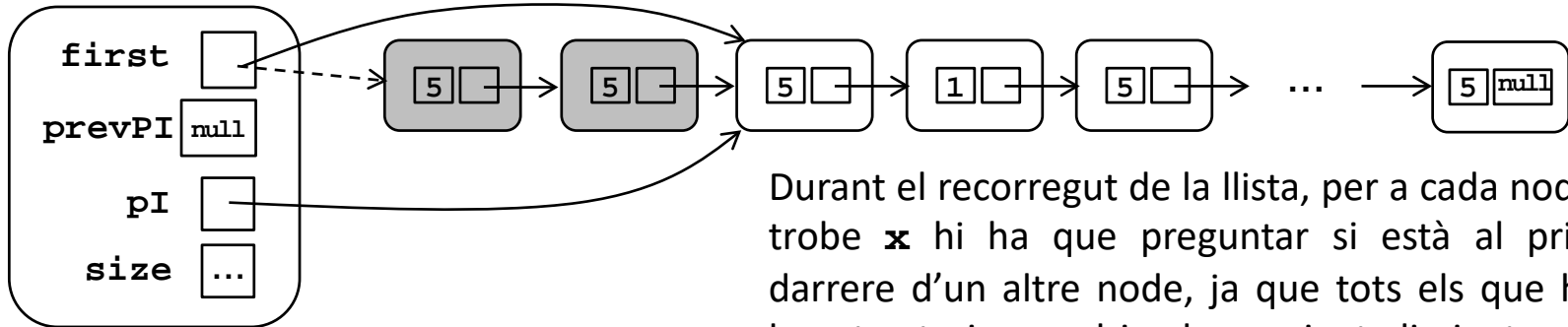
```
/** Precondició: PI != first */  
public void anterior()
```

que canvia el PI a la posició anterior a l'actual. L'operació no està definida si el PI està sobre el primer element de la llista.

```
public void anterior() {  
    NodeInt aux = first, ant = null;  
    /* COMPLETAR bucle per a situar aux en prevPI */  
  
    // quan acaba el bucle, aux és prevPI i ant és l'anterior  
    /* COMPLETAR actualitzar pI i prevPI */  
}
```

Llista amb PI – Exercicis

- **Completa** en la classe *ListPIIntLinked* un mètode `removeAll(int)` que, donat **x**, elimine totes les ocurrences de **x** de la llista, deixant el cursor o PI al final.



Durant el recorregut de la llista, per a cada node on es trobe **x** hi ha que preguntar si està al principi o darrere d'un altre node, ja que tots els que haguera hagut anteriors podrien haver sigut eliminats.

x 5

```
public void removeAll(int x) {  
    // s'usen pI i prevPI per anar visitant els nodes  
    pI = first; prevPI = null;  
    while (pI != null) {  
        if (pI.data == x) {  
            if (pI == first) {  
                /* COMPLETAR: actualitzar first */  
            } else {  
                /* COMPLETAR: actualitzar el següent de prevPI */  
            }  
            /* COMPLETAR: avançar pI */  
            size--;  
        } else {  
            /* COMPLETAR: avançar prevPI i pI */  
        }  
    }  
}
```

Llista amb PI – Exercicis

- **Completa** en la classe **ListPIIntLinkedPlus** un mètode **removeAll(int)** que, donat **x**, elimine totes les ocurrences de **x** de la llista, deixant el cursor o PI al final del tot. Aquest mètode ha d'implementar-se **fent ús exclusivament de les operacions** de la classe **ListPIIntLinked** (sense accedir a la seua representació interna).

```
public void removeAll(int x) {  
    // COMPLETAR: situar el PI a l'inici  
  
    // COMPLETAR: bucle de recorregut de totes les dades de la llista  
    // COMPLETAR: si la dada del PI és x, eliminar-la  
    //             sino, avançar el PI  
  
}
```

Llista amb PI – Exercicis

- **Completa** en la classe **ListPIIntLinked** un mètode amb el següent perfil:

/ Precondició: la llista està ordenada ascendentment */**

public void insertSort(int x)

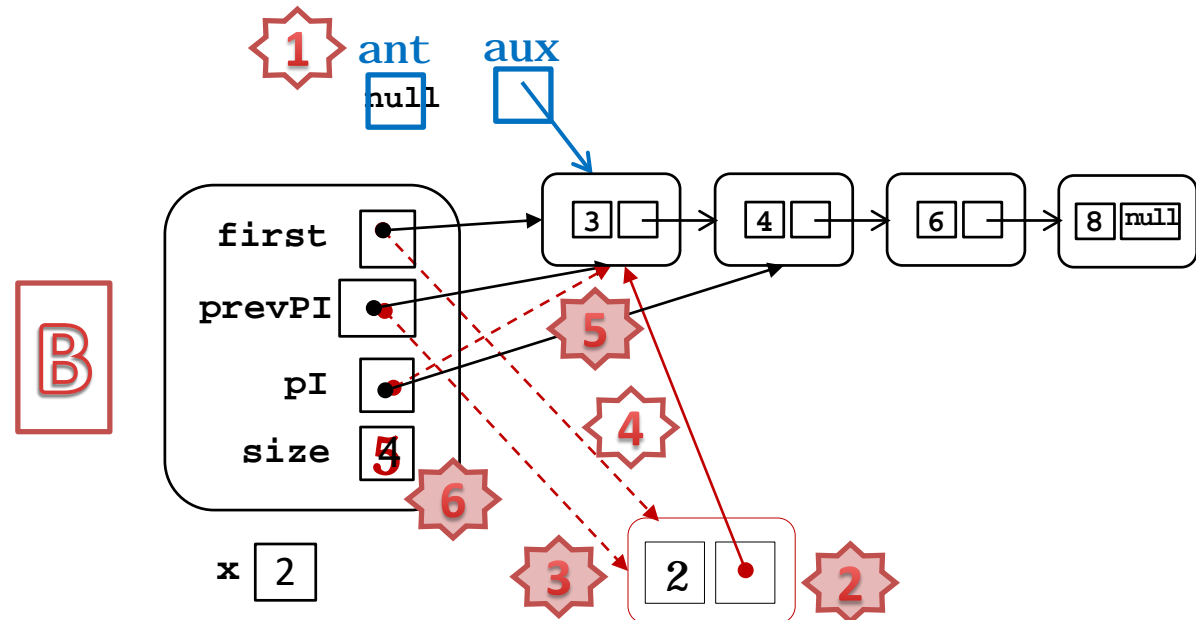
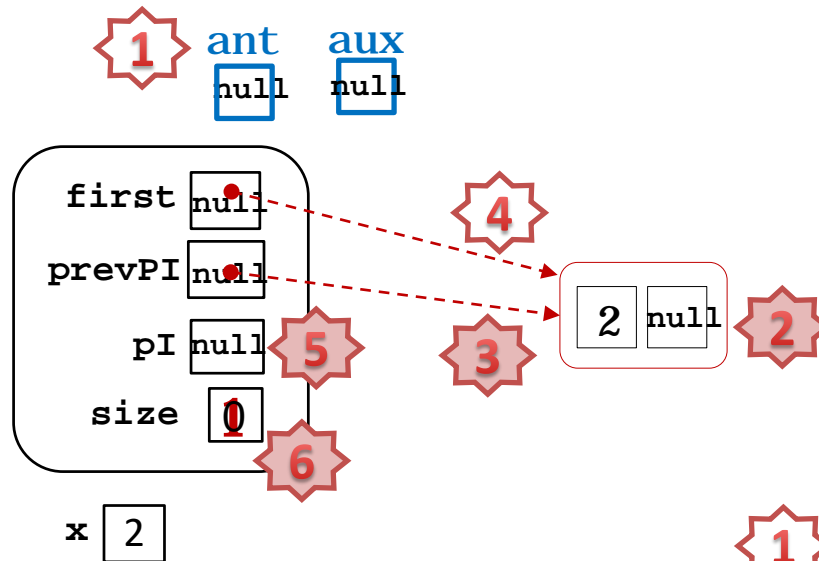
per tal d'**inserir ordenadament** un nou element en la llista amb punt d'interès, suposant definida la relació d'ordre \leq entre els seus elements.

```
public void insertSort(int x) {  
    NodeInt aux = first, ant = null;  
    /* COMPLETAR bucle de cerca del primer node amb dada>=x */  
  
    // quan acaba el bucle: aux és el node amb dada>=x  
    // o és null si tots els elements són <=x  
    // El nou node s'ha d'inserir entre ant i aux.  
    // en la posició anterior al PI.  
    /* COMPLETAR 2 crear el nou node i actualitzar prevPI, 3  
       first quan corresponga, pI i size */  
  
    4          5          6  
}
```

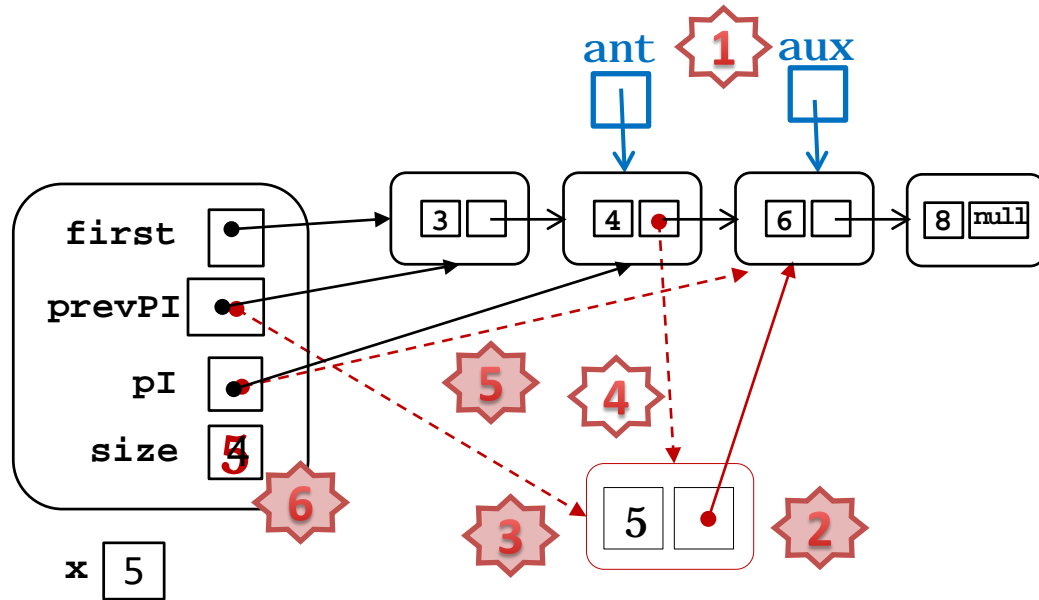
- **Cas 1:** x és < que tots els elements de la llista, és a dir, ha de ser el primer de la llista.

A: `empty()`

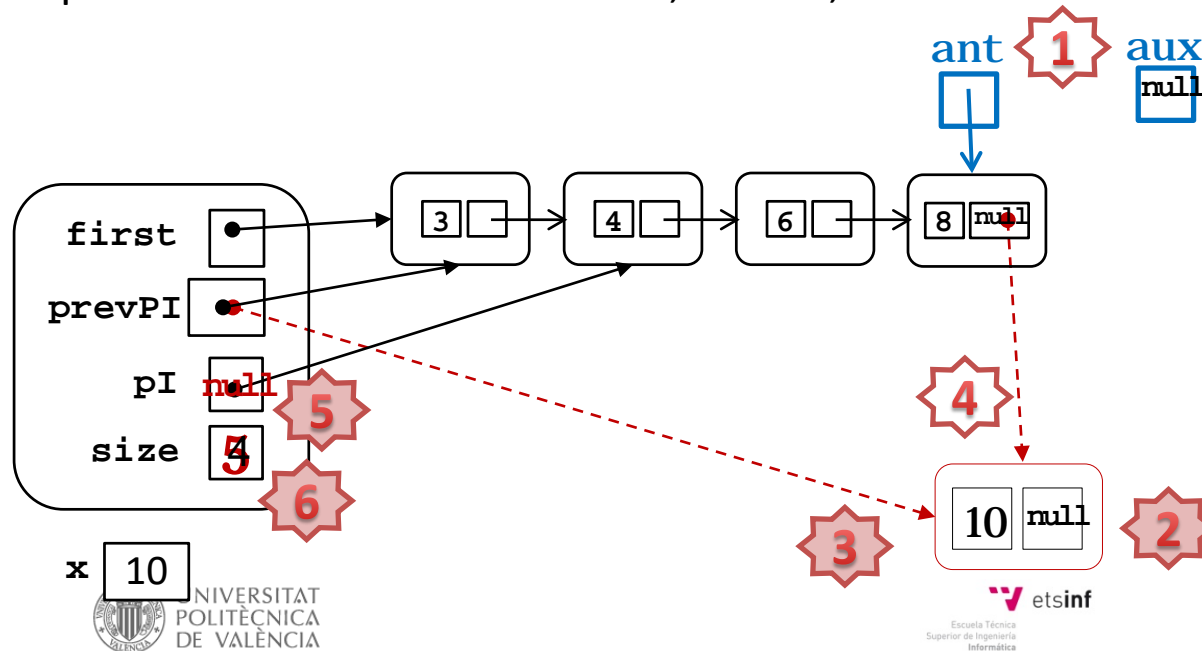
B: `!empty()`



- **Cas 2:** x és \leq que algun element intermedi de la llista.



- **Cas 3:** x és \geq que tots els elements de la llista, és a dir, ha de ser l'últim de la llista.



Llista amb PI – Exercicis

- **Completa** en la classe **ListPIIntLinkedPlus** un mètode amb el següent perfil:

```
/** Precondició: la llista està ordenada ascendentment */  
public void insertSort(int x)
```

per tal d'**inserir ordenadament** un nou element en la llista amb punt d'interès, suposant definida la relació d'ordre \leq entre els seus elements. Aquest mètode ha d'implementar-se **fent ús exclusivament de les operacions** de la classe **ListPIIntLinked** (sense accedir a la seua representació interna).

```
public void insertSort(int x) {  
    // COMPLETAR: situar el PI a l'inici  
  
    // COMPLETAR: bucle de cerca de la primera dada >= x en la llista  
  
    // quan acaba el bucle, el PI està sobre el node amb dada>=x  
    // o està al final de la llista si tots els elements són <x  
    // COMPLETAR: inserir x en la llista  
  
    // el PI no es mou i prevPI és el node inserit  
}
```


Llista amb PI – Exercicis

- **Completa** en la classe **ListPIIntLinkedPlus** un mètode **triplicar()** que modifique una llista amb punt d'interès triplicant tots els seus elements. Aquest mètode ha d'implementar-se **fent ús exclusivament de les operacions** de la classe **ListPIIntLinked** (sense accedir a la seua representació interna).

```
public void triplicar() {  
    // COMPLETAR: situar el PI a l'inici  
  
    // COMPLETAR: bucle de recorregut de totes les dades de la llista  
  
    // COMPLETAR: inserir el valor del PI * 3  
  
    // el PI segueix en l'element recuperat  
    // i l'anterior, l'inserit, és el triple del recuperat  
    // COMPLETAR: eliminar el valor del PI  
  
    // l'element recuperat s'ha esborrat  
    // i el PI està en el següent element a canviar  
}
```

this.**begin()**;

2 5 7 1

PI

this.**insert**(3***get**()); this.**remove**();

1^a iteració

6 **2** 5 7 1

6 **5** 7 1

2^a iteració

6 15 **5** 7 1

6 15 **7** 1

3^a iteració

6 15 21 **7** 1

6 15 21 **1**

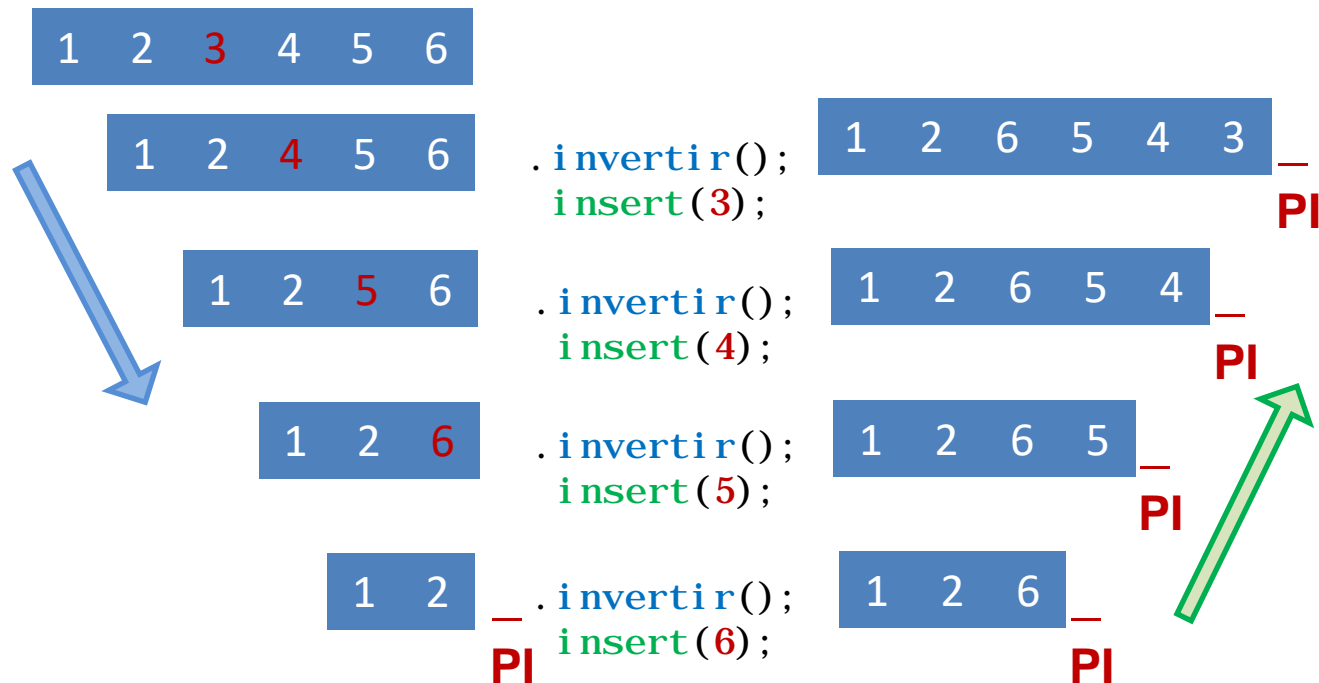
4^a iteració

6 15 21 3 **1**

6 15 21 3

Llista amb PI – Exercicis

- **Completa** en la classe *ListPIIntLinkedPlus* un mètode *invertir()* que, **recursivament**, invertisca els elements d'una llista amb punt d'interès **a partir del seu punt d'interès actual**, és a dir, que invertisca la subllista que es defineix des de l'element que ocupa el punt d'interès actual fins l'últim element de la llista. Aquest mètode ha d'implementar-se **fent ús exclusivament de les operacions** de la classe *ListPIIntLinked* (sense accedir a la seua representació interna).
- Per exemple, si la llista **1 2 [3] 4 5 6** té el PI en el tercer element, el resultat seria la llista **1 2 6 5 4 3 []** (amb el PI al final). És a dir, s'inverteix la subllista **[3] 4 5 6**.



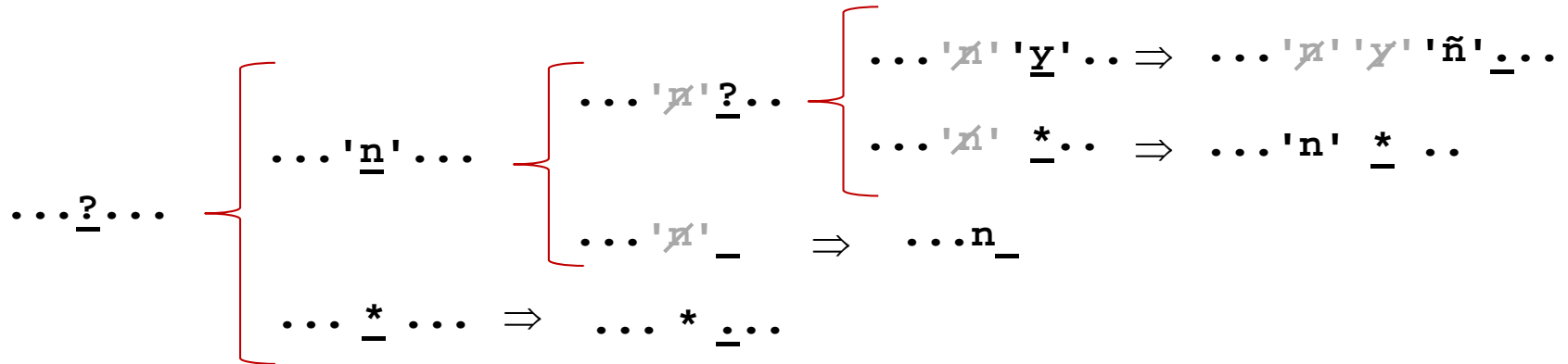
Llista amb PI – Exercicis

- La classe **CorregirChars** del paquet **usaLinear** crea una llista amb PI amb els caràcters d'un **string**, aprofitant que en Java un **char** és compatible amb el tipus **int**. I intercanvia tots els caràcters **ñ** per **ny** i viceversa.
- Completa** el mètode **corregirCV(ListPIIntLinked)** que, donada una llista amb PI de caràcters, canvie totes les ocurrences de **ñ** per **ny**.
- Per exemple, la llista **caña,anna,parañ** s'ha de canviar a **canya,anna,parany**
- Anàlisi de casos que es poden donar al recórrer la llista quan el PI està situat sobre un caràcter:

`'c''a'?'...` {
 `'c''a''ñ'... ⇒ 'c''a''ñ'...`
 `'c''a''n'...`
 `'c''a''n''y'...`
 `'c''a''l'... ⇒ 'c''a''l'...`

Llista amb PI – Exercicis

- **Completa** el mètode `corregirVC(ListPIIntLinked)` que, donada una llista amb PI de caràcters, canvie totes les ocurrences de *ny* per *ñ*.
- Per exemple: la llista `canya,anna,parany` s'ha de canviar a `caña,anna,parañ`
- Anàlisi de casos que es poden donar al recórrer la llista quan el PI està situat sobre un caràcter:



Llista amb PI – Exercicis

- En la classe **OpsLlistes** del paquet **usaLinear**, completa el mètode **intersectSort** que, donades dues llistes amb PI d'enters **l1** i **l2**, els elements de les quals estan **ordenats ascendentment**, obtinga una tercera llista amb els elements de la intersecció de les dues anteriors.
- Per exemple, si **l1** és 7 8 11 25 40 i **l2** és 1 3 7 11 40 89, aleshores la llista resultat serà 7 11 40.

```
/** Torna una nova llista amb els elements de la intersecció de l1 i l2.  
 * Precondicio: l1 i l2 han d'estar ordenades ascendentment.  
 */
```

```
public static ListPIIntLinked intersectSort(ListPIIntLinked l1, ListPIIntLinked l2)
```

Crear **l3** la llista amb PI resultat

Situar el PI a l'inici de **l1**

Situar el PI a l'inici de **l2**

Mentre el PI no estiga al final de **l1** i no estiga al final de **l2**:

Recuperar l'element del PI en **l1** (**e1**) i en **l2** (**e2**)

Si **e1** < **e2**, avançar en **l1**

Si no, si **e2** < **e1**, avançar en **l2**

Si no, vol dir que **e1** == **e2**

inserir **e1** en **l3**

avançar en **l1**

avançar en **l2**

Tornar **l3**

Llista amb PI – Exercicis

- En la classe **OpsLlistes** del paquet **usaLinear**, completa el mètode **intersection** que, donades dues llistes amb PI d'enters **l1** i **l2**, obtinga una tercera llista amb els elements de la intersecció de les dues anteriors.
- Per exemple, si **l1** és **11 7 8 40 25** i **l2** és **40 89 7 1 11 3**, aleshores la llista resultat serà **11 7 40**.

```
/** Torna una nova llista amb els elements de la intersecció de l1 i l2. */  
public static ListPIIntLinked intersection(ListPIIntLinked l1, ListPIIntLinked l2)
```

Crear **l3** la llista amb PI resultat

Situar el PI a l'inici de **l1**

Mentre el PI no estiga al final de **l1**:

Recuperar l'element del PI en **l1** (**e1**)

Buscar **e1** en **l2**

Si cerca amb èxit, inserir **e1** en **l3**

Avançar en **l1**

Tornar **l3**

Llista amb PI – Exercicis

- En la classe **OpsLlistes** del paquet **usaLinear**, completa el mètode **dif** que, donades dues llistes amb PI d'enters **l1** i **l2**, estant **l2** ordenada ascendentment, elimine de **l1** els elements que apareixen en **l2**.

Per exemple, si **l1** és 8 7 11 25 7 13 i **l2** és 1 3 7 11 13 17 19, aleshores **l1** ha de canviar a 8 25.

```
/** Elimina de l1 els elements que apareixen en l2.  
 * Precondicio: l2 ha d'estar ordenada ascendentment.  
 */  
public static void dif(ListPIIntLinked l1, ListPIIntLinked l2)
```

- Es poden revisar tots els elements de **l1** i buscar cadascun d'ells en **l2**. Si es troba, s'elimina (no hi ha que moure el PI per a passar al següent element), si no, s'avança el PI.
- La cerca de cada element de **l1** en **l2** es pot fer tenint en compte que **l2** està ordenada ascendentment.

Llista amb PI – Exercicis

Situar el PI a l'inici de **11**

Mentre el PI no estiga al final de **11**:

Buscar l'element del PI en **12** { Situar el PI a l'inici de **12**
Mentre el PI no estiga al final de **12** i elem. PI en **12** < elem. PI en **11**:
Avançar en **12**

Si cerca amb èxit perquè s'ha trobat l'element del PI en **12**:

eliminar l'element de **11**

Si no, avançar en **11**

- **Traça exemple** del bucle de revisió dels elements de **11** buscant-los en **12**:

	11	12	
inici:	<u>8</u> 7 11 25 7 13	1 3 7 11 13 17 19	
Després de la 1ª passada:	8 <u>7</u> 11 25 7 13	1 3 7 <u>11</u> 13 17 19	(no s'ha trobat el 8)
Després de la 2ª passada:	8 <u>11</u> 25 7 13	1 3 <u>7</u> 11 13 17 19	(s'ha trobat el 7)
Després de la 3ª passada:	8 <u>25</u> 7 13	1 3 7 <u>11</u> 13 17 19	(s'ha trobat el 11)
Després de la 4ª passada:	8 25 <u>7</u> 13	1 3 7 11 13 17 <u>19</u> _	(no s'ha trobat el 25)
Després de la 5ª passada:	8 25 <u>13</u>	1 3 <u>7</u> 11 13 17 19	(s'ha trobat el 7)
Després de la 6ª passada:	8 25 _	1 3 7 11 <u>13</u> 17 19	(s'ha trobat el 13)

Llista amb PI – Exercicis

- En la classe **OpsLlistes** del paquet **usaLinear**, completa el mètode **merge** que, donades dues llistes amb PI ordenades ascendentment, **l1** i **l2**, torne una llista que siga la mescla natural d'ambdues, és a dir, una llista que continga tots els elements de **l1** i **l2**, en ordre ascendent.

```
/** Torna la mescla natural de l1 i l2.  
 * Precondició: l1 i l2 han d'estar ordenades ascendentment.  
 */  
public static ListPIIntLinked merge(ListPIIntLinked l1, ListPIIntLinked l2)
```

- Es pot seguir una estratègia anàloga a la mescla natural o *merge* de dos arrays ordenats.
- Traça exemple** del primer bucle de selecció d'un element d'entre els de **l1** i **l2**:

	l1	l2	l3
inici:	<u>1</u> 4 5 7 8	<u>2</u> 3 6	_
Després de la 1ª passada:	1 <u>4</u> 5 7 8	<u>2</u> 3 6	1 _
Després de la 2ª passada:	1 <u>4</u> 5 7 8	<u>2</u> <u>3</u> 6	1 2 _
Després de la 3ª passada:	1 <u>4</u> 5 7 8	<u>2</u> <u>3</u> <u>6</u>	1 2 3 _
Després de la 4ª passada:	1 4 <u>5</u> 7 8	<u>2</u> <u>3</u> <u>6</u>	1 2 3 4 _
Després de la 5ª passada:	1 4 <u>5</u> <u>7</u> 8	<u>2</u> <u>3</u> <u>6</u>	1 2 3 4 5 _
Després de la 6ª passada:	1 4 5 <u>7</u> 8	<u>2</u> <u>3</u> <u>6</u> _	1 2 3 4 5 6 _

.... i amb un altre bucle acabar traslladant a la llista resultat els elements restants ...

Llista amb PI – Exercicis

- En la classe **OpsLlistes** del paquet **usaLinear**, completa el mètode **immersa** que, donades dues llistes amb PI d'enters **l1** i **l2**, determine si els elements de **l1** estan en **l2**, **en el mateix ordre encara que no necessàriament consecutius**. Per exemple:

- si **l1** és 6 2 6 i **l2** és 1 6 1 2 3 6 4, el mètode tornarà **true**,
- si **l1** és 6 5 6 i **l2** és 1 6 1 2 3 6 4, el mètode tornarà **false**,
- si **l1** està buida i **l2** és qualsevol llista amb PI, el mètode tornarà **true**.

El mètode ha de recórrer els elements de **l1** i **l2** com a molt una vegada. Es pot fer amb un bucle que, passada a passada, avança en **l2**, o en **l1** i **l2**.

- Traca exemple 1:**

	l1	l2
inici:	<u>6</u> 2 6	<u>1</u> 6 1 2 3 6 4
Després de la 1ª passada:	<u>6</u> 2 6	1 <u>6</u> 1 2 3 6 4
Després de la 2ª passada:	6 <u>2</u> 6	1 6 <u>1</u> 2 3 6 4
Després de la 3ª passada:	6 <u>2</u> 6	1 6 1 <u>2</u> 3 6 4
Després de la 4ª passada:	6 2 <u>6</u>	1 6 1 2 <u>3</u> 6 4
Després de la 5ª passada:	6 2 <u>6</u>	1 6 1 2 3 <u>6</u> 4
Després de la 6ª passada:	6 2 6 <u> </u>	1 6 1 2 3 6 <u>4</u>

torna **true**

Llista amb PI – Exercicis

- Traça exemple 2:

	11	12	cont1	cont2
inici:	<u>6</u> 5 6	<u>1</u> 6 1 2 3 6 4	3	7
Després de la 1ª passada:	<u>6</u> 5 6	1 <u>6</u> 1 2 3 6 4	3	6
Després de la 2ª passada:	6 <u>5</u> 6	1 6 <u>1</u> 2 3 6 4	2	5
Després de la 3ª passada:	6 <u>5</u> 6	1 6 1 <u>2</u> 3 6 4	2	4
Després de la 4ª passada:	6 <u>5</u> 6	1 6 1 2 <u>3</u> 6 4	2	3
Després de la 5ª passada:	6 <u>5</u> 6	1 6 1 2 3 <u>6</u> 4	2	2
Després de la 6ª passada:	6 <u>5</u> 6	1 6 1 2 3 6 <u>4</u>	2	1

torna **false**

Ací ja es detecta que el que queda de 11 no cap en el que queda de 12.

- Es poden usar les variables:
 - cont1**: número d'elements de 11 que falten per trobar en el mateix ordre consecutius en 12.
 - cont2**: número d'elements de 12 que queden per revisar.
- En el cas pitjor, tots els elements de 11 estan en 12, i 12 se recorre també completament (per exemple, si 11 és 6 1 6 i 12 és 1 6 1 2 3 4 6).
- El cost del mètode és $O(n^2)$ sent n la talla de 12.