UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escola Tècnica Superior d'Enginyeria Informàtica

etsinf

Unit 5. Linear data structures

Programming (PRG)

Academic Year 2016/2017

Departament de Sistemes Informàtics i Computació

Escola Tècnica Superior d'Enginyeria Informàtica

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escuela Técnica Superior de Ingeniería Informática

Contents

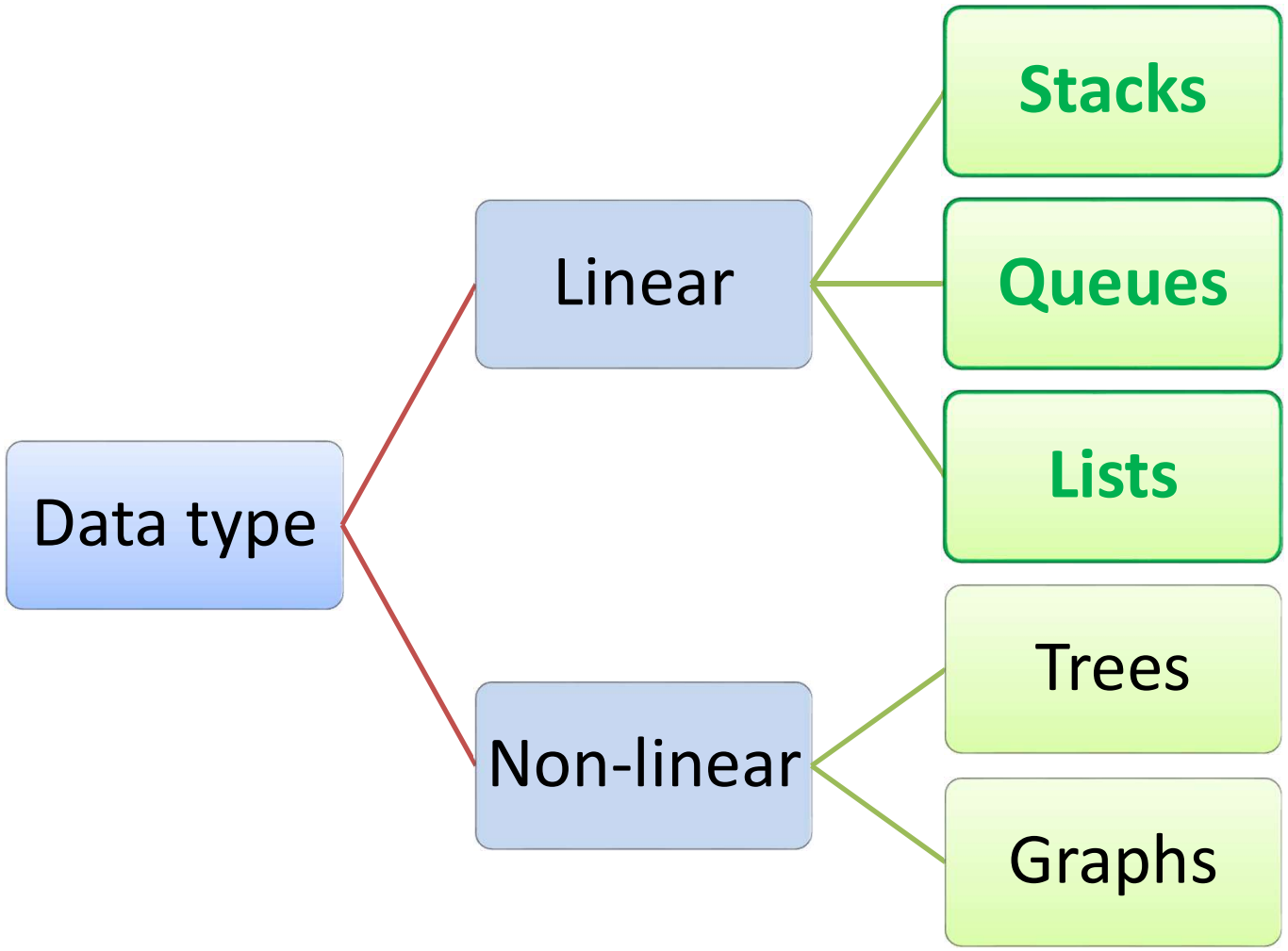
Introduction

Linear data types are those whose elements are formed by *linear sequences*

$$d_0 d_1 \ \ldots \ d_n, \quad n \geq 0$$

- where d_i are elements of the same data type,
- and over which some operations are possible, such as inserting or removing elements, consulting the data in a specific position, etc...
- According to the strategy behind the data manipulation, three linear data structures are presented in this chapter: *stacks*, *queues* and *lists*. The use of them is ideal in a wide variety of applications.

Introduction



Introduction

- The temporal behavior of the operations on a data type will depend on how these operations are implemented based on the representation or data structure chosen.

- *Data structure*: It is a data type with a specific representation of the data and the corresponding implementation of their operations.

- In this chapter, linear data structures will be presented based on two alternative ways for representing sequences of items:
  - With arrays: `StackIntArray`, `QueueIntArray`, `ListIPIntArray`.
  - Linked: `StackIntLinked`, `QueueIntLinked`, `ListIPIntLinked`.

Introduction

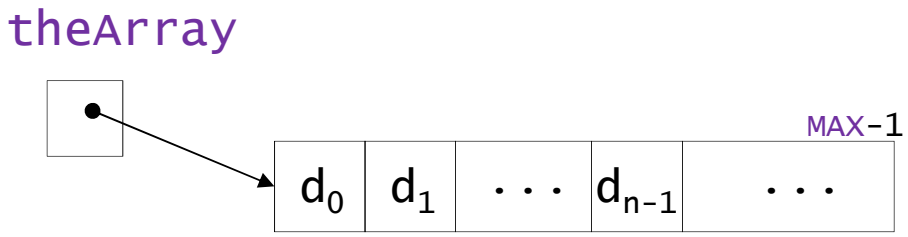
- This set of classes could be a first example of a user library, *linear*, organized as a Java package. All the classes must contain the following statement:

```
package linear;
```

  and include them all in the same folder '*linear*'.
- You can dedicate a project or folder, for example PRGlibraries, in order to have this and other packages of general interest. In Eclipse, a new package can be created by using the operation *File->New Package*.
- In order to install these libraries in the system, it is required to add the path PRGlibraries to the environment variable CLASSPATH.
- CLASSPATH is a variable that contains the path to the folders Java looks when looking for imported packages by a class when compiling or executing it.
- In Eclipse, the CLASSPATH variable can be modified with the option *Preferences>Java->ClassPath*.

Representation of sequences

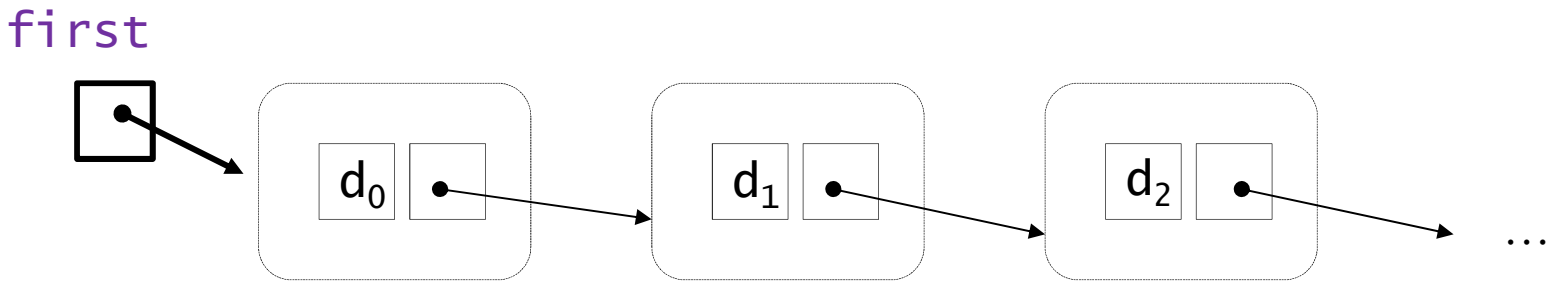
- Representation with arrays.
- An obvious way of representing a sequence consists of having its elements in successive positions in an array of a size large enough.



- This representation allows accessing to the elements of the sequence in a constant temporal cost.
- The size of the sequence is limited to the `MAX` length of the array.
- The insertion/deletion of any item at the i-th position of the sequence requires reorganizing `theArray[i..n-1]`.

Representation of sequences

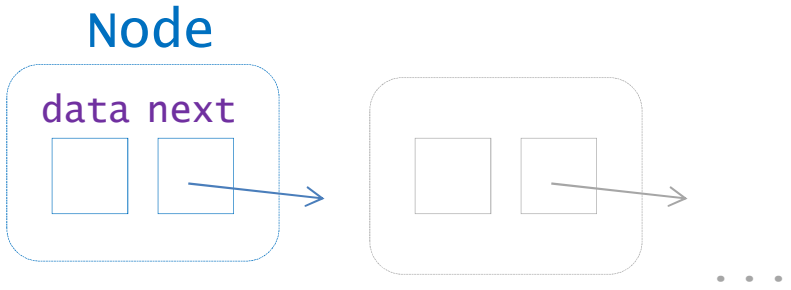
- Linked representation.
- A representation of a sequence in which new memory is allocated as new elements are inserted in the sequence.
- Every item has a link (or reference) to the position where the next item in the sequence is in the memory.



- Accessing to the elements of the sequence is no longer a direct access by using its position in the sequence.

Linked representation of sequences

- *Node:* Structure that contains an element and a link to the next one




```
/**
 *  NodeInt class: Node which
 *  data is an int
 */
class NodeInt {

   int  data;
   NodeInt next;


   ...
}
```

"friendly" class to be included in the same package of `Stack`, `Queue` and `List`.

Linked representation of sequences

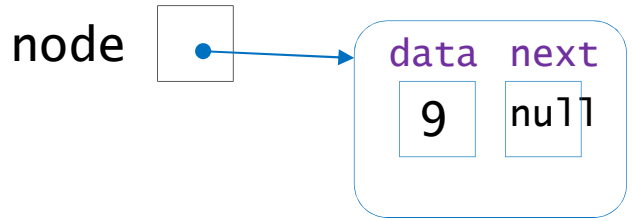
Two constructors are going to be considered in the class:

- Constructor A: Creates a node with data d that has no next.

```
class NodeInt{

   int  data;
   NodeInt next;

   /** Constructor A */
   NodeInt(int d) {
      data = d;
      next = null;
   }
    ...
}
```

Example:

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

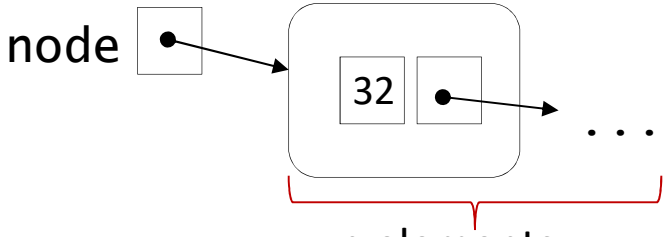


Linked representation of sequences

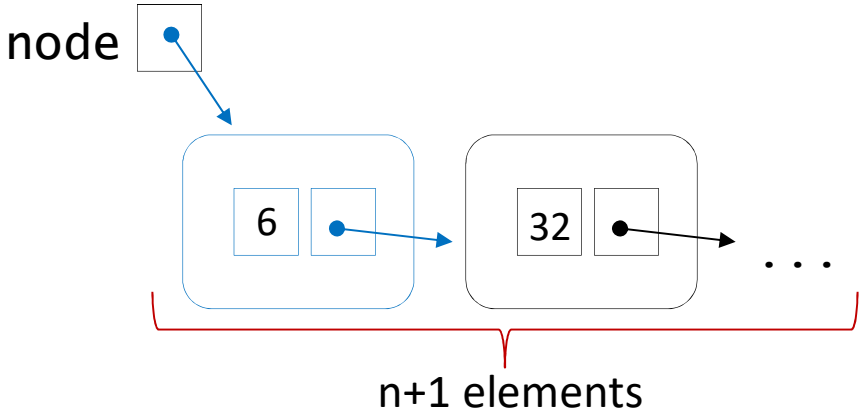
- Constructor B: Creates a node with data d linked with a pre-existing node.

```
class NodeInt {

    int  data;
    NodeInt next;

    /** Constructor A */
    NodeInt(int d) {
        data= d;
        next= null;
    }

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

Example:




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



Linked representation of sequences

- **Traversal** of arrays (linked lists), by running an operation with each item.

```
// Traversal of an array
int i = 0;
while (i<a.length) {
   operation(a[i]);
   i++;
}
```

```
// Traversal of a linked list
NodeInt aux = sec;
while (aux!=null) {
   operation(aux.data);
   aux = aux.next;
}
```

Linked representation of sequences

- **Search** in arrays or linked sequences of an element which fulfils the `property`.

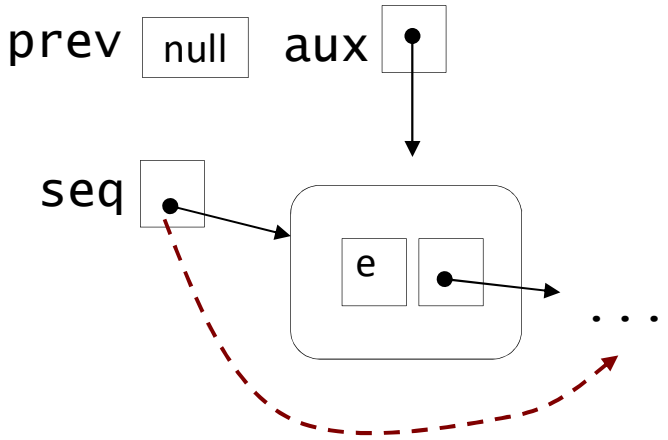
```
// search in an array
int i = 0;
while (i<a.length && !property(a[i]))
   i++;
if (i<a.length) return i;
else return -1;
```

```
// search in a linked sequence
NodeInt aux = sec;
while( aux!=null && !propeerty(aux.data) )
   aux = aux.next;
return aux;
```

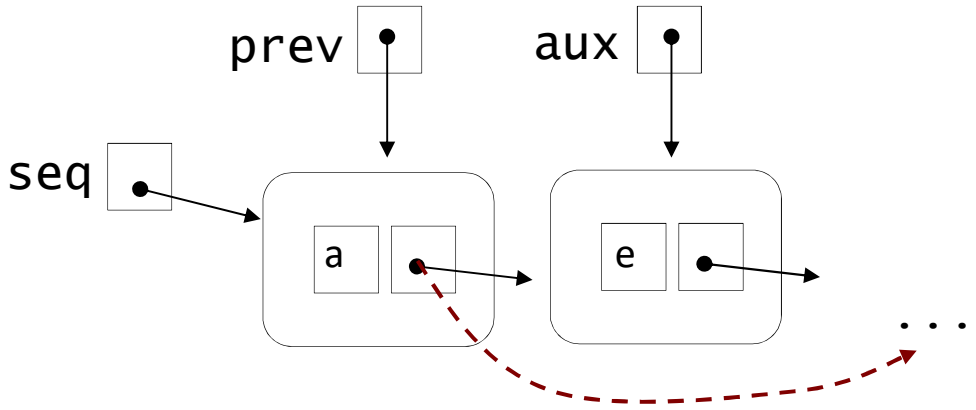
Linked representation of sequences

- **Deleting** a item, e, of the sequence.

```
NodeInt aux = seq, prev = null;
while (aux!=null && aux.data!=e) {
   prev = aux;
   aux = aux.next;
}
if (aux!=null) // Successful search
   if (prev!=null) prev.next = aux.next;
   else seq = aux.next;
```



(1) aux was the fist item, has no previous

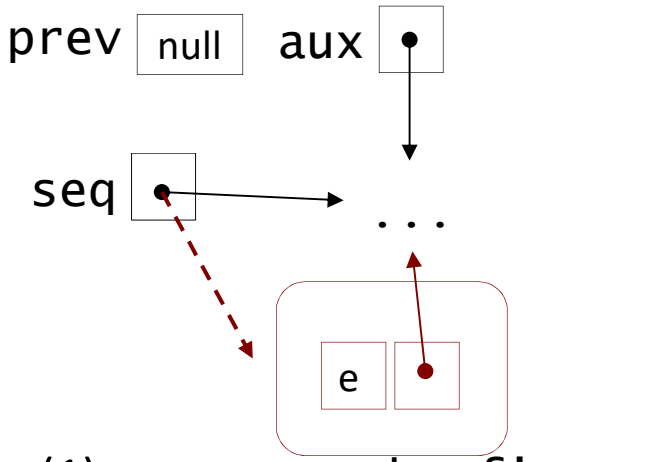


(2) aux is not the fist item, has previous

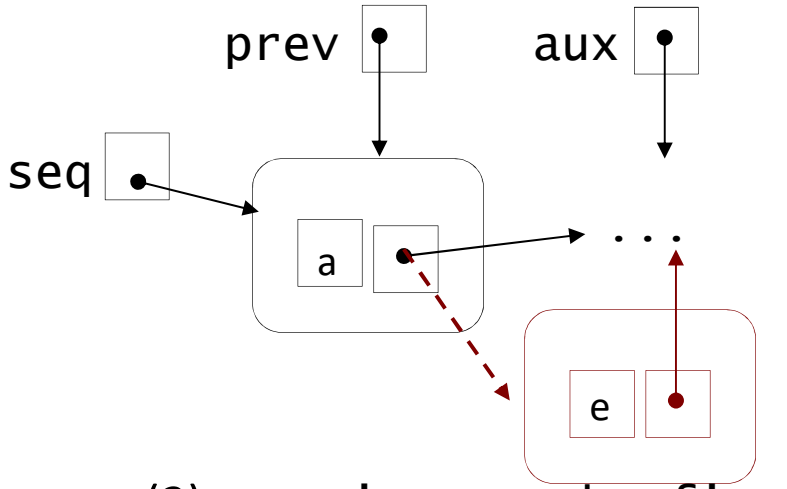
Linked representation of sequences

- **Inserting** a new item, e, into a sorted sequence.

```
NodeInt aux = seq, prev = null;
while( aux!=null && aux.data<e ) {
    prev = aux;
    aux = aux.next;
}
if (prev!=null) ant.next = new NodeInt(e,aux);
    else seq = new NodeInt(e,aux);
```



(1) aux was the fist item, has no previous



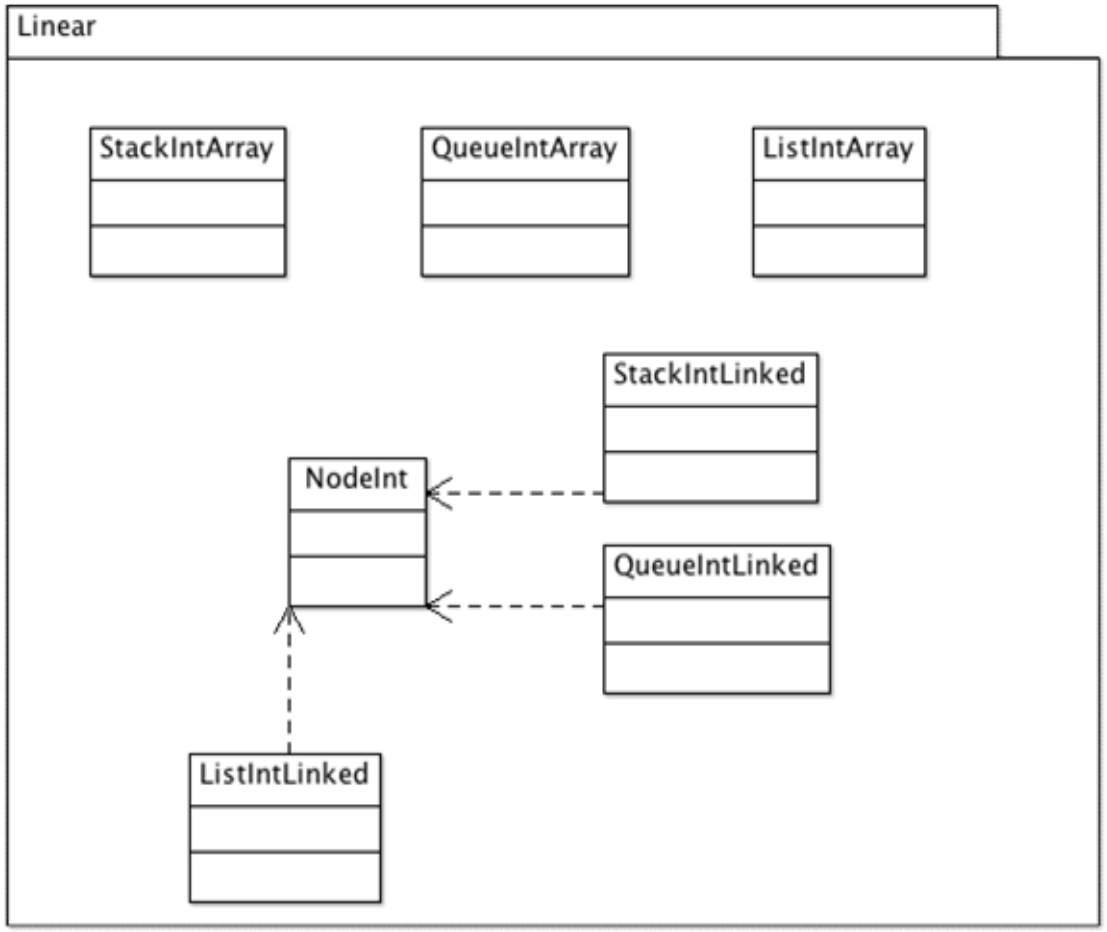
(2) aux is not the fist item, has previous

Linked representation of sequences

- The linked implementation of `Stack`, `Queue` and `List` whose data are of the type integer will be based on the class `NodeInt`.
- All these classes will be in the same `linear` package.
- The class `NodeInt` and its components, attributes and methods, will be *friendly*: they will be accessible to the code of the classes of the package, and *`private`* for the rest.

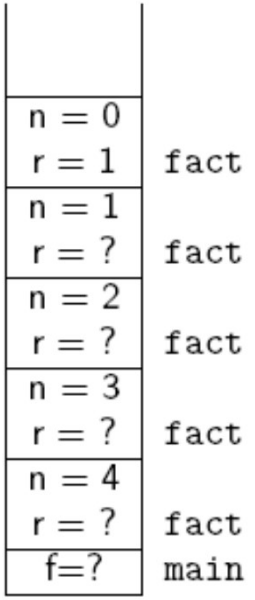
- The package `linear` is completed with the implementation of `Stack`, `Queue` and `List` of `int`, based on arrays.

Linked representation of sequences



Stacks

- A *stack* is a sequence where the access to the first element is performed following a standard *LIFO* strategy (*L*ast *I*n *F*irst *O*ut).
  - The elements of a stack will always be removed in the reverse order of the order followed in their insertion, so that the last inserted element will be the very first to be removed, and the first element inserted will be the last one.
- Example of stacks:



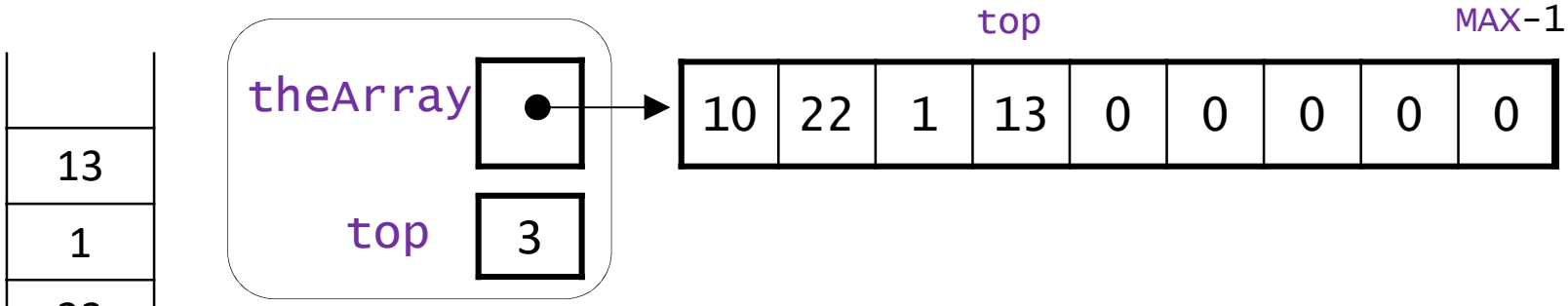
Operations on Stacks

Operation	Description
Stack: $\rightarrow$ Stack	Creates an empty stack.
push: Stack element $\rightarrow$ Stack	Inserts an element.
pop: Stack $\rightarrow$ element	Consults and extracts an element, only if the stack is not empty.
top: Stack $\rightarrow$ element	Consults the top, only if the stack is not empty.
isEmpty: Stack $\rightarrow$ boolean	Decides whether the stacks is empty.
size: Stack $\rightarrow$ int	Consults the number of elements.

It is possible to implement these operations with a constant temporal cost.

Stacks– Implementation with arrays

- A stack can be implemented using an array (`theArray`) that stores the data, and index (`top`) that marks the access point to the stack and a constant the defines the maximum size of the array (`MAX`). A stack of elements of the type `int` will be represented:




```
public class StackIntArray {
     private int[] theArray;
     private int top;
     private static final int MAX = ...;

     //Implementation of the operations:
     ...
}
```

Stacks– Implementation with arrays

- Constructor `StackIntArray`: Creates the array and initializes the access point to -1, which means that the stack is empty.

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

- Operation `push`:

```
public void push(int x) {
    if (top+1==MAX)
       System.out.println("No space for more elements");
    else {
            theArray[++top]=x;
         }
}
```

Stacks– Implementation with arrays

- Operations pop and top: Only when the stack is not empty.

```
public int pop() {
    return theArray[top--];
}
```

```
public int top () { return theArray[top]; }
```

- Operation isEmpty:

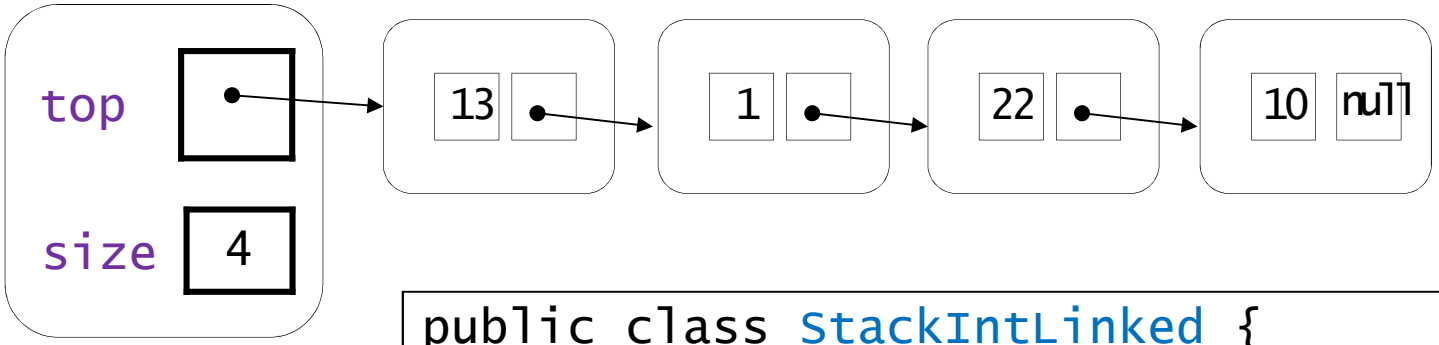
```
public boolean isEmpty() { return top==-1; }
```

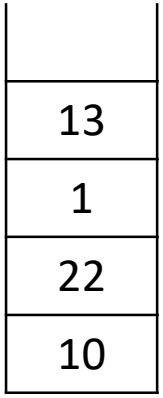
- Operation size:

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

Stacks– Linked implementation

- It's a linked implementation of a stack. The top is the first element of it (its reference) and it is represented as an attribute called `top`. Besides, an attribute `size` is added that represents the number of elements of the stack.






```
public class StackIntLinked {
     private NodeInt top;
     private int size;

 // Implementation of the operations:
     ...

}
```

Stacks– Linked implementation

- Constructor `StackIntLinked`: Assigns `null` to the `top` reference and initializes `size` to 0.

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

- Operation `push`:

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

Stacks– Linked implementation

- Operations `pop` and `top`: Only if the stack is not empty (either `top!=null` or `size!=0`).

```
public int pop() {
     int x = this.top.data;
     this.top = this.top.next;
     this.size--;
     return x;
}

public int top () {
    return this.top.data;
}
```

Stacks– Linked implementation

- Operation `isEmpty`:

```
public boolean isEmpty () {
    return (this.top==null);
    // or return (this.size==0);
}
```

- Operation `size`:

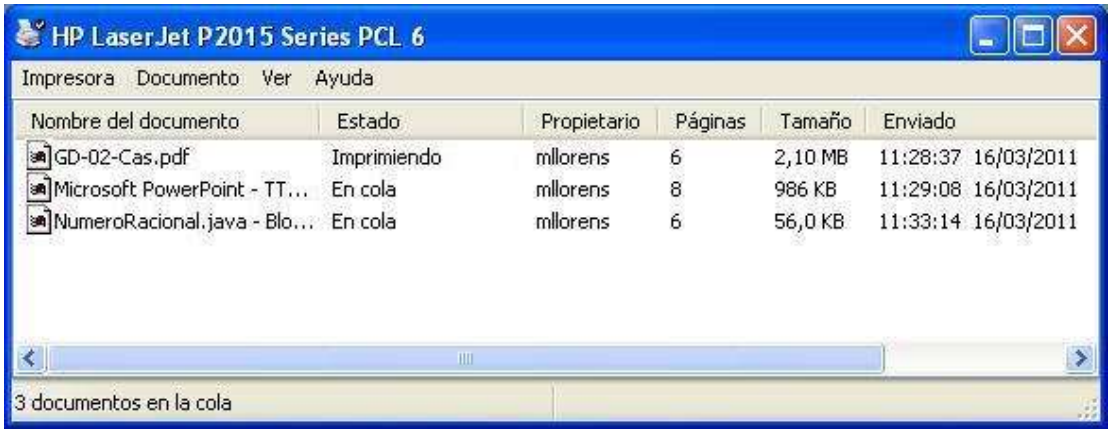
```
public int size() {
    return this.size;
}
```

Stacks– Implementation comparison

- The temporal complexity of all the operations in the two studied representations is independent of the size of the problem: $\Theta(1)$.
- In terms of spatial complexity, the implementation with arrays has the disadvantage of the estimation of the maximum size of the array, allocating space that in many cases will be wasted.
- On the other hand, the linked representation requires an additional memory space for the links.
- This additional memory consumption will not have much importance if the type of the components of `theArray` is relatively small, as is the case of a stack of `int`, or stacks of objects (the components of the array are then references).

Queues

- A *queue* is a collection of data of the same type in which the access is made following a FIFO (*F*irst *I*n *F*irst *O*ut) strategy.
  - The first inserted element (the first which arrives to the queue) is the first to be served (to be removed from the queue).
- Examples of queues:



Operations on Queues

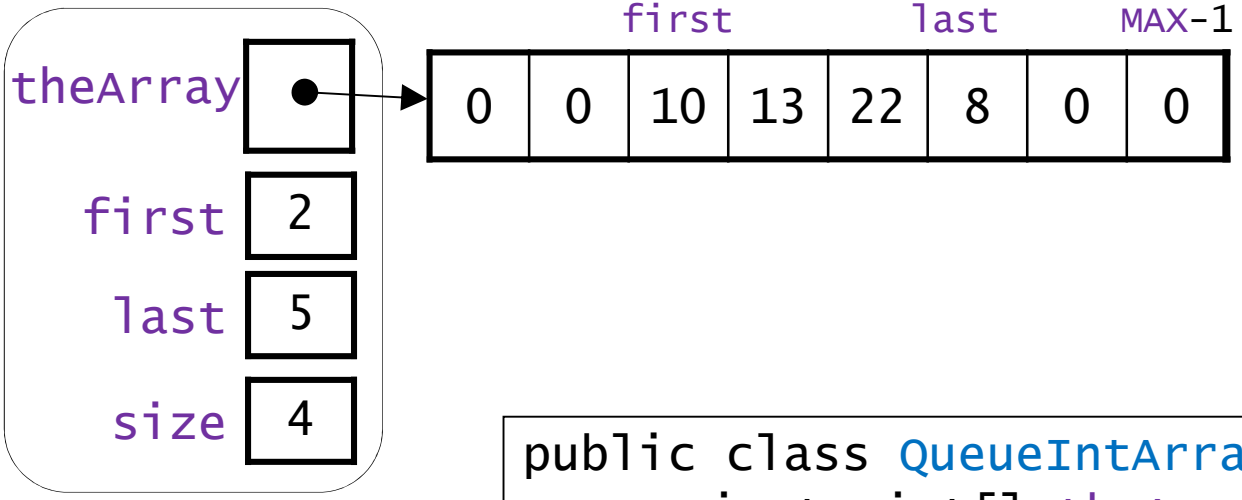
Operation	Description
Queue: $\rightarrow$ Queue	Creates an empty queue.
enqueue: Queue element $\rightarrow$ Queue	Inserts an element at the end.
dequeue: Queue $\rightarrow$ element	Consults and removes the first element, only if the queue is not empty.
first: Queue $\rightarrow$ element	Consults the first element of the head, only if the queue is not empty.
isEmpty: Queue $\rightarrow$ boolean	Returns whether the queue is empty or not.
size: Queue$\rightarrow$ int	Consults the number of elements.

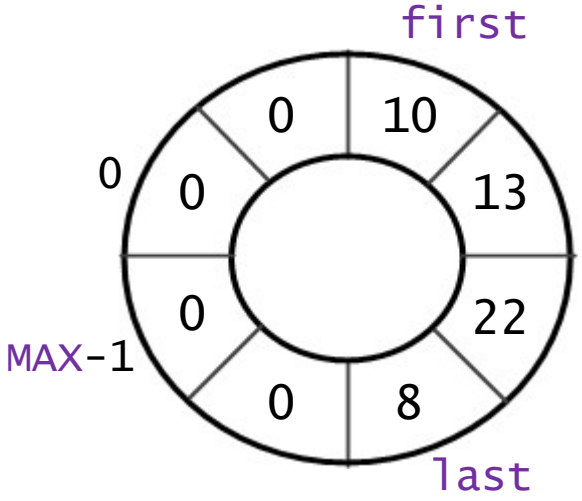
Queues– Implementation with arrays

- A queue can be implemented using an array (`theArray`) that stores the data, two indices (`first` and `last`) to mark the two access points to the queue and a constant that defines the maximum size of the array (`MAX`).
- In order to be able to reuse all the array components without having to move elements and being able to occupy all its positions, the array is going to be considered as a *circular structure*, without beginning or end, where after the last item comes the first one.
- Furthermore, in order to distinguish the situations of an empty queue and a queue with its maximum number of elements (`MAX`), an attribute (`size`) is added to count the number of elements of the queue.

Queues– Implementation with arrays

- Queue which data is of the type `int`:






```
public class QueueIntArray {
     private int[] theArray;
     private int first, last, size;
     private static final int MAX = ...;

     // Implementation of the operations:
      ...
}
```

Queues– Implementation with arrays

- Constructor `QueueIntArray`: Create the array and initializes the access points, indicating that the queue is empty.

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

- Operation `enqueue`:

```
public void enqueue(int x) {
    if (size==MAX)
       System.out.println("No more elements");
    else {
            last= increment(last);
            theArray[last]=x;
            size++;
         }
}
```

Queues– Implementation with arrays

- Operations `dequeue` and `first`: Only when the queue is not empty.

```
public int dequeue() {
        int x = theArray[first];
        first= increment(first);
        size--;
        return x;
}

public int first() {
        return theArray[first];
}
```

Queues– Implementation with arrays

- Operation `isEmpty`:

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

- Operation `size`:

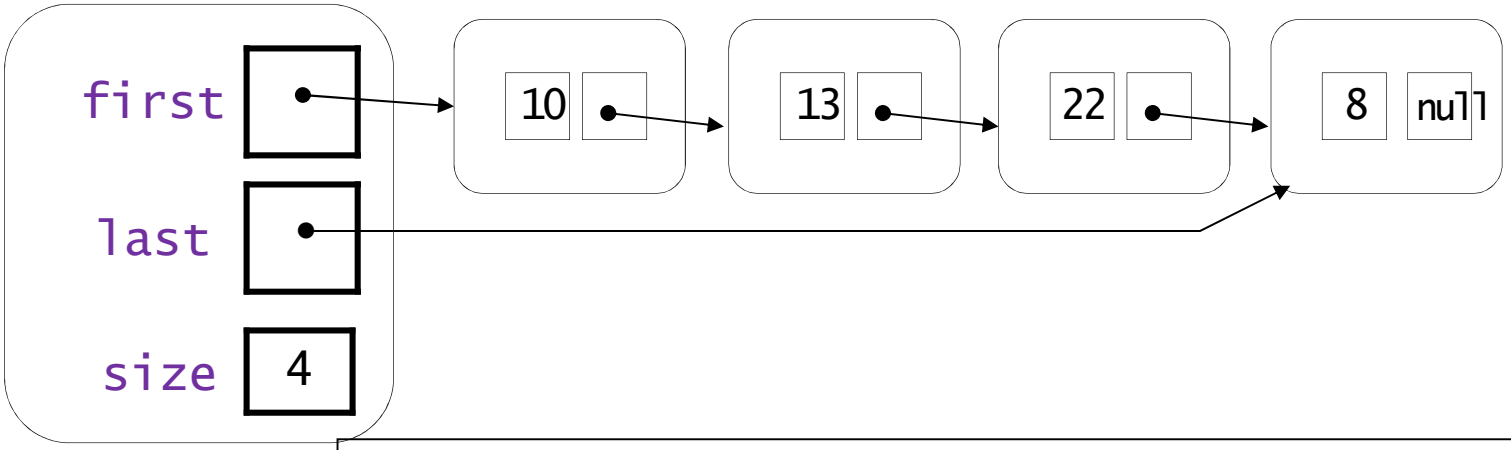
```
public int size() { return size; }
```

- Private operation `increment`: Implements the circular behavior, `MAX` is equals to the size of the array.

```
private int increment (int i) {
    return (i + 1) % MAX;
}
```

Queues– Linked implementation

- It's a linked implementation, the queue will have two attributes to maintain the references to the head and tail of the queue.




```
public class QueueIntLinked {
     private NodeInt first, last;
     private int size;

    // Implementation of the operations:
     ...
}
```

Queues– Linked implementation

- Constructor `QueueIntLinked`: Assigns `null` to the references `first` and `last` and initializes `size` to 0.

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

- Operation `enqueue`:

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

Queues– Linked implementation

- Operations `dequeue` and `first`: only when the queue is not empty (either `first!=null` or `size!=0`).

```
public int dequeue () {
     int x = first.data;
     first = first.next;
     if (first==null) last = null;
     size--;
     return x;
}

public int first () { return first.data; }
```

Queues– Linked implementation

- Operation `isEmpty`:

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

- Operation `size`:

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

Queues– Comparison of implementations

- The temporal complexity of all the operations studied is independent of the size of the problem: $\Theta(1)$.
- In terms of spatial complexity, the implementation with arrays has the disadvantage of the estimation of the maximum size of the array, allocating space that in many cases will be wasted.
- Also in this case, the linked representation requires an additional memory of the links. But just like in the stacks, this additional space will not be important if the components of `theArray` are relatively small.

Lists with interest points

- A *list* is a sequence in which the access can be made in any point.
- An *interest-point list* is a list which maintains a currently active item. This item is known as the interest point or cursor.
- Example of lists:

Profesores de la asignatura (2010/2011)

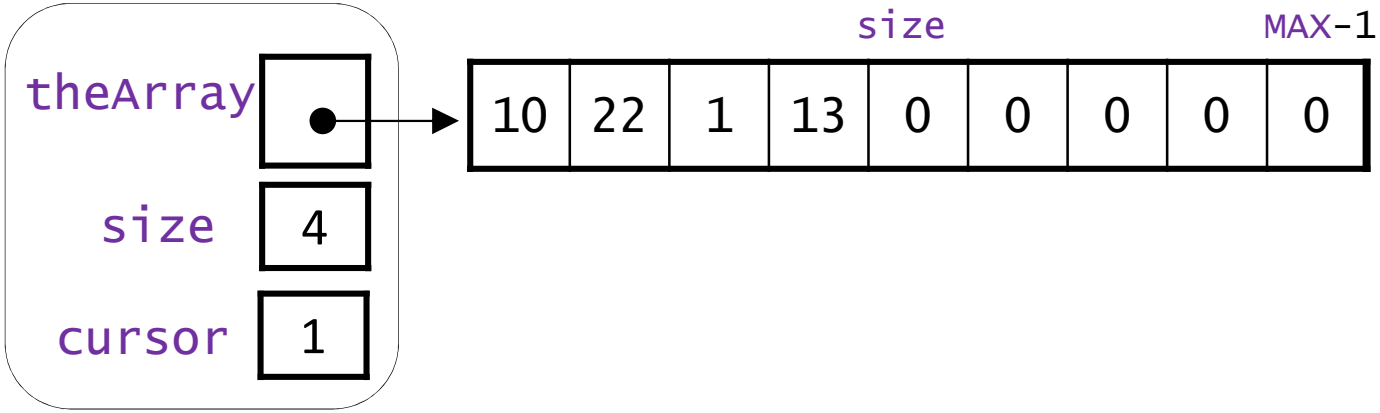
Nombre
Casanova Faus, Assumpció
Gómez Adrian, Jon Ander
González Mollá, Jorge
Herrero Cuco, Carlos
Llorens Agost, María Luisa
Marqués Hernández, Francisco
Piris Ruano, Francisco Javier
Ramos Peinado, Enrique
Torres Goterris, Francisco
Belenguer Faguas, Jorge
Martínez Hinarejos, Carlos David
Toselli ., Alejandro Héctor

Operations on Lists

Operations	Description
public List()	Creates an empty list
public boolean isEmpty()	Decides whether the list is empty or not
public int search(int x)	Looks for the position of a given element **x** Returns -1 if it is not in the list
public int size()	Returns the number of elements
public void begin()	Sets the cursor on the first item
public void end()	Sets the cursor on the last item
public void next()	Moves the cursor to the next item
public void previous()	Moves the cursor to the previous item
public int get()	Returns the value of the currently active item
public void remove()	Deletes the currently active item and changes the cursor to the next item in the list
public void insert(int x)	Inserts a new value in the list where the cursor is positioned

Lists– Implementation with arrays

- The elements of the list are in the first consecutive positions of the array. The components of the array are of the type `int`:




```
public class ListIntArray {
     private int[] theArray;
     private int size, cursor;
     private static final int MAX = ...;

    //  Implementation of the operations:
    ........
}
```

Lists– Implementation with arrays

- Constructor `ListIntArray`: Creates an empty list.

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

- Consulting operation `isEmpty`:

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

- Consulting operation `size`:

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

- Checking if the cursor is at the end:

```
public boolean isAtTheEnd() { cursor>=size; }
```

Lists– Implementation with arrays

- Sets the cursor at the beginning of the list:

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

- Moves the cursor to the next item in the list. The cursor doesn't must be at the end:

```
public void next() {
   cursor++;
}
```

- Returns the value of the currently active item. The cursor doesn't must be at the end:

```
public int get() {
   return theArray[cursor];
}
```

Lists – Implementation with arrays

- Operation `search`: Looks for the position of x in the list. If it is not in the list, returns -1.

```
public int search( int x )
{
    cursor = -1;
    return searchNext( x );
}

public int searchNext( int x )
{
    cursor++;
    while( cursor < size && theArray[cursor] != x )
       cursor++;

    return ( cursor < size ) ? cursor : -1;
}
```

Lists – Implementation with arrays

- Operation insert: Inserts the element x in the cursor position. Elements from cursor to the end must be shifted one position to the right.

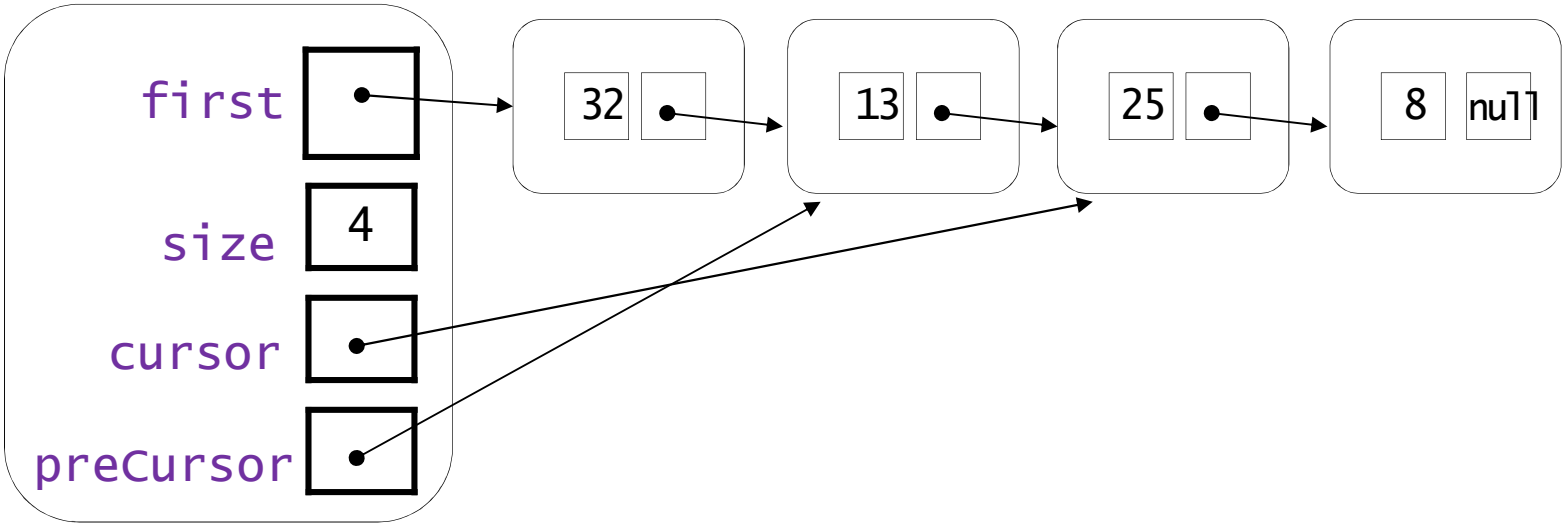
```
public void insert( int x )
{
    if (size==MAX)
       System.out.println( “No more elements” );
    else {
       for( int k = size-1; k>=cursor; k--)
              theArray[k+1] = theArray[k];
       theArray[cursor] = x;
       size++;
    }
}
```

Lists – Implementation with arrays

- Operation remove: Returns and removes the element at the cursor position.

```
public int remove()
{
    int aux = theArray[cursor];
    for( int k = cursor+1; k < size; k++ )
        theArray[k-1] = theArray[k];

    size--;
    return aux;
}
```

Lists– Linked implementation




```
public class ListIntLinked {
     private NodeInt first, cursor, preCursor;
     private int size;

    //  Implementing the operations:
    ........
}
```

Lists– Linked implementation

- Constructor `ListIPIntLinked`: Creates an empty list.

```
public ListIPIntLinked() {
    first = preCursor = cursor =
    null;
    size = 0;
}
```

- Consulting operation `isEmpty`:

```
public boolean isEmpty () {
    return size==0;
  // Alternativelly: return first==null;
}
```

- Consulting operation `size`:

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

- Checking if the cursor is at the end:

```
public boolean isAtTheEnd() { return cursor == null; }
```

Lists– Linked implementation

- Sets the cursor at the beginning of the list:

```
public void begin() {
    cursor = first;
    preCursor = null;
}
```

- Moves the cursor to the next item in the list. The cursor should not be at the end:

```
public void next() {
    preCursor = cursor;
    cursor = cursor.next;
}
```

- Returns the value of the currently active item. The cursor should not be at the end:

```
public int get() {
    return cursor.data;
}
```

Lists– Linked implementation

- Operation `search`: Checks whether **x** is in the list. If it existed cursor is positioned on it.

```
public int search( int x )
{
  int pos=0;
  NodeInt aux = first, preAux = null;

  while( aux != null && aux.data != x ) {
       preAux = aux;
       aux = aux.next;
       pos++;
  }
  if ( aux != null ) {
       cursor = aux;
       preCursor = preAux;
       return pos;
  } else
       return -1;
}
```

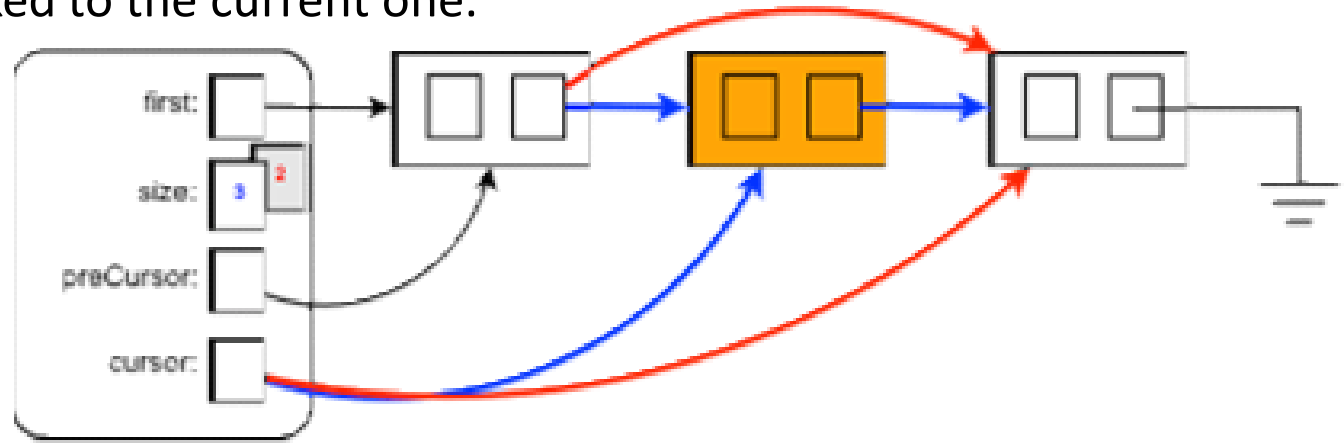
Lists– Linked implementation

- Operation `remove`: Returns and removes the item in the list at the cursor position.

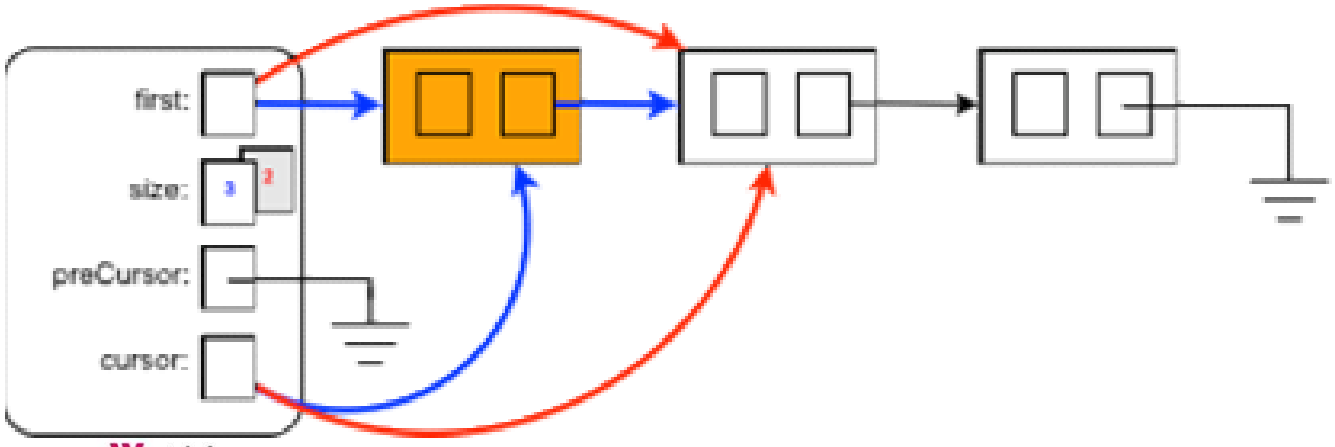
```
public int remove()
{
  NodeInt current = cursor;
  cursor = cursor->next;

  if ( preCursor != null ) {
       preCursor.next = cursor;
  } else if ( first == current ) {
       first = cursor;
  }
  size--;

  return current->data;
}
// Removing the last item in the list implies that
cursor position becomes not valid.
```

Lists – Linked implementation

- Operation remove: Returns and removes the currently active item.
  - To remove a node after the first position, previous node must be linked to the current one.



  - The first node has no predecessor, its removal affects to the `first`.



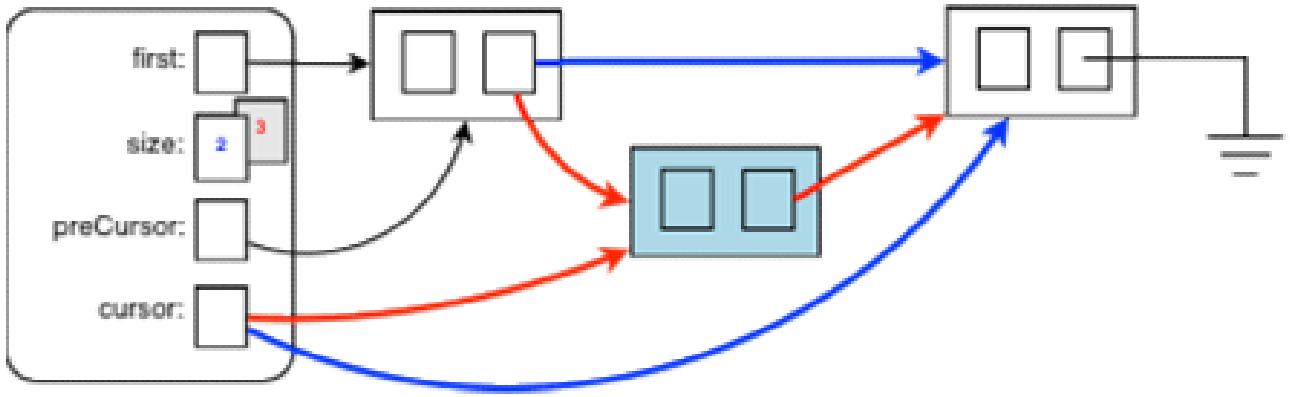
Lists– Linked implementation

- Operation `insert`: Inserts **x** in the list at the cursor position.

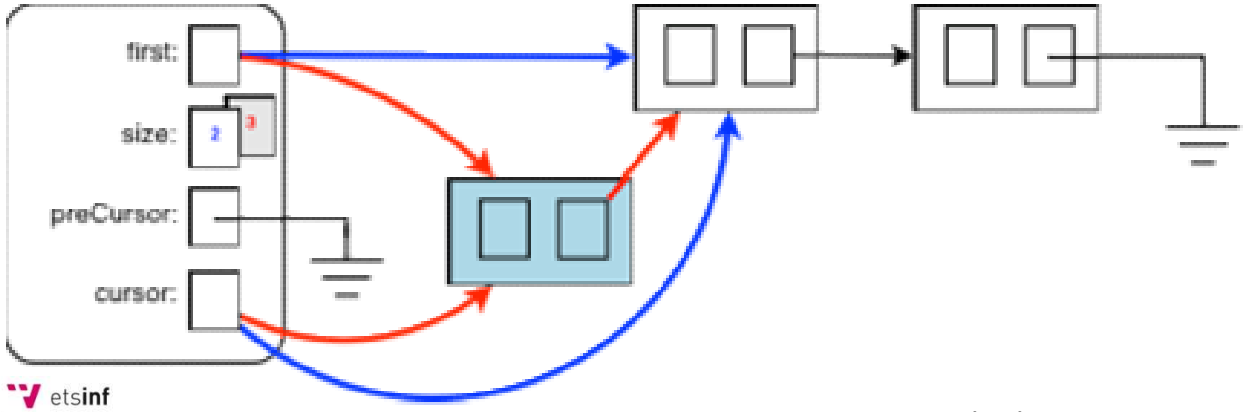
```
public void insert( int x )
{
  NodeInt ni = new NodeInt( x, cursor );

  if ( cursor == null ) {
       first = cursor = ni;
       preCursor = null;
  } else if ( cursor == first ) {
       first = ni;
       preCursor = first;
  } else {
       preCursor.next = ni;
       preCursor = ni;
  }
  size++;
}
```

Lists – Linked implementation

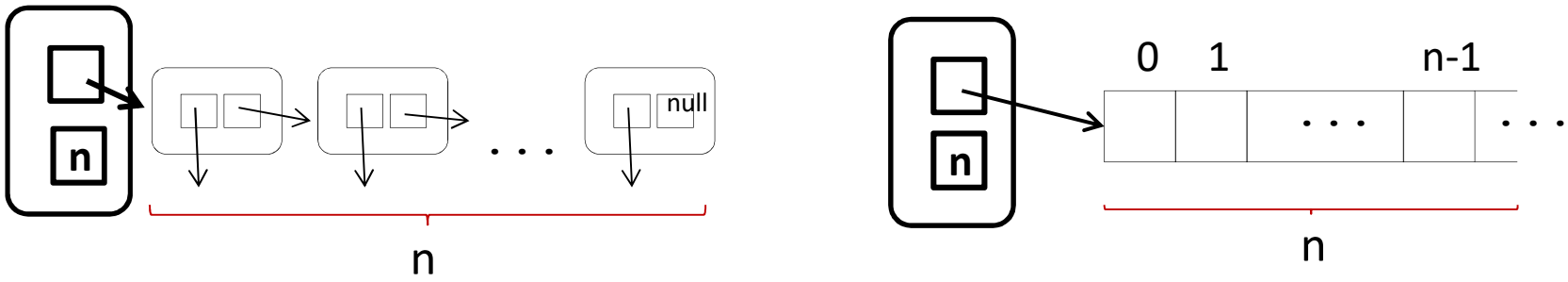
- Operation insert: Inserts a new item at the cursor position.
  - To add a new item after the first position the cursor position must be properly linked using the preCursor instance variable.



  - The first node has no predecessor, inserting a new one affects to the `first`.



Lists– Implementations comparison



Operation	Linked representation	Array representation
isEmpty()	$\Theta(1)$	$\Theta(1)$
size()	$\Theta(1)$	$\Theta(1)$
search(int x)	$\Omega(1)$, $O(n)$	$\Omega(1)$, $O(n)$
get()	$\Theta(1)$	$\Theta(1)$
remove()	$\Theta(1)$	$\Theta(n)$
insert(int x)	$\Theta(1)$	$\Theta(n)$