Estructuras de datos fundamentales

Unidad 3

Renato Mazzanti

Objetivos

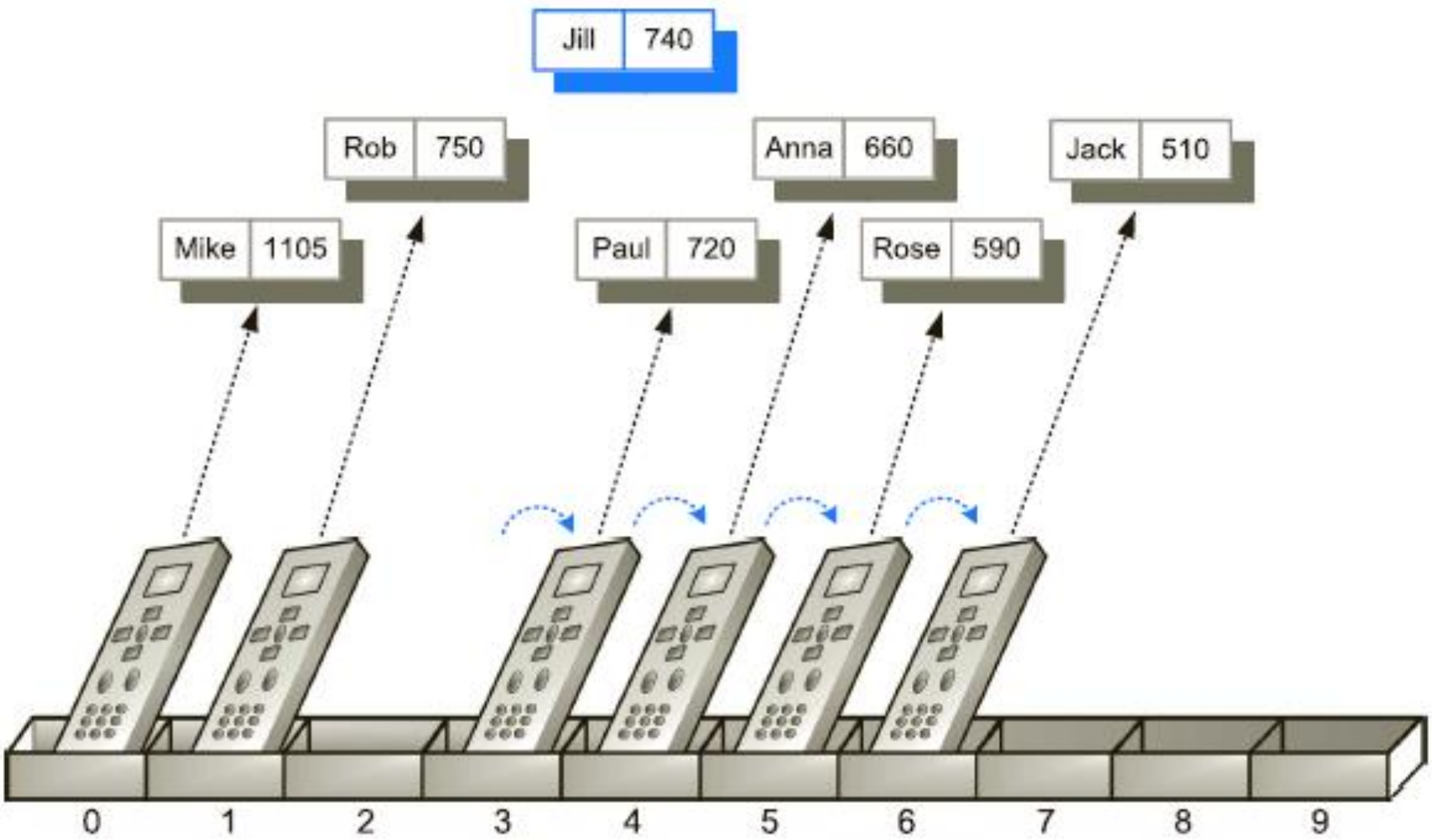
- Arreglos
- Listas simplemente enlazadas
- Listas circulares
- Listas doblemente enlazadas
- Testing de arreglos vs listas
- Clonación de estructuras de datos

Usando Arreglos

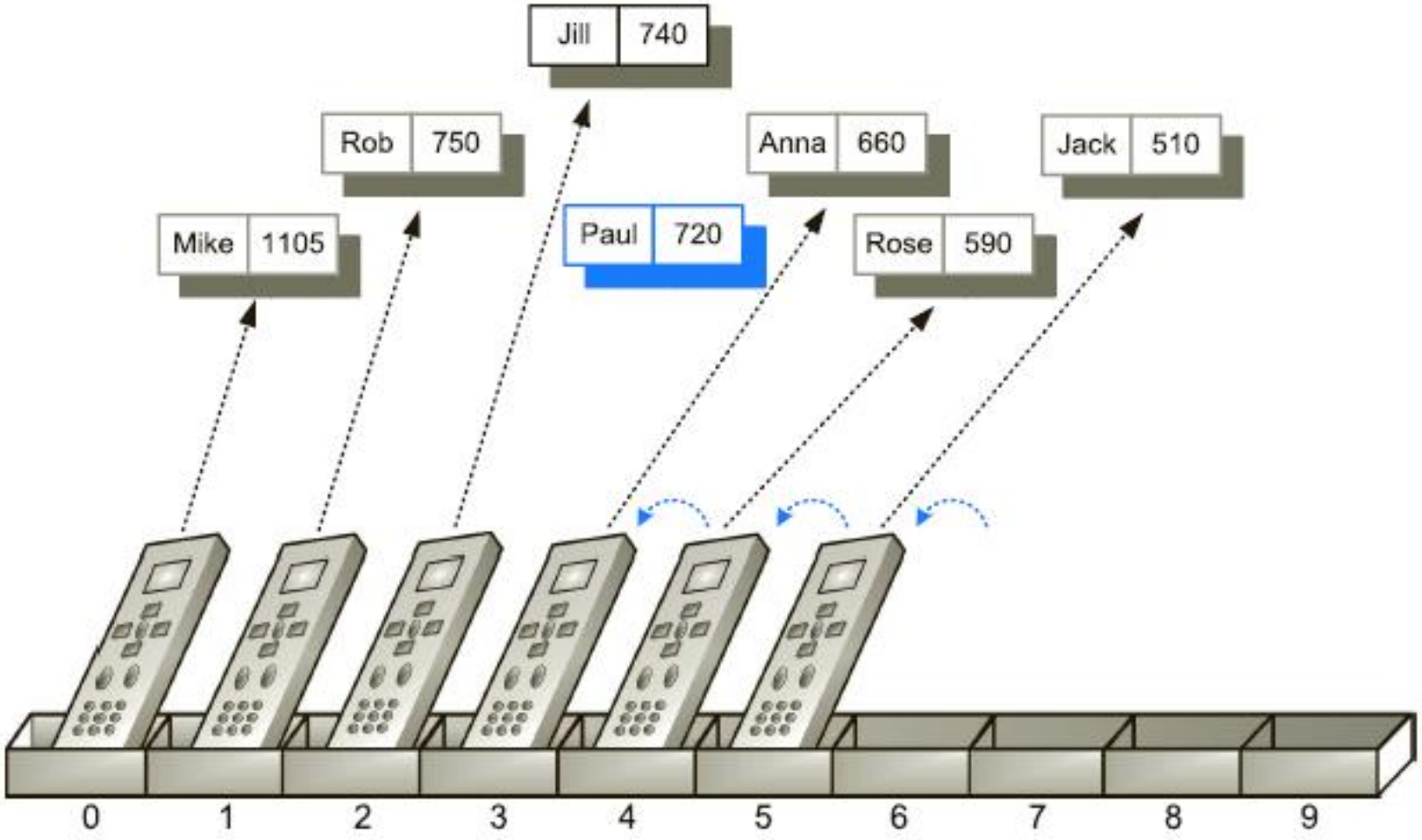
- La primera aplicación que estudiamos es almacenar una secuencia de entradas de puntuación para un videojuego en un arreglo.
- Esto es representativo de muchas aplicaciones en las que se debe almacenar una secuencia de objetos. (ej. las historias clínicas de los pacientes en un hospital o los nombres de jugadores en un equipo de fútbol).
- Un componente a incluir es un entero que representa representa el puntaje de la partida o juego.
- Otra cosa útil de incluir es el nombre de la persona que gana este puntaje.
- Podríamos continuar desde aquí, añadiendo campos que representan la fecha en que se ganó la partida o juego y estadísticas que llevaron a esa puntuación, etc.
- ***Formas de declararción de arreglos:***
  - » **new** elementType[length]

$$elementType[\,]\ arrayName = \{initialValue_0, initialValue_1, \ldots, initialValue_{N-1}\};$$

Agregar

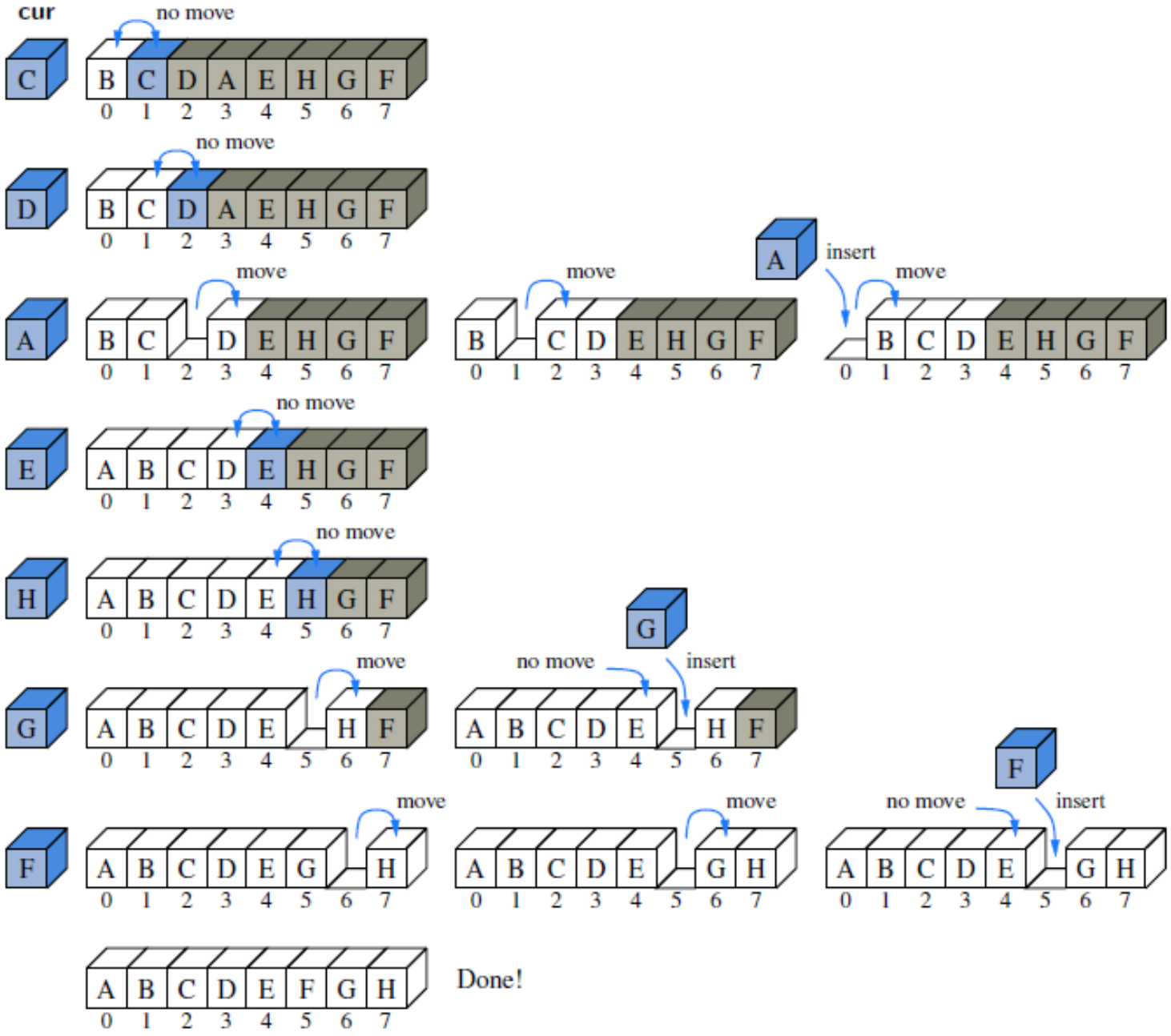

Jill 740
Rob 750
Anna 660
Jack 510
Mike 1105
Paul 720
Rose 590
0
1
2
3
4
5
6
7
8
9

Extraer


Jill 740
Rob 750
Anna 660
Jack 510
Mike 1105
Paul 720
Rose 590
0
1
2
3
4
5
6
7
8
9

Ordenar un arreglo

Algorithm InsertionSort(*A*):
Input: Un arreglo *A* de *n* elementos comparables
Output: El arreglo *A* con los elementos ordenados en orden creciente
for *k* from 1 to *n*−1 **do**
Insert *A*[*k*] a este en el lugar apropiado de *A*[0], *A*[1], . . .,*A*[*k*].


cur
no move
C
B C D A E H G F
0 1 2 3 4 5 6 7
no move
D
B C D A E H G F
0 1 2 3 4 5 6 7
move
A
B C D E H G F
0 1 2 3 4 5 6 7
move
B C D E H G F
0 1 2 3 4 5 6 7
A
insert
move
B C D E H G F
0 1 2 3 4 5 6 7
no move
E
A B C D E H G F
0 1 2 3 4 5 6 7
no move
H
A B C D E H G F
0 1 2 3 4 5 6 7
G
move
G
A B C D E H F
0 1 2 3 4 5 6 7
no move
insert
A B C D E H F
0 1 2 3 4 5 6 7
F
move
F
A B C D E G H
0 1 2 3 4 5 6 7
move
A B C D E G H
0 1 2 3 4 5 6 7
F
no move
insert
A B C D E G H
0 1 2 3 4 5 6 7
A B C D E F G H
0 1 2 3 4 5 6 7
Done!

Métodos de java.util para Arreglos y números Random

- equals(*A*, *B*)
- fill(*A*, *x*)
- copyOf(*A*, *n*)
- copyOfRange(*A*, *s*, *t*)
- toString(*A*)
- sort(*A*)
- binarySearch(*A*, *x*)

- nextBoolean( ):
- nextDouble( ): entre 0.0 y 1.0.
- nextInt( ):
- nextInt(*n*): entre 0 y n
- setSeed(*s*): long *s*.

Prueba de los métodos

```
1  import java.util.Arrays;
2  import java.util.Random;
3  /** Program showing some array uses. */
4  public class ArrayTest {
5    public static void main(String[ ] args) {
6      int data[ ] = new int[10];
7      Random rand = new Random( );              // a pseudo-random number generator
8      rand.setSeed(System.currentTimeMillis( ));          // use current time as a seed
9      // fill the data array with pseudo-random numbers from 0 to 99, inclusive
10     for (int i = 0; i < data.length; i++)
11       data[i] = rand.nextInt(100);                  // the next pseudo-random number
12     int[ ] orig = Arrays.copyOf(data, data.length); // make a copy of the data array
13     System.out.println("arrays equal before sort: "+Arrays.equals(data, orig));
14     Arrays.sort(data);                        // sorting the data array (orig is unchanged)
15     System.out.println("arrays equal after sort: " + Arrays.equals(data, orig));
16     System.out.println("orig = " + Arrays.toString(orig));
17     System.out.println("data = " + Arrays.toString(data));
18   }
19 }
```

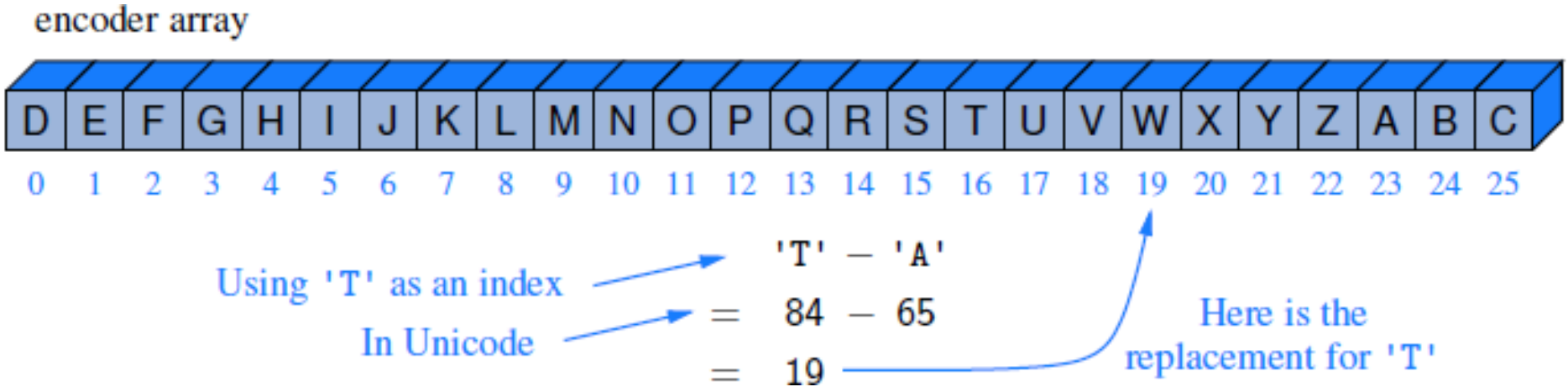
Prueba de los métodos

```
arrays equal before sort: true
arrays equal after sort: false
orig = [41, 38, 48, 12, 28, 46, 33, 19, 10, 58]
data = [10, 12, 19, 28, 33, 38, 41, 46, 48, 58]
```

Criptografía Simple con Arreglos de Caracteres

- La encriptación de Julio César.
- Los strings son inmutables, no se puede editar directamente una instancia
- Una técnica es crear un arreglo de caracteres equivalente, editar el arreglo y luego crear el string basado en el arreglo
- Se puede crear un arreglo de caracteres coincidente con el string S usando el método**,** ***S*.toCharArray( )**. Ejemplo, si ***s*="bird"**, el método retorna el arreglo ***A*={'b', 'i', 'r', 'd'}**.
- Si tenemos un arreglo ***A*={'b', 'i', 'r', 'd'}**, con **String(*A*)** producimos el string **"bird“**
- Afortunadamente el hecho que los caracteres se representen en Unicode un código de enteros y como el código del alfabeto latino es consecutivo podemos utilizar esta propiedad (por simplicidad nos restringimos a las mayúsculas)

Criptografía Simple con Arreglos de Caracteres




```
Encryption code = DEFGHIJKLMNOPQRSTUVWXYZABC
Decryption code = XYZABCDEFGHIJKLMNOPQRSTUVW
Secret:  WKH HDJOH LV LQ SODB; PHHW DW MRH'V.
Message: THE EAGLE IS IN PLAY; MEET AT JOE'S.
```

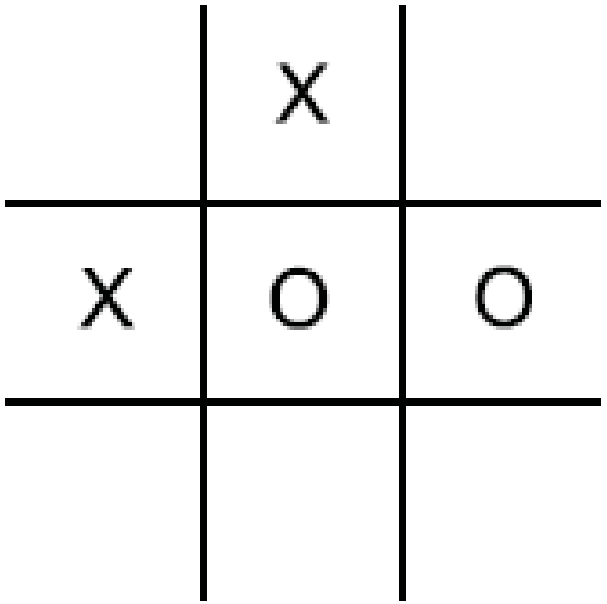
H-A
72-65=7 -> K
E-A
69-65=4 -> H

Matriz

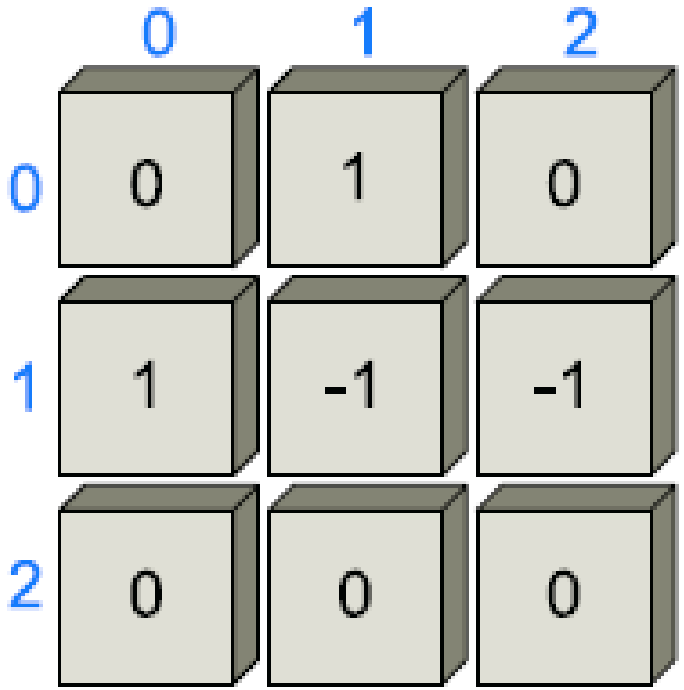
```
int[ ][ ] data = new int[8][10];
data[i][i+1] = data[i][i] + 3;
j = data.length;          // j is 8
k = data[4].length;       // k is 10
// data[3][5] is 100
// data[6][2] is 632
```

	0	1	2	3	4	5	6	7	8	9
0	22	18	709	5	33	10	4	56	82	440
1	45	32	830	120	750	660	13	77	20	105
2	4	880	45	66	61	28	650	7	510	67
3	940	12	36	3	20	100	306	590	0	500
4	50	65	42	49	88	25	70	126	83	288
5	398	233	5	83	59	232	49	8	365	90
6	33	58	632	87	94	5	59	204	120	829
7	62	394	3	4	102	140	183	390	16	26

Ta-Te-Ti

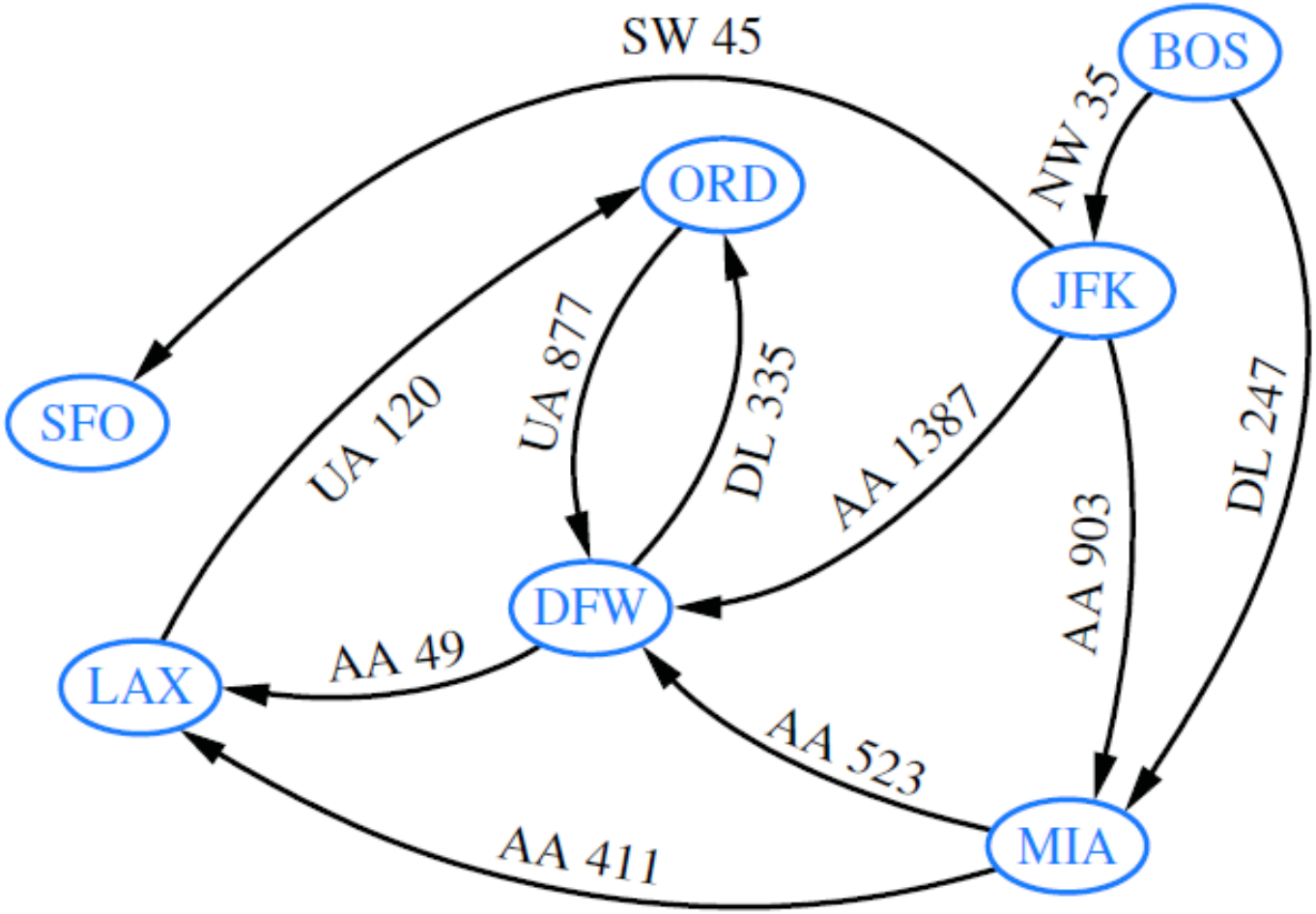


playing board

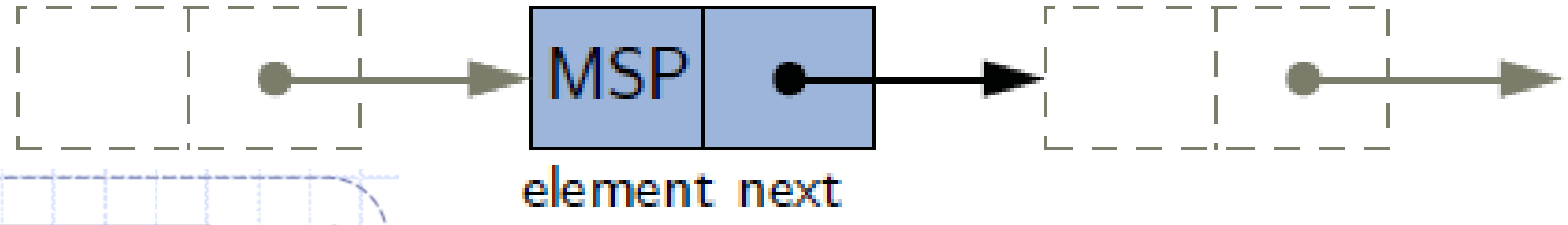


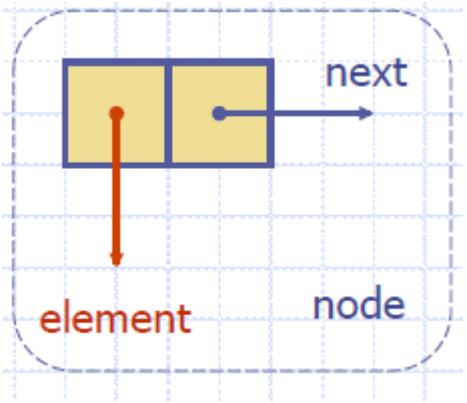
board array

Lista simplemente enlazada



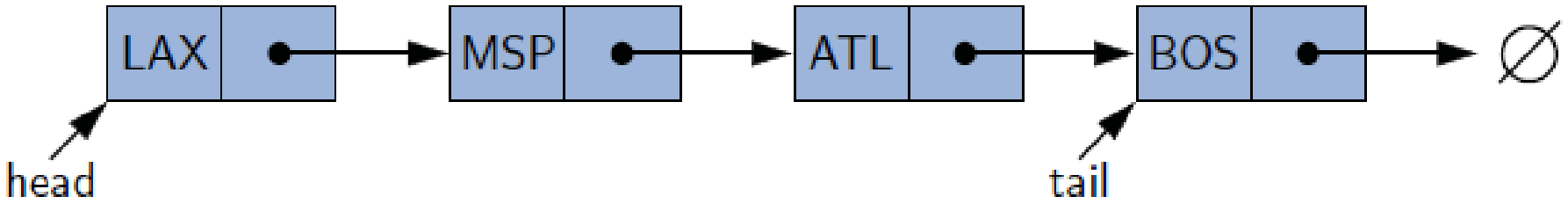
Lista simplemente enlazada



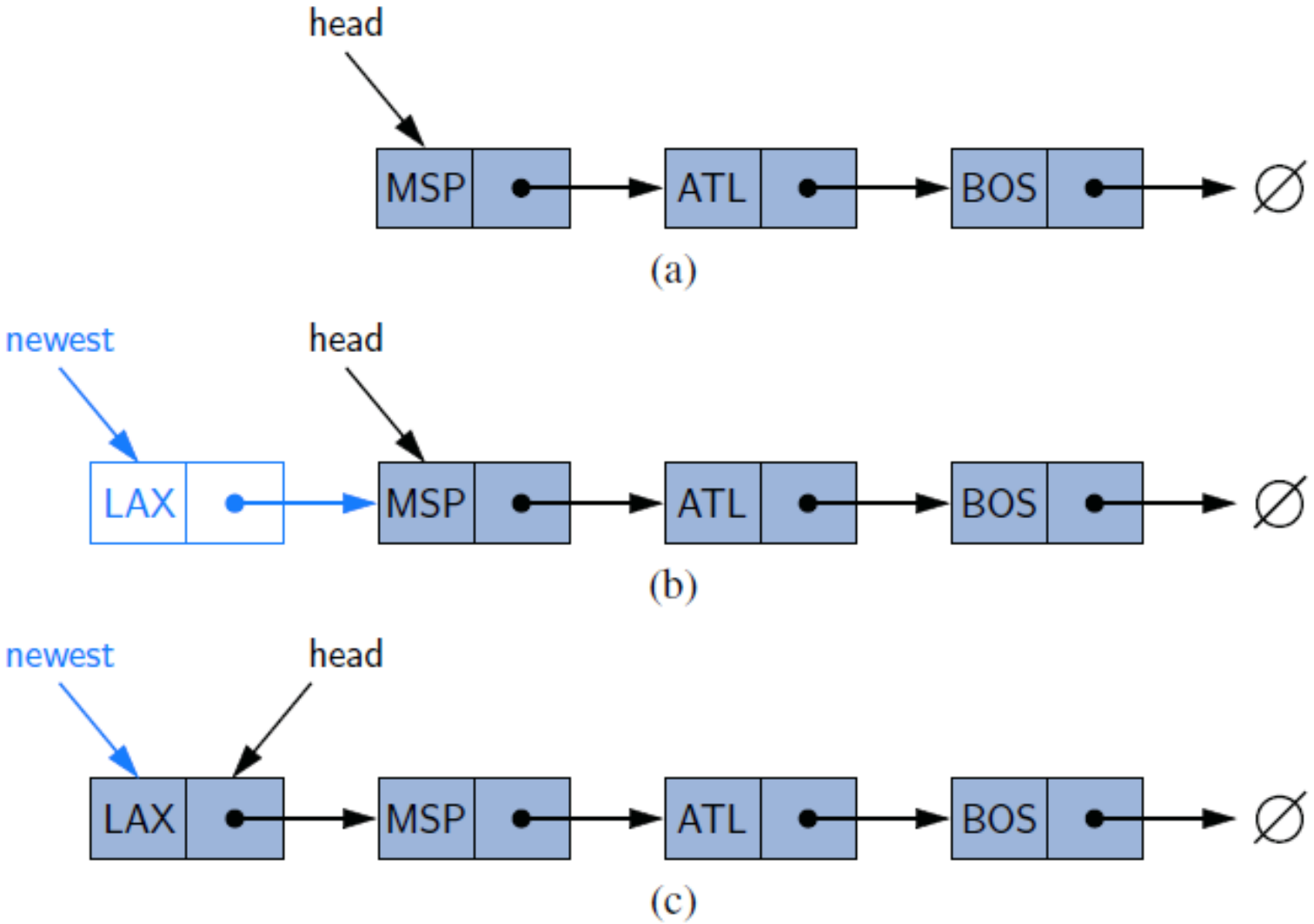


Propiedades:

- No tiene prefijado su tamaño
- Su tamaño es proporcional a su cardinalidad actual



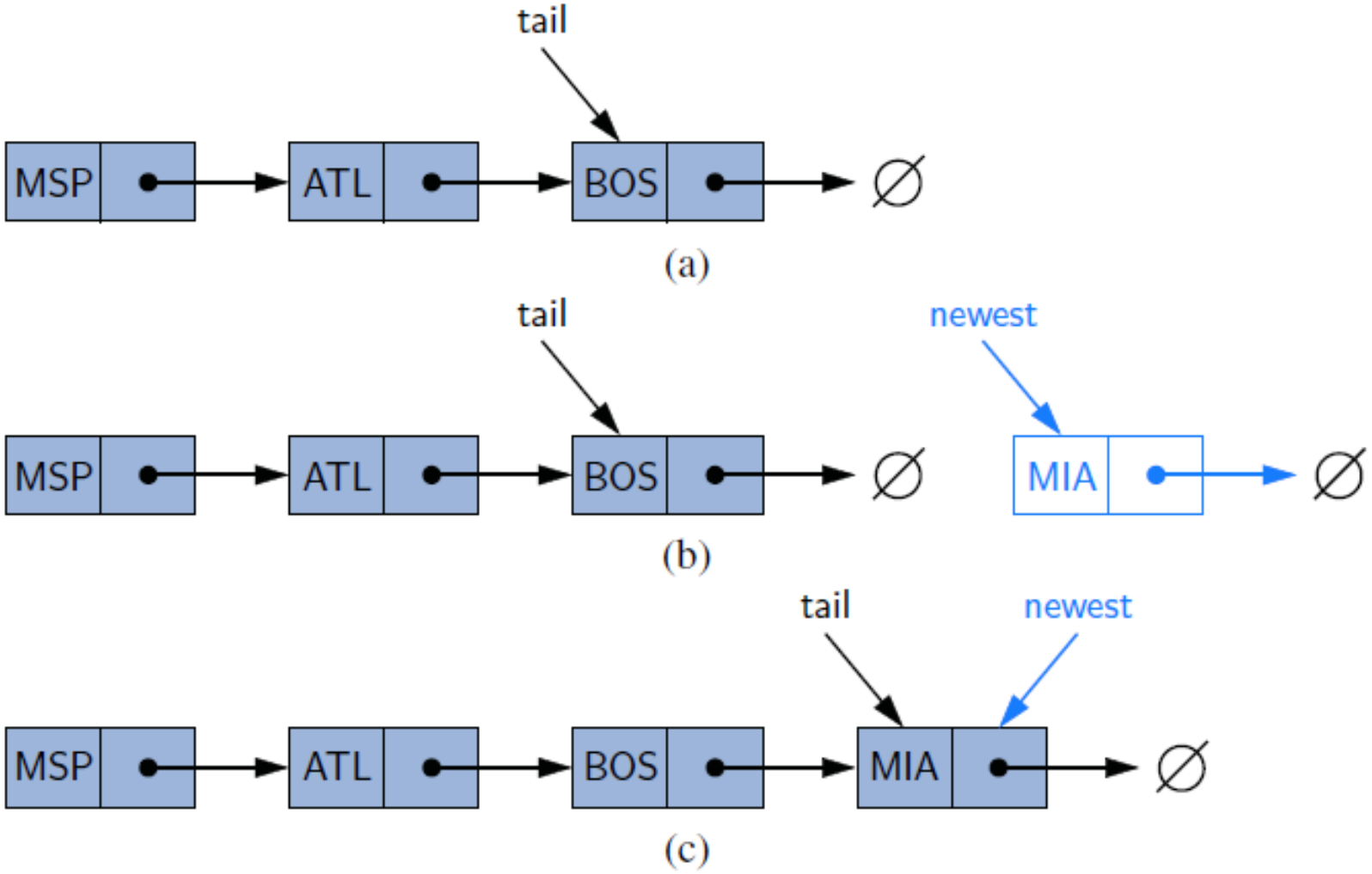
Inserción por la cabeza



Pseudocódigo de inserción por la cabeza

```
Algorithm addFirst(e):
    newest = Node(e)   {create new node instance storing reference to element e}
    newest.next = head      {set new node's next to reference the old head node}
    head = newest                    {set variable head to reference the new node}
    size = size + 1                                        {increment the node count}
```

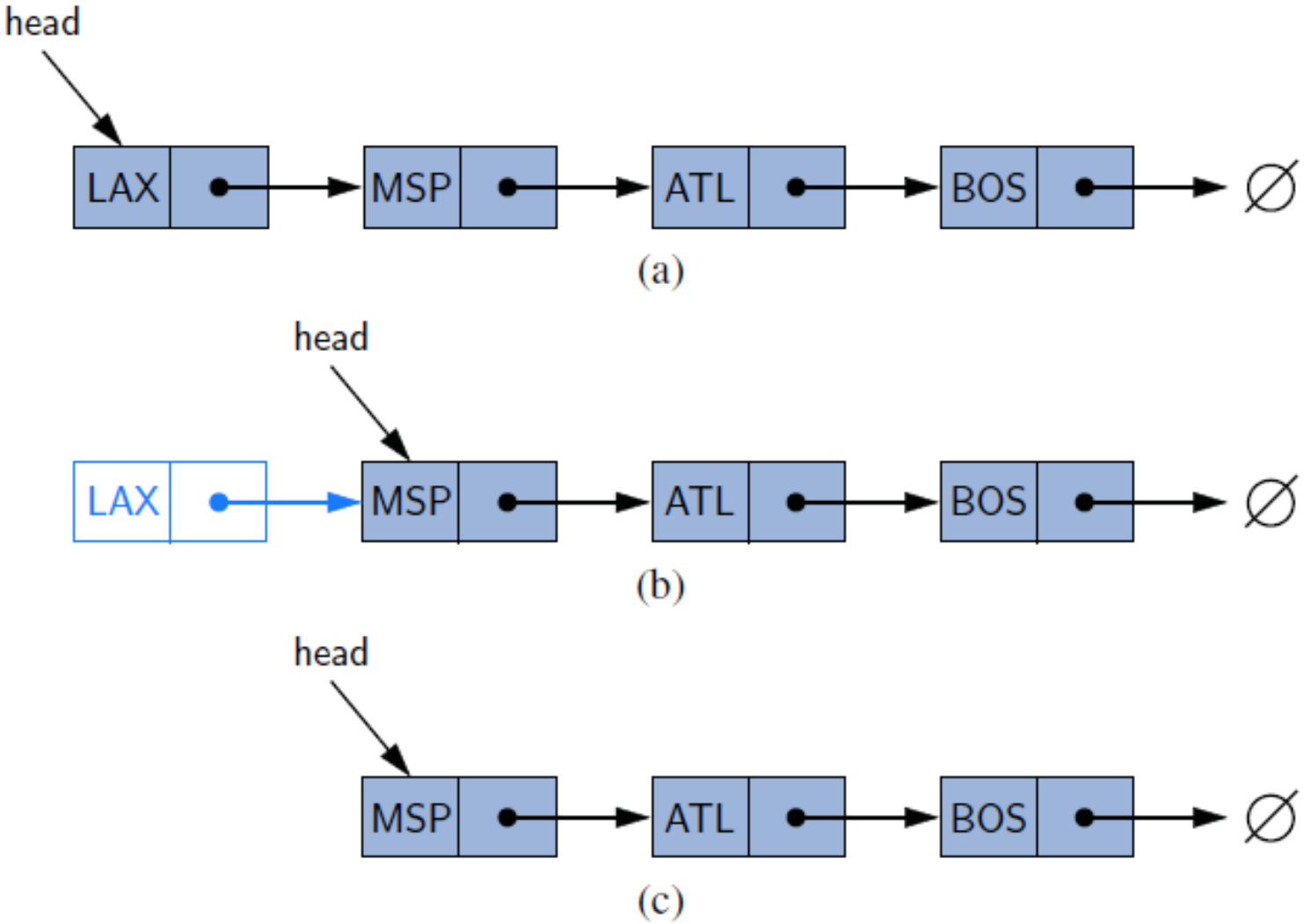
Inserción por la cola



Pseudocódigo de inserción por la cola

```
Algorithm addLast(e):
    newest = Node(e)      {create new node instance storing reference to element e}
    newest.next = null    {set new node's next to reference the null object}
    tail.next = newest    {make old tail node point to new node}
    tail = newest         {set variable tail to reference the new node}
    size = size + 1       {increment the node count}
```

Eliminar el primer elemento de la lista



Pseudocódigo de eliminar el primer elemento de la lista

```
Algorithm removeFirst( ):
    if head == null then
        the list is empty.
    head = head.next                    {make head point to next node (or null)}
    size = size − 1                     {decrement the node count}
```

Implementando una Clase de Lista Simple Enlazada

- size( ): Retorna el número de elementos de la lista.
- isEmpty( ): Retorna true si la lista está vacía de lo contrario false.
- first( ): Retorna (pero no elimina) al primer elemento de la lista.
- last( ): Retorna (pero no elimina) al último elemento de la lista.
- addFirst(e): agrega nuevo elemento al frente de la lista.
- addLast(e): agrega nuevo elemento al final de la lista.
- removeFirst( ): Remueve y retorna el primer elemento de la lista.

Implementando una Clase de Lista Simple Enlazada

```
1  public class SinglyLinkedList<E> {
2    //---------------- nested Node class ----------------
3    private static class Node<E> {
4      private E element;                  // reference to the element stored at this node
5      private Node<E> next;               // reference to the subsequent node in the list
6      public Node(E e, Node<E> n) {
7        element = e;
8        next = n;
9      }
10     public E getElement( ) { return element; }
11     public Node<E> getNext( ) { return next; }
12     public void setNext(Node<E> n) { next = n; }
13   } //---------- end of nested Node class ----------
     ... rest of SinglyLinkedList class will follow ...
```

Implementando una Clase de Lista

```
1   public class SinglyLinkedList<E> {
...     (nested Node class goes here)
14      // instance variables of the SinglyLinkedList
15      private Node<E> head = null;          // head node of the list (or null if empty)
16      private Node<E> tail = null;          // last node of the list (or null if empty)
17      private int size = 0;                 // number of nodes in the list
18      public SinglyLinkedList( ) { }        // constructs an initially empty list
19      // access methods
20      public int size( ) { return size; }
21      public boolean isEmpty( ) { return size == 0; }
22      public E first( ) {                   // returns (but does not remove) the first element
23        if (isEmpty( )) return null;
24        return head.getElement( );
25      }
26      public E last( ) {                    // returns (but does not remove) the last element
27        if (isEmpty( )) return null;
28        return tail.getElement( );
29      }
30      // update methods
31      public void addFirst(E e) {           // adds element e to the front of the list
32        head = new Node<>(e, head);         // create and link a new node
33        if (size == 0)
34          tail = head;                      // special case: new node becomes tail also
35        size++;
36      }
```

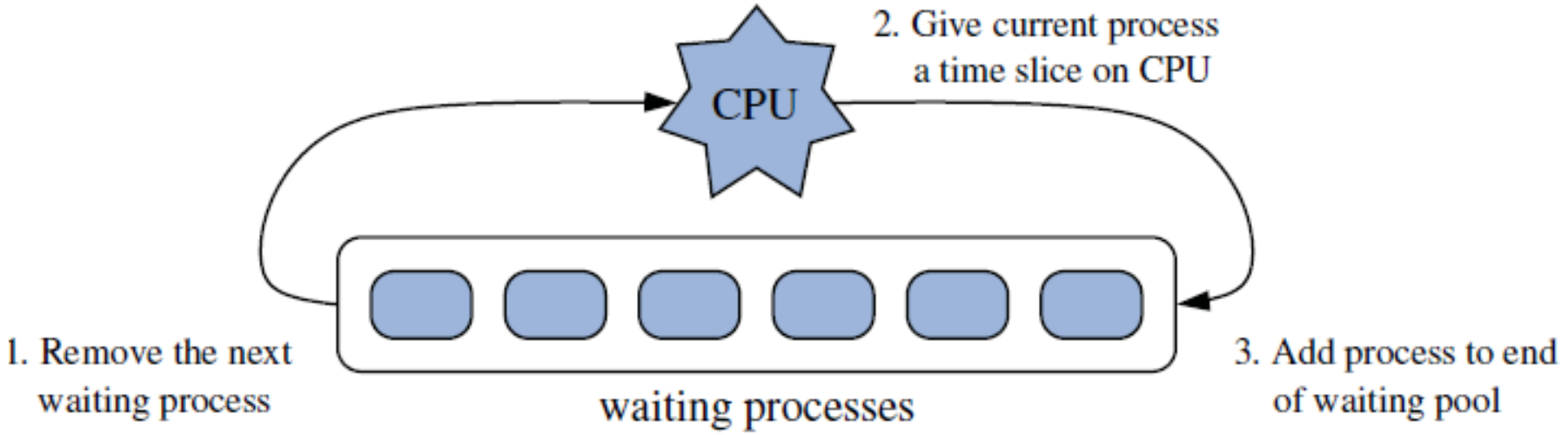
Accessors methods

Implementando una Clase de Lista Simple Enlazada

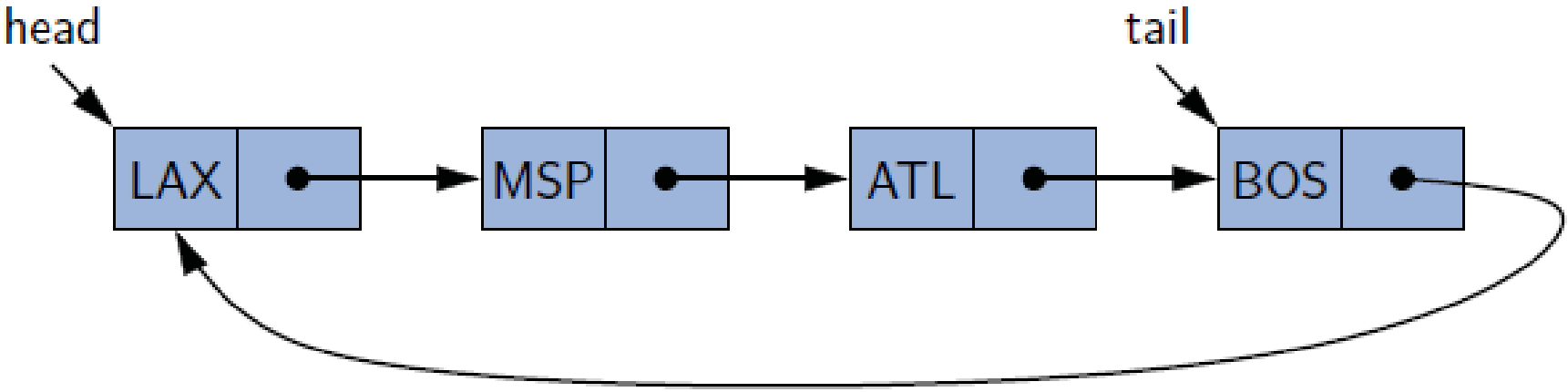
```
37    public void addLast(E e) {                  // adds element e to the end of the list
38      Node<E> newest = new Node<>(e, null);     // node will eventually be the tail
39      if (isEmpty())
40        head = newest;                          // special case: previously empty list
41      else
42        tail.setNext(newest);                   // new node after existing tail
43      tail = newest;                            // new node becomes the tail
44      size++;
45    }
46    public E removeFirst( ) {                   // removes and returns the first element
47      if (isEmpty( )) return null;              // nothing to remove
48      E answer = head.getElement( );
49      head = head.getNext( );                   // will become null if list had only one node
50      size--;
51      if (size == 0)
52        tail = null;                            // special case as list is now empty
53      return answer;
54    }
55  }
```

Lista Enlazada Circular

- Round-Robin Scheduling (time slice)
  - 1. proceso *p* = *L*.removeFirst( )
  - 2. Dar un time slice al proceso *p*
  - 3. *L*.addLast(*p*)



Diseño e implementación de una lista circular

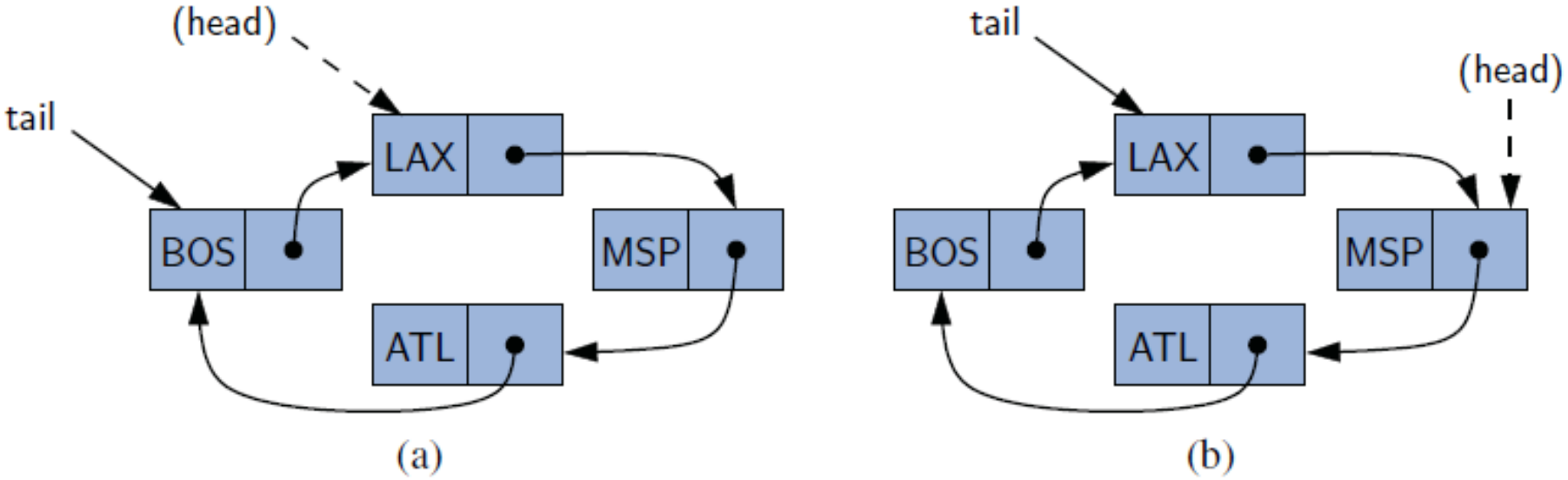


- Se agrega el método *rotate* a la lista simple-
- *rotate( ):* mueve el primer elemento al final de la lista.
- Con este método el schedule de CPU puede ser más eficientemente implementado.

round-robin scheduling

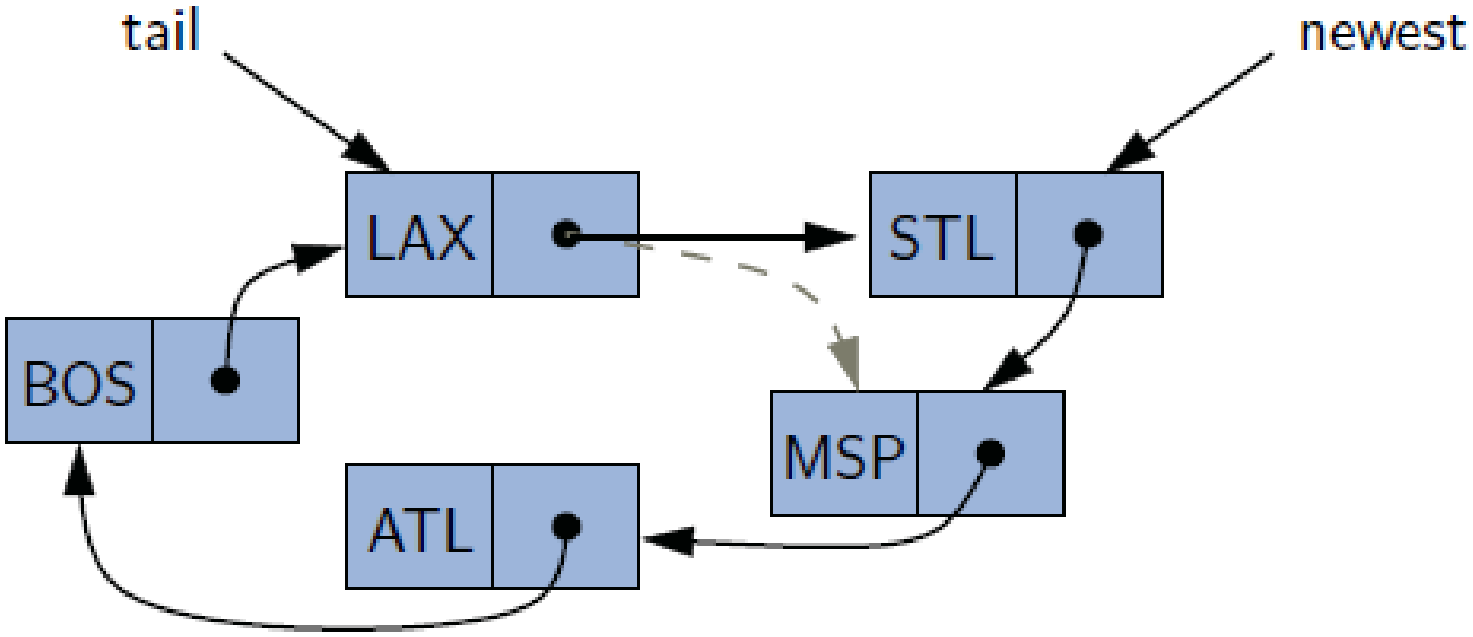
- 1. Give a time slice to process *C*.first( )
- 2. *C*.rotate( )
- Hacemos una optimización adicional: ya no mantenemos explícitamente la referencia de la cabeza. Mientras mantengamos una referencia a la cola, podemos localizar la cabeza como *tail.getNext ()*.

Operaciones en una lista circular



- (a) antes de la rotación (b) luego de la rotación

Operaciones en una lista circular



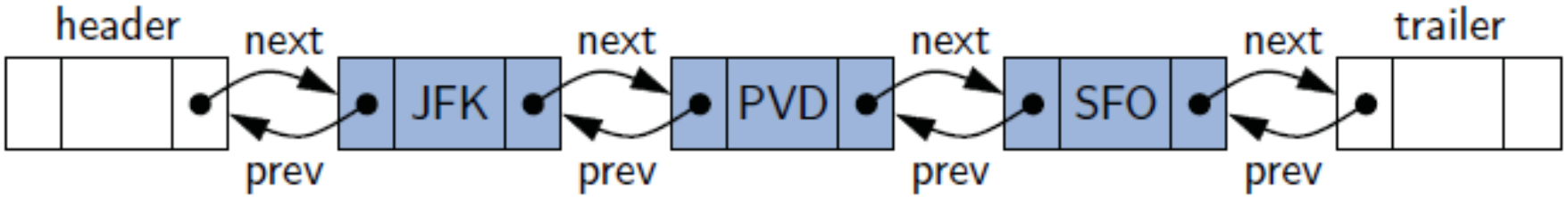
- addFirst(STL)
- Notemos que cuando la operación se completa, SLT es el primer elemento de la lista, porque es almacenado en la cabeza implícita , *tail.getNext*( ).

```
1   public class CircularlyLinkedList<E> {
...     (nested node class identical to that of the SinglyLinkedList class)
14    // instance variables of the CircularlyLinkedList
15    private Node<E> tail = null;                  // we store tail (but not head)
16    private int size = 0;                         // number of nodes in the list
17    public CircularlyLinkedList( ) { }            // constructs an initially empty list
18    // access methods
19    public int size( ) { return size; }
20    public boolean isEmpty( ) { return size == 0; }
21    public E first( ) {                  // returns (but does not remove) the first element
22      if (isEmpty( )) return null;
23      return tail.getNext( ).getElement( );       // the head is *after* the tail
24    }
25    public E last( ) {                   // returns (but does not remove) the last element
26      if (isEmpty( )) return null;
27      return tail.getElement( );
28    }
29    // update methods
30    public void rotate( ) {              // rotate the first element to the back of the list
31      if (tail != null)                           // if empty, do nothing
32        tail = tail.getNext( );                   // the old head becomes the new tail
33    }
```

```
34    public void addFirst(E e) {                    // adds element e to the front of the list
35      if (size == 0) {
36        tail = new Node<>(e, null);
37        tail.setNext(tail);                        // link to itself circularly
38      } else {
39        Node<E> newest = new Node<>(e, tail.getNext());
40        tail.setNext(newest);
41      }
42      size++;
43    }
44    public void addLast(E e) {                     // adds element e to the end of the list
45      addFirst(e);                                 // insert new element at front of list
46      tail = tail.getNext();                       // now new element becomes the tail
47    }
48    public E removeFirst() {                       // removes and returns the first element
49      if (isEmpty()) return null;                  // nothing to remove
50      Node<E> head = tail.getNext();
51      if (head == tail) tail = null;               // must be the only node left
52      else tail.setNext(head.getNext());           // removes "head" from the list
53      size--;
54      return head.getElement();
55    }
56  }
```

Lista Doblemente Enlazada

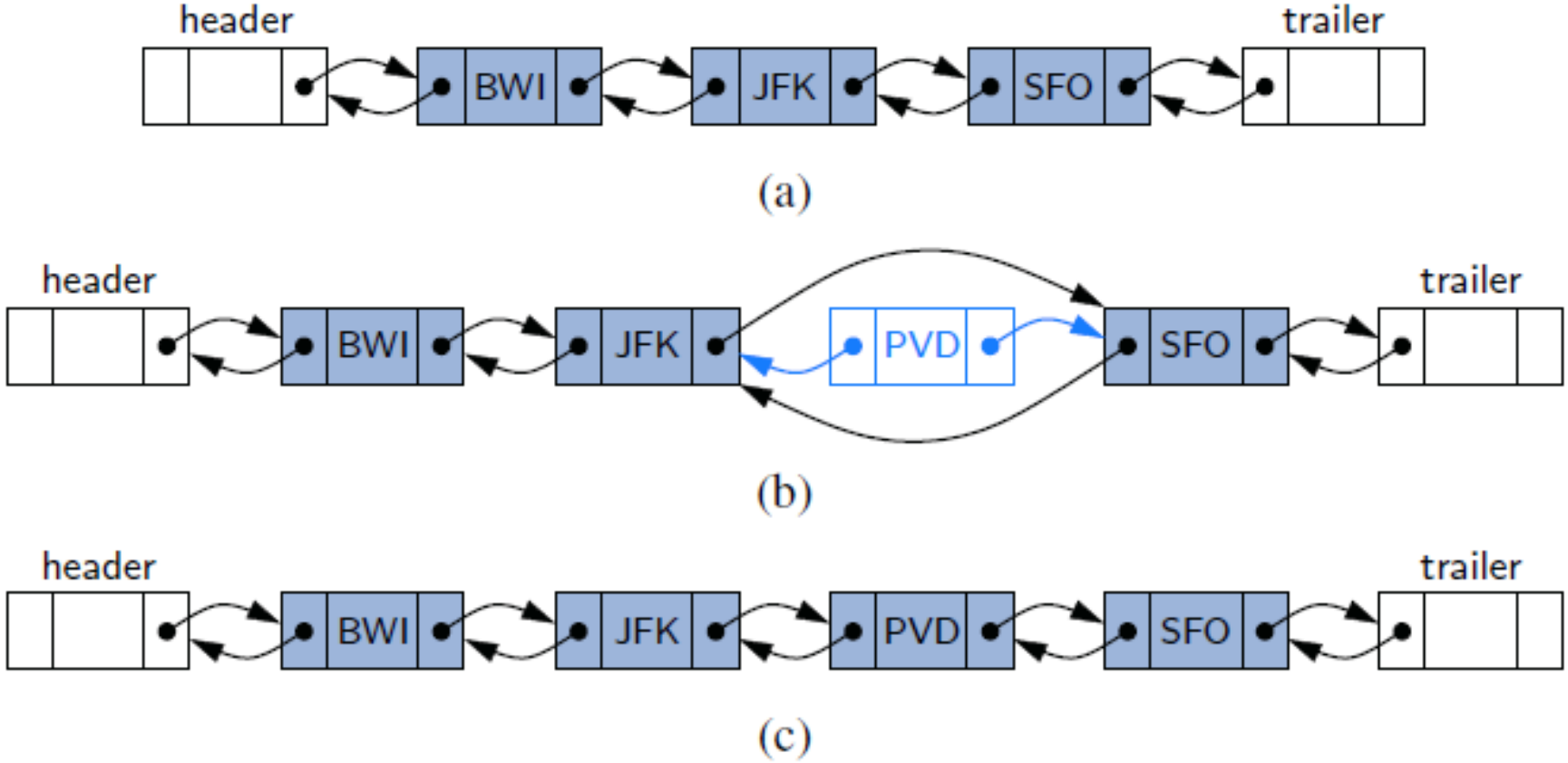
- Centinelas (nodos “tontos”) de encabezado y cola



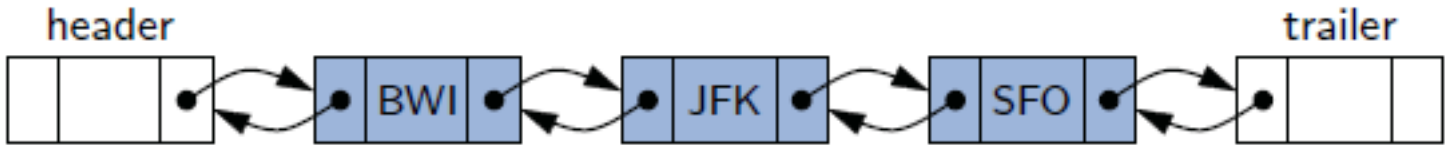
Las ventajas de usar centinelas

- El costo de memoria se justifica por la simplificación del algoritmo.
- Podemos tratar todas las inserciones de una manera unificada, porque un nuevo nodo siempre se colocará entre un par de nodos existentes.
- De manera similar ocurre con las eliminaciones.
- Se eliminan los casos especiales relativos al primer elemento y último (if).

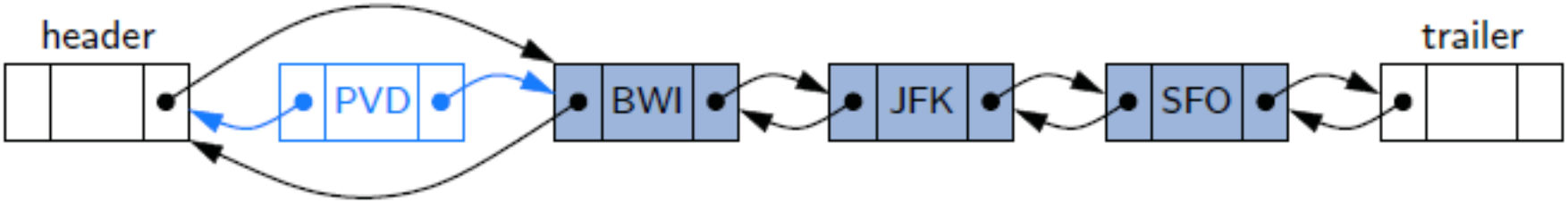
Inserción


header
trailer
BWI
JFK
SFO
(a)
header
trailer
BWI
JFK
PVD
SFO
(b)
header
trailer
BWI
JFK
PVD
SFO
(c)

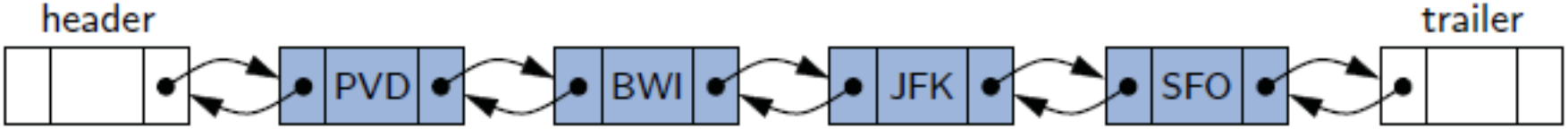
Inserción adelante



(a)

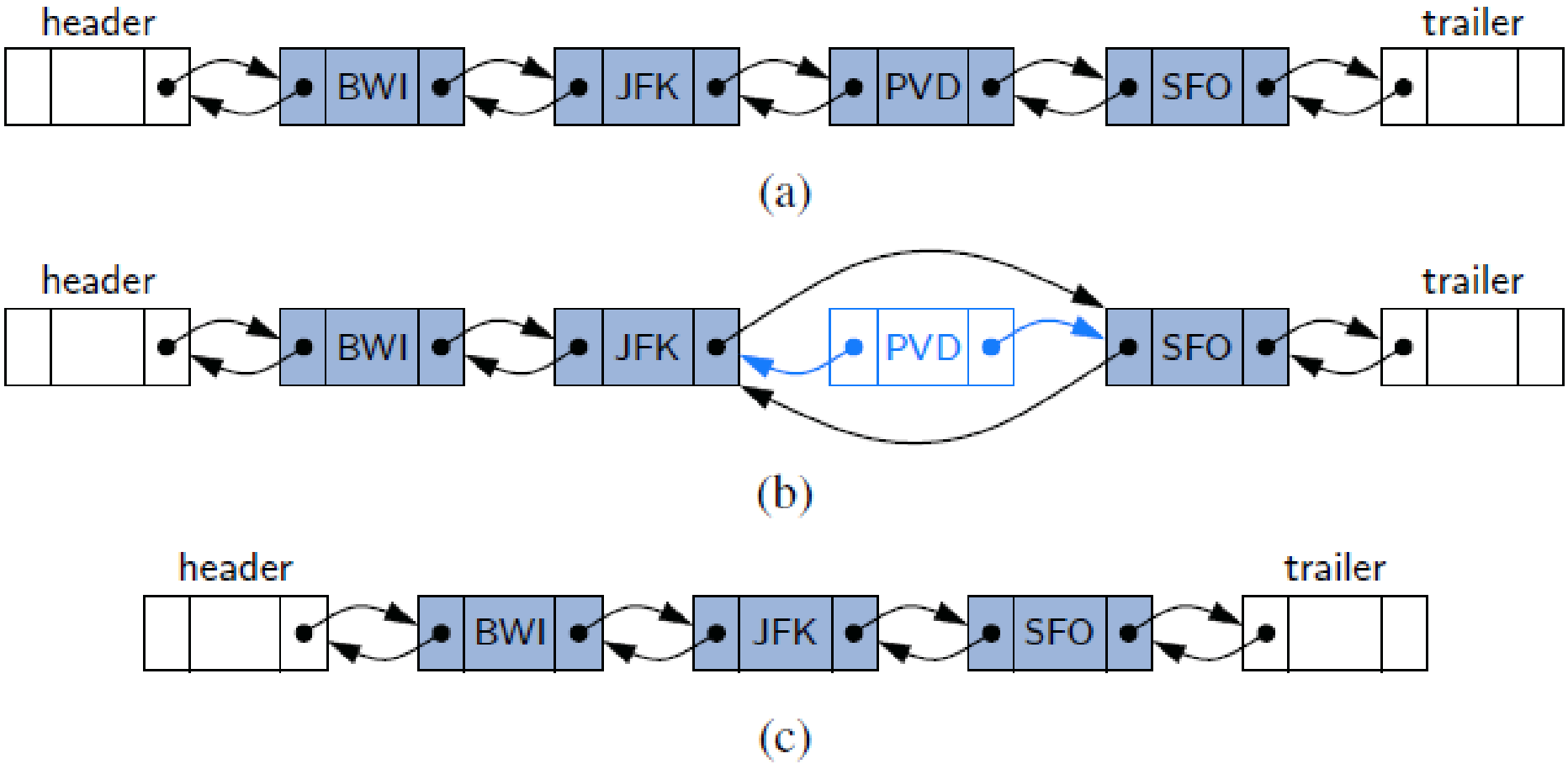


(b)



(c)

Eliminación



Implementación

- size( ): Returns the number of elements in the list.
- isEmpty( ): Returns true if the list is empty, and false otherwise.
- first( ): Returns (but does not remove) the first element in the list.
- last( ): Returns (but does not remove) the last element in the list.
- addFirst(*e*): Adds a new element to the front of the list.
- addLast(*e*): Adds a new element to the end of the list.
- removeFirst( ): Removes and returns the first element of the list.
- removeLast( ): Removes and returns the last element of the list.

```
1  /** A basic doubly linked list implementation. */
2  public class DoublyLinkedList<E> {
3    //--------------- nested Node class ---------------
4    private static class Node<E> {
5      private E element;                  // reference to the element stored at this node
6      private Node<E> prev;               // reference to the previous node in the list
7      private Node<E> next;               // reference to the subsequent node in the list
8      public Node(E e, Node<E> p, Node<E> n) {
9        element = e;
10       prev = p;
11       next = n;
12     }
13     public E getElement( ) { return element; }
14     public Node<E> getPrev( ) { return prev; }
15     public Node<E> getNext( ) { return next; }
16     public void setPrev(Node<E> p) { prev = p; }
17     public void setNext(Node<E> n) { next = n; }
18   } //---------- end of nested Node class ----------
19
20   // instance variables of the DoublyLinkedList
21   private Node<E> header;                 // header sentinel
22   private Node<E> trailer;                // trailer sentinel
23   private int size = 0;                   // number of elements in the list
```

```
24    /** Constructs a new empty list. */
25    public DoublyLinkedList( ) {
26      header = new Node<>(null, null, null);        // create header
27      trailer = new Node<>(null, header, null);     // trailer is preceded by header
28      header.setNext(trailer);                      // header is followed by trailer
29    }
30    /** Returns the number of elements in the linked list. */
31    public int size( ) { return size; }
32    /** Tests whether the linked list is empty. */
33    public boolean isEmpty( ) { return size == 0; }
34    /** Returns (but does not remove) the first element of the list. */
35    public E first( ) {
36      if (isEmpty( )) return null;
37      return header.getNext( ).getElement( );       // first element is beyond header
38    }
39    /** Returns (but does not remove) the last element of the list. */
40    public E last( ) {
41      if (isEmpty( )) return null;
42      return trailer.getPrev( ).getElement( );      // last element is before trailer
43    }
```

```
44    // public update methods
45    /** Adds element e to the front of the list. */
46    public void addFirst(E e) {
47      addBetween(e, header, header.getNext( ));          // place just after the header
48    }
49    /** Adds element e to the end of the list. */
50    public void addLast(E e) {
51      addBetween(e, trailer.getPrev( ), trailer);        // place just before the trailer
52    }
53    /** Removes and returns the first element of the list. */
54    public E removeFirst( ) {
55      if (isEmpty( )) return null;                       // nothing to remove
56      return remove(header.getNext( ));                  // first element is beyond header
57    }
58    /** Removes and returns the last element of the list. */
59    public E removeLast( ) {
60      if (isEmpty( )) return null;                       // nothing to remove
61      return remove(trailer.getPrev( ));                 // last element is before trailer
62    }
63
```

```
64    // private update methods
65    /** Adds element e to the linked list in between the given nodes. */
66    private void addBetween(E e, Node<E> predecessor, Node<E> successor) {
67      // create and link a new node
68      Node<E> newest = new Node<>(e, predecessor, successor);
69      predecessor.setNext(newest);
70      successor.setPrev(newest);
71      size++;
72    }
73    /** Removes the given node from the list and returns its element. */
74    private E remove(Node<E> node) {
75      Node<E> predecessor = node.getPrev( );
76      Node<E> successor = node.getNext( );
77      predecessor.setNext(successor);
78      successor.setPrev(predecessor);
79      size--;
80      return node.getElement( );
81    }
82  } //---------- end of DoublyLinkedList class ----------
```

Prueba de equivalencia

- Cuando se trabaja con tipos de referencia, existen muchas nociones diferentes de lo que significa que una expresión sea igual a otra
- La expresión a == b prueba si a y b se refieren al mismo objeto (o si ambas referencias tienen el valor nulo).
- Sin embargo, para muchos tipos existe una noción de nivel superior de que dos variables se consideran "equivalentes" incluso si en realidad no se refieren a la misma instancia de la clase.
- Por ejemplo, normalmente queremos considerar que dos instancias de String son equivalentes entre sí, si representan la secuencia idéntica de caracteres.

Prueba de equivalencia

- Para admitir una noción más amplia de equivalencia, todos los tipos de objetos admiten un método denominado *equals*.
- Los usuarios de tipos de referencia deben confiar en la sintaxis *a.equals (b)*, a menos que tengan una necesidad específica de probar la noción más específica de identidad.
- El método *equals* se define formalmente en la clase *Object*, que sirve como una superclase para todos los tipos de referencia, pero esa implementación va a devolver el valor de la expresión a == b.
- Definir una noción de equivalencia más significativa requiere conocimiento sobre una clase y su representación.
- El autor de cada clase tiene la **responsabilidad** de proporcionar una implementación del método *equals*, que anula el heredado de *Object*.

Prueba de equivalencia

- Se debe tener mucho cuidado al sobreescribir la noción de igualdad, ya que la consistencia de las bibliotecas de Java depende del método *equals* que define lo que se conoce como relación de equivalencia en matemáticas, satisfaciendo las siguientes propiedades:
- **Tratamiento de nulo:** para cualquier variable de referencia no nula x, la llamada *x.equals* (nulo) debe devolver **falso** (es decir, nada es igual a nulo excepto nulo).
- **Reflexividad:** para cualquier variable de referencia no nula *x*, la llamada *x.equals (x)* debe devolver **verdadero** (es decir, un objeto debe ser igual a sí mismo).
- **Simetría:** para cualquier variable de referencia no nula *x* e *y*, las llamadas *x.equals (y)* e *y.equals (x)* deben devolver el mismo valor.
- **Transitividad:** Para cualquier variable de referencia no nula *x*, *y* y *z*, si ambas llamadas *x.equals (y)* e *y.equals (z)* devuelven **verdadero**, entonces la llamada *x.equals (z)* también debe devolver **verdadero**.

Prueba de equivalencia

- Si bien estas propiedades pueden parecer intuitivas, puede ser un desafío implementar correctamente *equivalencia* para algunas estructuras de datos, especialmente en un contexto orientado a objetos, con herencia y genéricos.

Prueba de equivalencia: arreglos

La clase **java.util.Arrays** y asumiendo que las variables a y b se refieren a objetos de arreglo:

- **a == b:** comprueba si a y b se refieren a la misma instancia de arreglo subyacente.
- **a.equals (b):** Curiosamente, esto es idéntico a *a == b*. Los arreglos no son un verdadero tipo de clase y no anula el método *Object.equals*.
- **Arrays.equals (a, b):** esto proporciona una noción más intuitiva de equivalencia, devolviendo verdadero si los arreglos tienen la misma longitud y todos los pares de elementos correspondientes son "iguales" entre sí. Más específicamente, si los elementos del arreglo son primitivos, entonces usa el estándar == para comparar valores. Si los elementos de los arreglos son un tipo de referencia, entonces hace comparaciones por pares *a [k].equals(b[k])* al evaluar la equivalencia.

Prueba de equivalencia: arreglos

Para respaldar la noción más natural de igualdad de los arreglos multidimensionales, la clase java.util.Arrays class proporciona un método adicional **Arrays.deepEquals (a, b):**

int[][] a = {{1, 2}, {3, 4}};

int[][] b = {{1, 2}, {3, 4}};

*boolean equal = Arrays.equals(a, b); // **returns false***

*boolean deepEqual = deepEquals(a, b); // **returns true***

¿Prueba de equivalencia con listas?

Clonación de estructuras de datos

- En un entorno de programación, una expectativa común es que una copia de un objeto tiene su propio estado y que, una vez hecha, la copia es independiente del original (por ejemplo, para que los cambios hechos a uno no afecten directamente al otro).
- ¿En el caso de una libreta de direcciones si la clonamos queremos que los cambios de una aparezcan en la otra?

Clonación

- Object define un método llamado *clone*
- Que se puede utilizar para producir lo que se conoce como una copia superficial (*shallow copy)* de un objeto.
- Las referencias de la copia del nuevo objeto se refiera a la misma instancia subyacente que el campo del objeto original.
- Una copia poco profunda no siempre es apropiada para todas las clases y, por lo tanto, Java desactiva intencionalmente el uso del método *clone()* declarándolo como protegido, y cuando se llama lanza una *CloneNotSupportedException*.

Clonación

- El autor de una clase debe declarar explícitamente el soporte a la clonación declarando formalmente que la clase implementa la interfaz *Clonable* y declarando una versión pública del método *clone()*.

Clonando arreglos

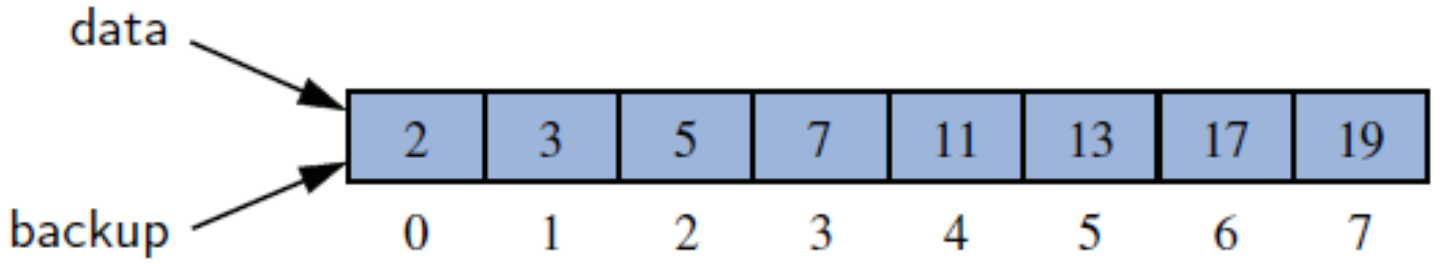
```
int[ ] data = {2, 3, 5, 7, 11, 13, 17, 19};
int[ ] backup;
backup = data;                    // warning; no es una copia
```

Lo que se debe hacer para tener una copia:

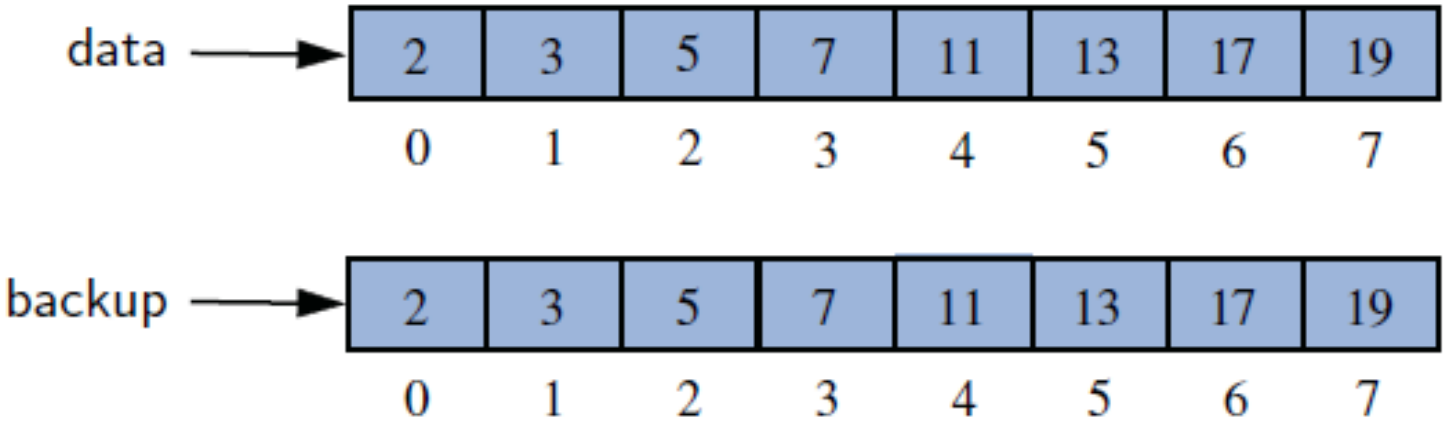
```
backup = data.clone( );
```

Clonación

- Superficial

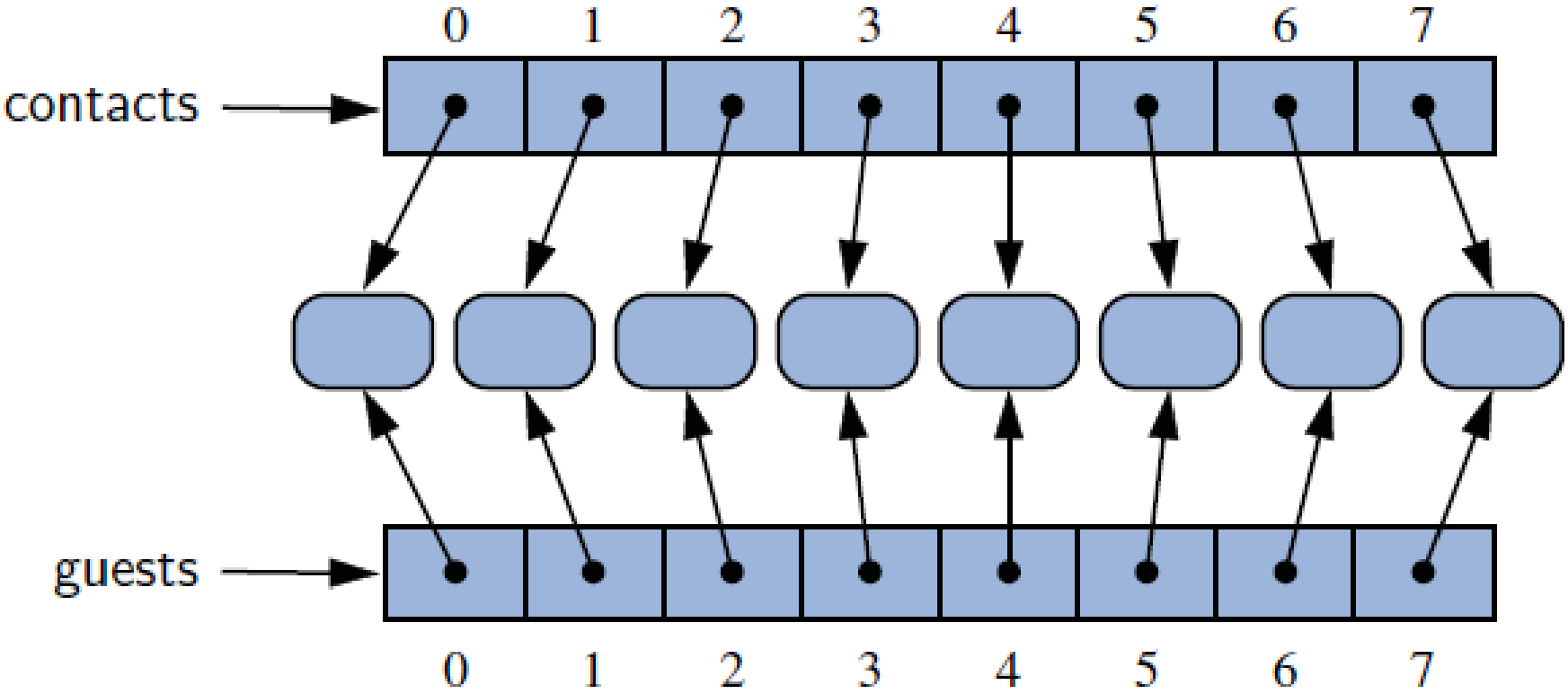


- Profunda



Clonando estructuras de datos

- Como esto solo funciona con tipos primitivos



Clonando estructuras de datos

```
Person[ ] guests = new Person[contacts.length];
for (int k=0; k < contacts.length; k++)
  guests[k] = (Person) contacts[k].clone( );        // returns Object type
```

```
1  public static int[ ][ ] deepClone(int[ ][ ] original) {
2    int[ ][ ] backup = new int[original.length][ ];      // create top-level array of arrays
3    for (int k=0; k < original.length; k++)
4      backup[k] = original[k].clone( );                  // copy row k
5    return backup;
6  }
```

Clonando Lista enlazada

public class SinglyLinkedList<E> implements Cloneable {

```
1   public SinglyLinkedList<E> clone( ) throws CloneNotSupportedException {
2     // always use inherited Object.clone() to create the initial copy
3     SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone( ); // safe cast
4     if (size > 0) {                                  // we need independent chain of nodes
5       other.head = new Node<>(head.getElement( ), null);
6       Node<E> walk = head.getNext( );    // walk through remainder of original list
7       Node<E> otherTail = other.head;    // remember most recently created node
8       while (walk != null) {                         // make a new node storing same element
9         Node<E> newest = new Node<>(walk.getElement( ), null);
10        otherTail.setNext(newest);                   // link previous node to this one
11        otherTail = newest;
12        walk = walk.getNext( );
13      }
14    }
15    return other;
16  }
```

Bibliografía

- Data Structures and Algorithms in Java™. Sixth Edition. Michael T. Goodrich, Department of Computer Science University of California. Roberto Tamassia, Department of Computer Science Brown University. Michael H. Goldwasser, Department of Mathematics and Computer Science Saint Louis University. Wiley. 2014.
- Accesoria: http://www.arquitecturajava.com/uso-de-java-generics/