Algoritmos y Estructuras de Datos:

Listas de Posiciones

Guillermo Román Díez, Lars-Åke Fredlund

Universidad Politécnica de Madrid

Curso 2021/2022

Motivación

Motivación

- Una **lista** es un TAD contenedor que consiste en una **secuencia lineal** de elementos
- El acceso (búsqueda) a los elementos *suele* ser secuencial o lineal
- No está acotada (idealmente), puede crecer de acuerdo a las necesidades del programa y de la capacidad del computador

Motivación

- Una **lista** es un TAD contenedor que consiste en una **secuencia lineal** de elementos
- El acceso (búsqueda) a los elementos *suele* ser secuencial o lineal
- No está acotada (idealmente), puede crecer de acuerdo a las necesidades del programa y de la capacidad del computador
- Ejemplos de listas son `IndexedList` y `PositionList` (lista de posiciones)

Reflexiones sobre `IndexedList<E>`

- ¿Qué inconvenientes tiene programar con `IndexedList<E>`?

Reflexiones sobre `IndexedList<E>`

- ¿Qué inconvenientes tiene programar con `IndexedList<E>`?
- Se usan índices numéricos (ints) para acceder secuencialmente a los elementos – *¿realmente es siempre eficiente?*

Reflexiones sobre `IndexedList<E>`

- ¿Qué inconvenientes tiene programar con `IndexedList<E>`?
- Se usan índices numéricos (ints) para acceder secuencialmente a los elementos – *¿realmente es siempre eficiente?*
- insertar/borrar elementos en una lista indexada es lento – O(n) (esté implementada con un array o con una lista enlazada)

Una lista "más abstracta": `PositionList<E>`

- No usa enteros (ints) como índices
- Usa objetos (*nodos* o *posiciones*) para trabajar con la lista, que además son "*persistentes*":
  - Si sabemos que un elemento *e* es alcanzable con el objeto *posición obj* en una lista *list*
  - y con ellos podemos hacer cualquier cambio en la lista *list* (excepto el borrado)
  - después podemos volver acceder a *e* usando el objecto *posición obj*
- No hay ninguna relación de orden entre dos objetos `Position<E>`, mientras que en los índices sí que la hay (p.e. $1 < 2$)
- El acceso y recorrido de los elementos de una `PositionList<E>` se hace usando objetos de tipo `Position<E>`

Interfaz `Position<E>`

- Se trata del interfaz que usamos para representar el concepto de **nodo abstracto**
- Únicamente permite acceder al contenido del nodo
- No permite modificar el contenido del nodo y cambiar su posición en la estructura de datos
- Se utilizará posteriormente en los *árboles* y en los *grafos*

```
public interface Position<E> {

  public E element();

}
```

- Ahora vamos a ver el interfaz `PositionList<E>`, que hace uso del interfaz `Position<E>`

Interfaz `PositionList<E>`

```
public interface PositionList<E> extends Iterable<E> {

  public int size();

  public boolean isEmpty();

  public Position<E> first();

  public Position<E> last();

  public Position<E> next(Position<E> p)
      throws IllegalArgumentException;

  public Position<E> prev(Position<E> p)
      throws IllegalArgumentException;
  ...
```

Interfaz PositionList<E>

```
  public void addFirst(E elem);

  public void addLast(E elem);

  public void addBefore(Position<E> p, E elem)
       throws IllegalArgumentException;

  public void addAfter(Position<E> p, E elem)
       throws IllegalArgumentException;

  public E remove(Position<E> p)
       throws IllegalArgumentException;

  public E set(Position<E> p, E elem)
       throws IllegalArgumentException;

  public Object [] toArray();

  public E [] toArray(E[] a);
}
```

Interfaz `PositionList<E>` y `Position<E>`

- La descripción de los interfaces las tenéis en: `http://costa.ls.fi.upm.es/~entrega/aed/docs/aedlib/es/upm/aedlib/positionlist/package-summary.html`
- Las operaciones de `PositionList` son
  - Interrogadores: `size`, `isEmpty`
  - Acceso: `first`, `last`, `next`, `prev`
  - Inserción: `addFirst`, `addLast`, `addBefore`, `addAfter`
  - Modificación: `set`
  - Borrado: `remove`
  - Conversión: `toArray`

Interfaz `PositionList<E>` y `Position<E>`

- La descripción de los interfaces las tenéis en: `http://costa.ls.fi.upm.es/~entrega/aed/docs/aedlib/es/upm/aedlib/positionlist/package-summary.html`
- Las operaciones de `PositionList` son
  - Interrogadores: `size`, `isEmpty`
  - Acceso: `first`, `last`, `next`, `prev`
  - Inserción: `addFirst`, `addLast`, `addBefore`, `addAfter`
  - Modificación: `set`
  - Borrado: `remove`
  - Conversión: `toArray`
- La excepción `IllegalArgumentException` será lanzada cuando no se recibe una posición correcta

Implementación de `PositionList`

- Para usar una `PositionList<E>` no tiene por qué interesarnos como está implementado el interfaz, sólo nos interesa el funcionamiento de los métodos que proporciona el interfaz
  - En la realidad, conocer la implementación nos puede ayudar a mejorar la eficiencia en función del uso que vayamos a hacer de la lista
- Pero es necesario conocer las clases que implementan el interfaz, así como los constructores de la misma
- En `aedlib` incluimos la clase `NodePositionList<E>` que implementa el interfaz `PositionList<E>`

Recorrido de las listas

- Las listas de posiciones se recorren usando bucles y nodos cursor de tipo `Position<E>`
- La inicialización suele consistir hacer que el cursor apunte al primer nodo de la lista usando `l.first()` (o al último, `l.last()`)
- Para avanzar moveremos el cursor a la siguiente posición con `l.next(cursor)` o a la anterior `l.prev(cursor)`
- La condición de parada depende del problema
  - Suele incluir la condición de rango (`cursor != null`)
  - NOTA: Ojo con los posibles elementos `null`
    - ★ Pueden saltar `NullPointerException`
    - ★ La comparación debe comprobar antes que el cursor y/o el elemento no sean `null`

Ejemplos: Mostrar los elementos de una lista

Ejercicio

Mostrar los elementos de una lista con un `while`

Ejemplos: Mostrar los elementos de una lista

Ejercicio

Mostrar los elementos de una lista con un `while`

```
public static <E> void show(PositionList<E> list) {
  Position<E> cursor = list.first(); // i = 0
  while (cursor != null) {           // i<l.size()
    ...println(cursor.element());    // print(l.get(i))
    cursor = list.next(cursor);      // i++
  }
}
```

- En los comentarios vemos la analogía con el recorrido de una `IndexedList<E>`

Ejemplos: Mostrar los elementos de una lista

Ejercicio

Mostrar los elementos de una lista con un `while`

```
public static <E> void show(PositionList<E> list) {
  Position<E> cursor = list.first(); // i = 0
  while (cursor != null) {           // i<l.size()
    ...println(cursor.element());    // print(l.get(i))
    cursor = list.next(cursor);      // i++
  }
}
```

- En los comentarios vemos la analogía con el recorrido de una `IndexedList<E>`

Ejercicio

Hacer el mismo bucle pero con `for`

Ejercicios

Ejercicio

Implementar el método `member`

```
public static <E> boolean member(PositionList<E> list,
                                 E element)
```

Ejercicios

Ejercicio

Implementar el método `member`

```
public static <E> boolean member(PositionList<E> list,
                                 E element)
```

Ejercicio

Implementar el método `addBeforeElement`

```
public static <E> void
  addBeforeElement(PositionList<E> list,
                   E element,E add)
```

Ejercicios

Ejercicio

Implementar el método `member`

```
public static <E> boolean member(PositionList<E> list,
                                 E element)
```

Ejercicio

Implementar el método `addBeforeElement`

```
  public static <E> void
    addBeforeElement(PositionList<E> list,
                     E element,E add)
```

Ejercicio

Implementar el método `deleteAll`

```
public static <E> void deleteAll( E elem, PositionList<E>
    list)
```

Ejercicios

Ejercicio

Implementar el método `member`

```
public static <E> boolean member(PositionList<E> list,
                                 E element)
```

Ejercicio

Implementar el método `addBeforeElement`

```
  public static <E> void
    addBeforeElement(PositionList<E> list,
                     E element,E add)
```

Ejercicio

Implementar el método `deleteAll`

```
public static <E> void deleteAll( E elem, PositionList<E>
    list)
```

Ejercicios

Ejercicio

Implementar el método `reverseList`

```
static <E> PositionList<E> reverseList(PositionList<E> list)
```

Ejercicios

Ejercicio

Implementar el método `reverseList`

```
static <E> PositionList<E> reverseList(PositionList<E> list)
```

Ejercicio

Implementar el método `reverseInPlace`

```
static <E> void reverseInPlace(PositionList<E> list)
```

Ejercicios

Ejercicio

Implementar el método `reverseList`

```
static <E> PositionList<E> reverseList(PositionList<E> list)
```

Ejercicio

Implementar el método `reverseInPlace`

```
static <E> void reverseInPlace(PositionList<E> list)
```

Ejercicio

Implementar el método `trimToSize`

```
static <E> void trimToSize(PositionList<E> list, int size)
```

Ejercicios

Ejercicio

Implementar el método `reverseList`

```
static <E> PositionList<E> reverseList(PositionList<E> list)
```

Ejercicio

Implementar el método `reverseInPlace`

```
static <E> void reverseInPlace(PositionList<E> list)
```

Ejercicio

Implementar el método `trimToSize`

```
static <E> void trimToSize(PositionList<E> list, int size)
```

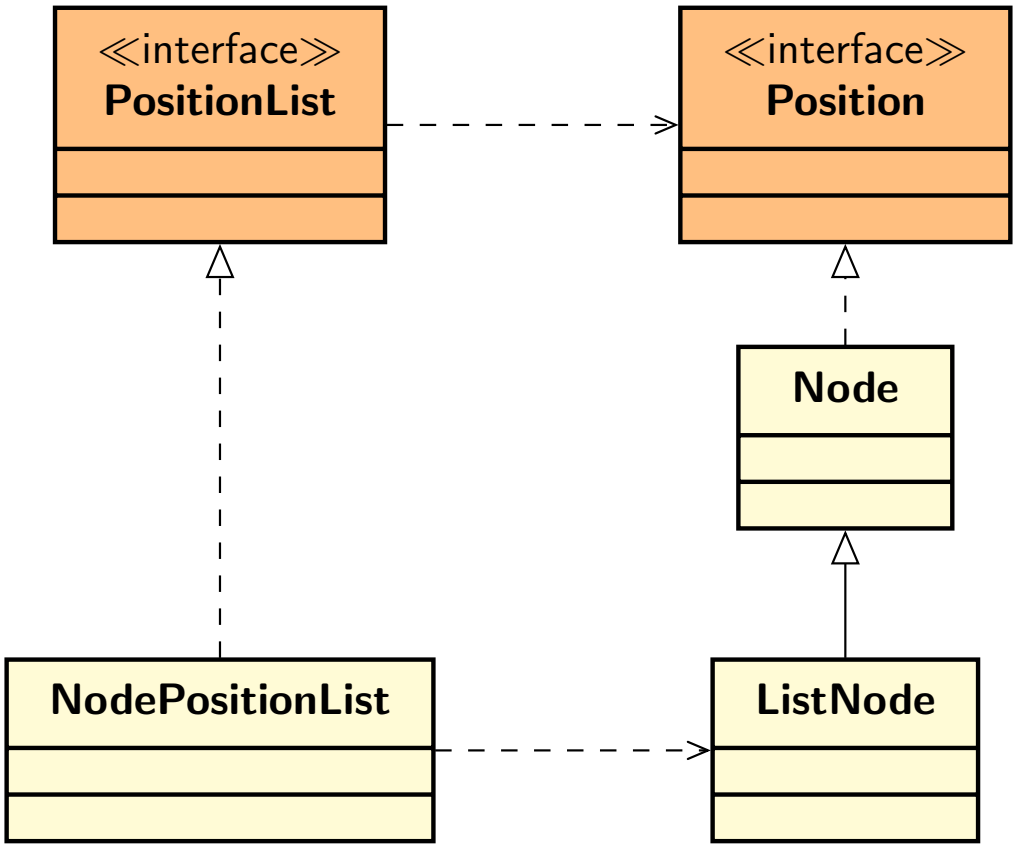
Ejercicio

Implementar el método `insertionSort`

```
static <E> PositionList<E> insertionSort(PositionList<E>
    list)
```

Implementación de `Position<E>` y `PositionList<E>`

- `Node<E,O> implements Position<E>`
- `ListNode<E> extends Node<E,PositionList<E>>`
- `NodePositionList<E> implements PositionList<E>`



Implementación de `Node<E,O>`

`Node<E,O>` implementa el interfaz `Position<E>`:

```
public interface Position<E> {
  public E element();
}
```

Implementación de `Node<E,O>`

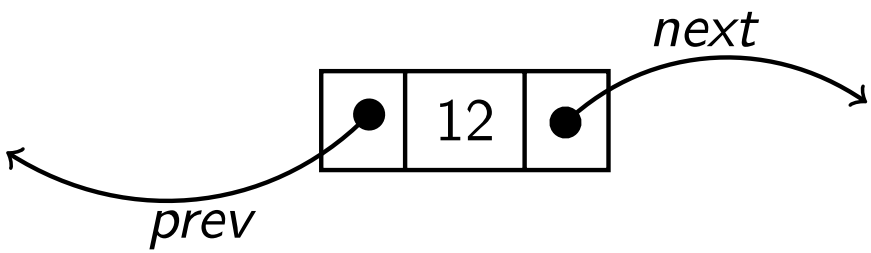
`Node<E,O>` implementa el interfaz `Position<E>`:

```
public interface Position<E> {
  public E element();
}
```

- Tiene 2 atributos:
  - final O **owner** – en qué lista esta el nodo
  - E **elem** – el contenido del nodo
  - El atributo `owner` es *final*, un nodo ya creado no se puede reutilizar en otra lista
- Métodos:
  - `n1.kinOf(n2)` indica si `n1` y `n2` son *parientes* (tiene el mismo *owner*)
  - `checkNode(Position<E> p)` comprueba si la posición `p` es *pariente* (tiene el mismo *owner*) de `this` (el nodo)
  - `setElement(E element)` asigna a `elem` el valor `element`

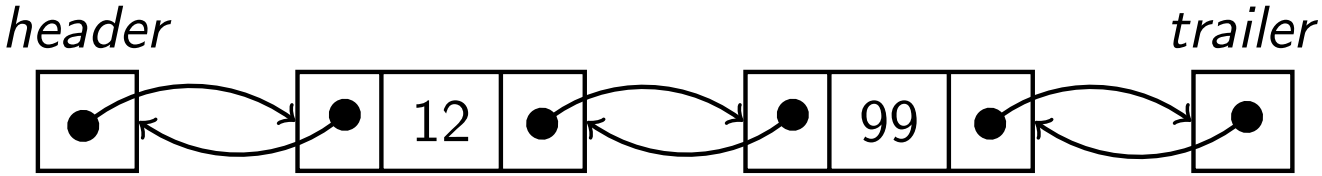
Implementación de `ListNode<E>`

- `ListNode<E>` extiende `Node<E,O>` para usarla en la implementación de `NodePositionList<E>`
  - Tiene 2 atributos mas: `prev`, `next`
  - Tiene *getter* y *setter* para los atributos `prev`, `next`, `elem`
  - `setPrev` y `setNext` pueden recibir `null`, ¿por qué?
- Un `ListNode<Integer>` (sin `owner`):



Implementación de `NodePositionList<E>`

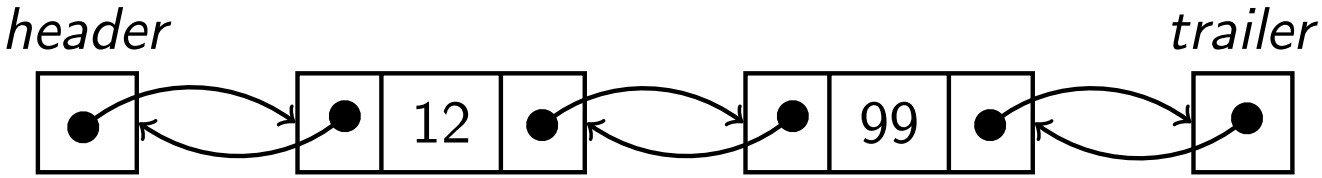
- Una lista de nodos doblemente enlazadas
- Tiene 3 atributos: el tamaño, y 2 nodos especiales: `header` y `trailer`
- Una lista con dos elementos 12 y 99:



- Tiene 3 constructores: (1) vacio, (2) un array, (3) una lista
- Limitaciones de tamaño
  - El tipo del atributo `size`
  - La memoria disponible
- El método privado `checkNode` comprueba si una posición `p` es realmente un nodo válido
  - No es `null`
  - Es de clase `ListNode<E>`
  - Es un nodo de la lista – usando `header.checkNode(p)`

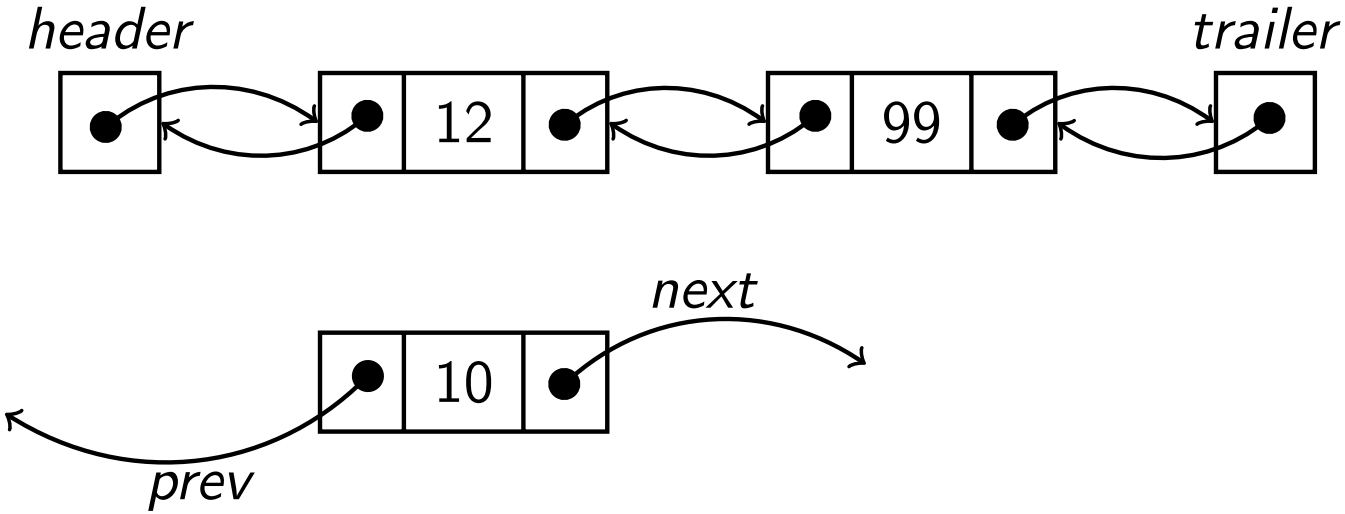
Ejemplo: l.addFirst(10)

- Al principio:



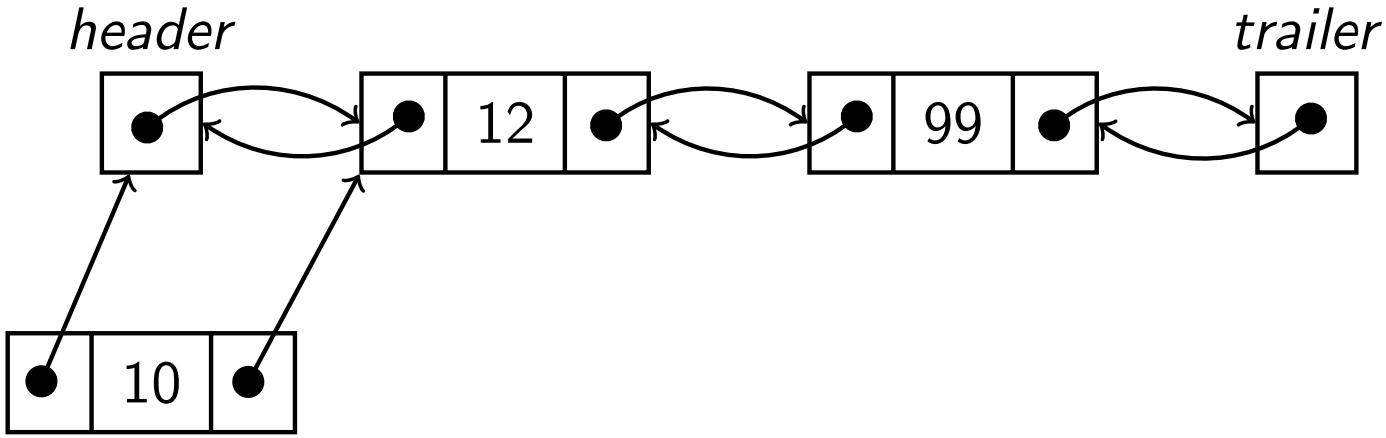
Ejemplo: l.addFirst(10)

- Creamos un nodo nuevo *n* con el elemento 10:



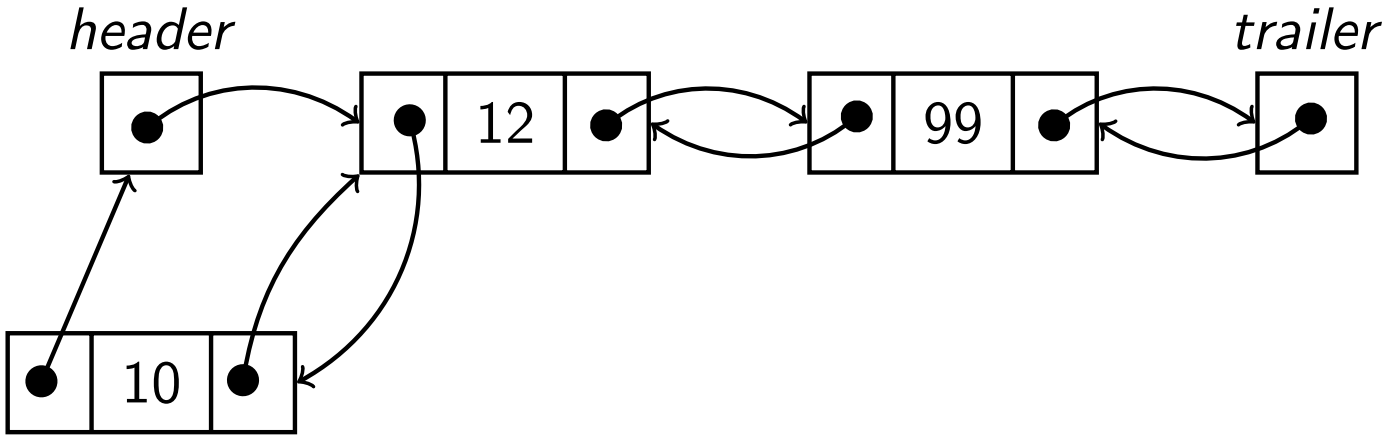
Ejemplo: l.addFirst(10)

- Asignamos los atributos `prev` y `next` de n:



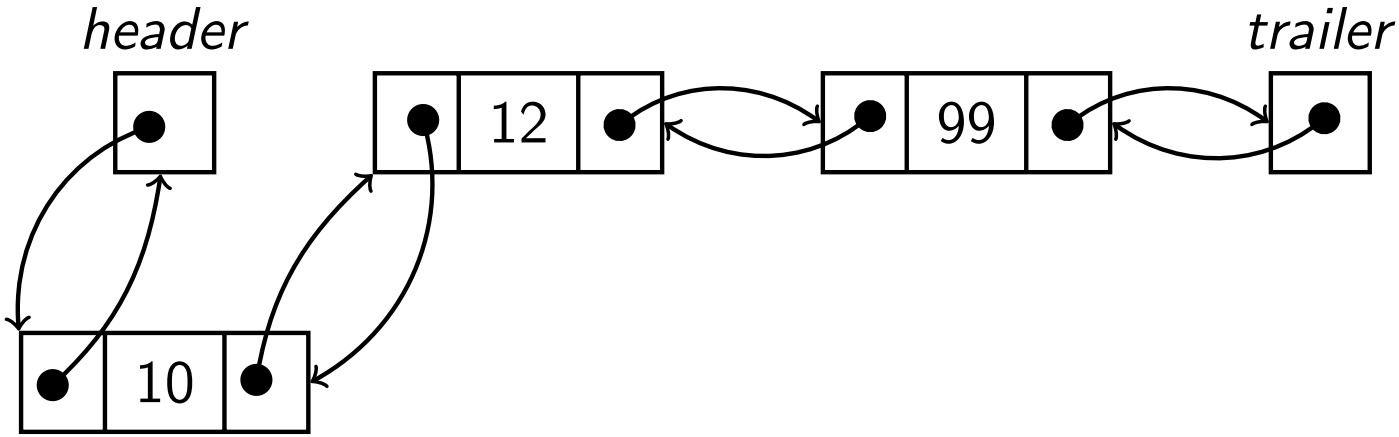
Ejemplo: l.addFirst(10)

- Asignamos el `prev` de `header.getNext()` a *n*:



Ejemplo: l.addFirst(10)

- Y dejamos que el `header` apunta a *n*:



Ejemplo: addFirst en código Java

```
public void addFirst(E elem) {

  ListNode<E> newNode = new ListNode<E>(this,
                                        header,
                                        elem,
                                        header.getNext());
  header.getNext().setPrev(newNode);
  header.setNext(newNode);
  size++;
}
```

Complejidad de `NodePositionList<E>`

`size()`	$O(1)$
`isEmpty()`	$O(1)$
`first()`	$O(1)$
`last()`	$O(1)$
`next(Position<E> p)`	$O(1)$
`prev(Position<E> p)`	$O(1)$
`addFirst(E elem)`	$O(1)$
`addLast(E elem)`	$O(1)$
`addBefore(Position<E> p, E elem)`	$O(1)$
`addAfter(Position<E> p, E elem)`	$O(1)$
`remove(Position<E> p)`	$O(1)$
`set(Position<E> p, E elem)`	$O(1)$
`toArray()`	$O(N)$

Comparación con `ArrayIndexedList<E>`

- Añadiendo elementos:

`addFirst(E)`	$O(1)$	`add(0,E)`	$O(N)$
`addLast(E)`	$O(1)$	`add(l.size(),E)`	$O(1)$ ó $O(N)$
`addBefore(P,E)`	$O(1)$	`add(i,E)`	$O(N)$

- Acceso secuencial:

`P.element()`	$O(1)$	`get(i)`	$O(1)$
`remove(P)`	$O(1)$	`removeElementAt(i)`	$O(N)$

- Acceso aleatorio:

`next(...),...,remove(P)`	$O(N)$	`removeElementAt(i)`	$O(N)$
`next(..),...`	$O(N)$	`get(i)`	$O(1)$

Comparación ArrayIndexedList y NodePositionList

Supongamos que todos los accesos a los elementos de de la lista son secuenciales: `get(0)`, `get(1)`, `...`, `get(i)`, `get(i+1)`, `...`

Pregunta

¿Aun así puede tener alguna ventaja usar ArrayIndexedList en vez de un NodePositionList?

Comparando implementaciónes

Pregunta

¿Cómo se puede averiguar qué implementación es más eficiente?

Comparando implementaciónes

Pregunta

¿Cómo se puede averiguar qué implementación es más eficiente?

Pregunta

¿Qué implementacion tiene mejor complejidad asintótica? (teórica)

Comparando implementaciónes

Pregunta

¿Cómo se puede averiguar qué implementación es más eficiente?

Pregunta

¿Qué implementacion tiene mejor complejidad asintótica? (teórica)

Solución

Hagamos experimentos: ¡vamos a medir!

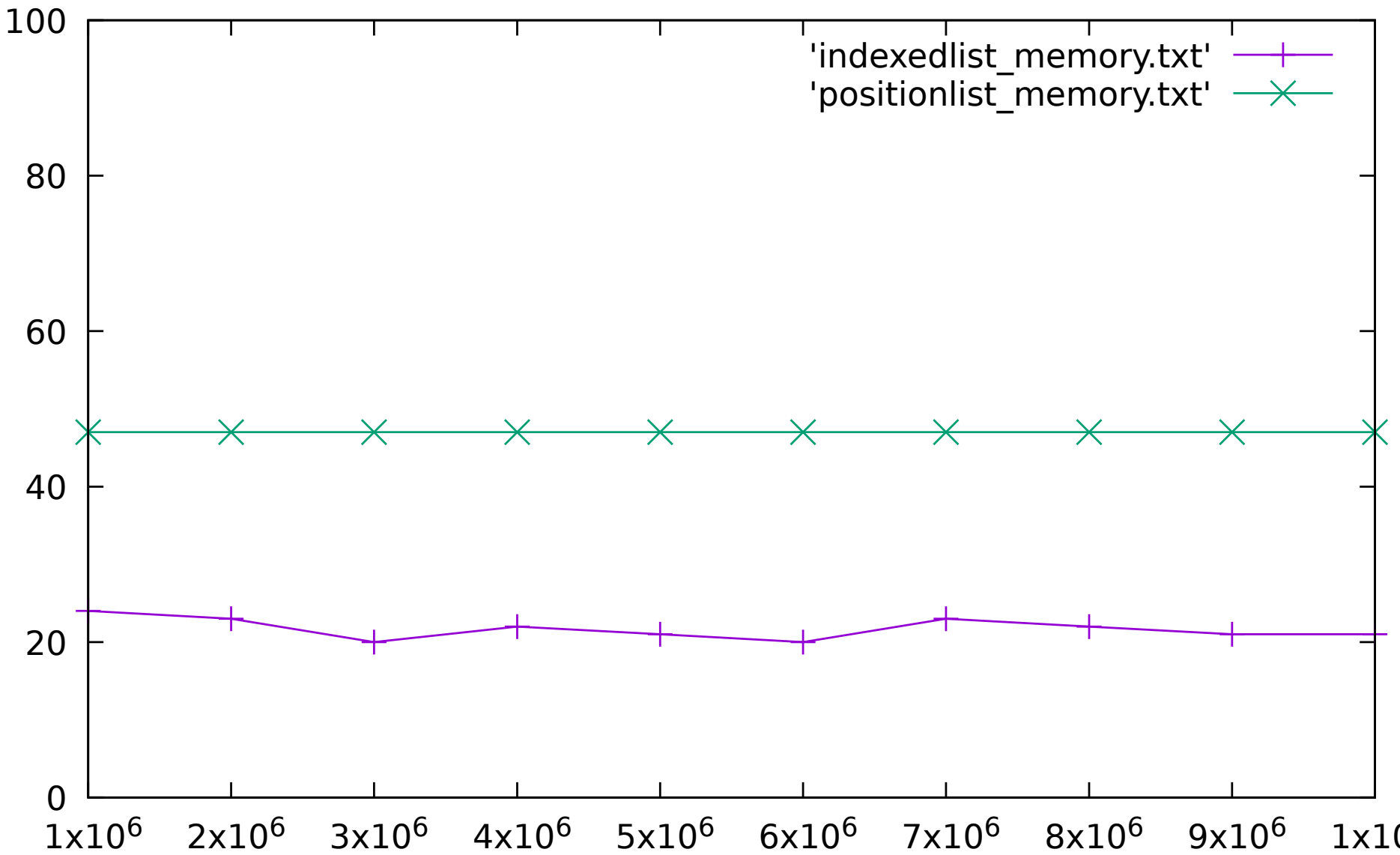
Código a comparar (`NodePositionList` vs `ArrayIndexedList`)

```
for (int i=0; i<n; i++)
  l.addLast(i);

Position<Integer> cursor = l.first();
while (cursor != null) {
  sum += cursor.element();
  cursor = l.next(cursor);
}
```

```
for (int i=0; i<n; i++)
  l.add(i,i);

for (int i=0; i<l.size(); i++)
  sum += l.get(i);
```

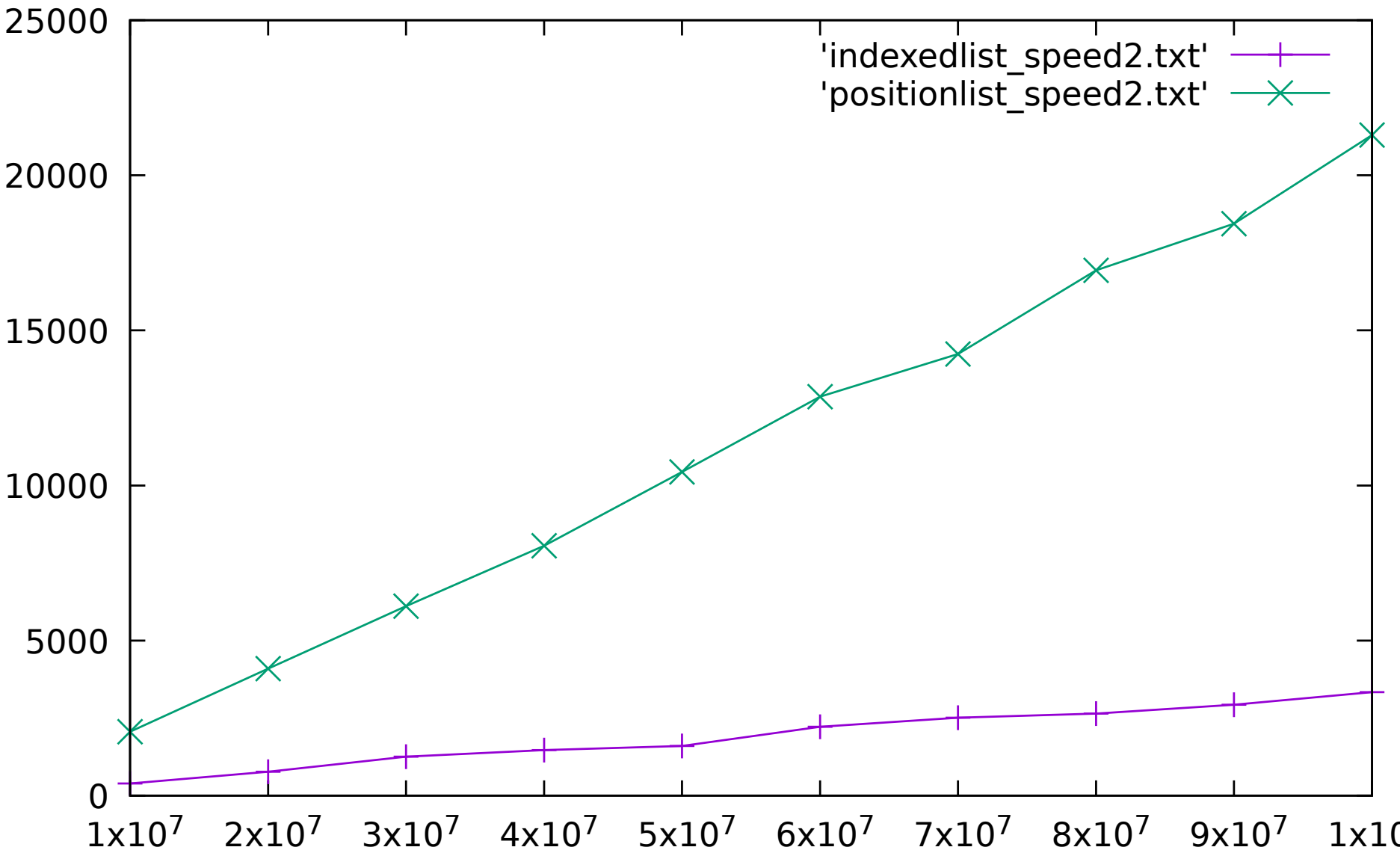
Uso de memoria (bytes por elemento)

Uso de memoria (bytes por elemento)



Tiempo (en milisegundos)

Tiempo (en milisegundos)



¿qué tarda más?: crear o recorrer una NodePositionList

- Crear la lista:

```
for (int i=0; i<n; i++)
  l.addLast(i);
```

- Recorrer la lista:

```
Position<Integer> cursor = l.first();
while (cursor != null) {
  sum += cursor.element();
  cursor = l.next(cursor);
}
```

¿qué tarda más?: crear o recorrer una ArrayIndexedList

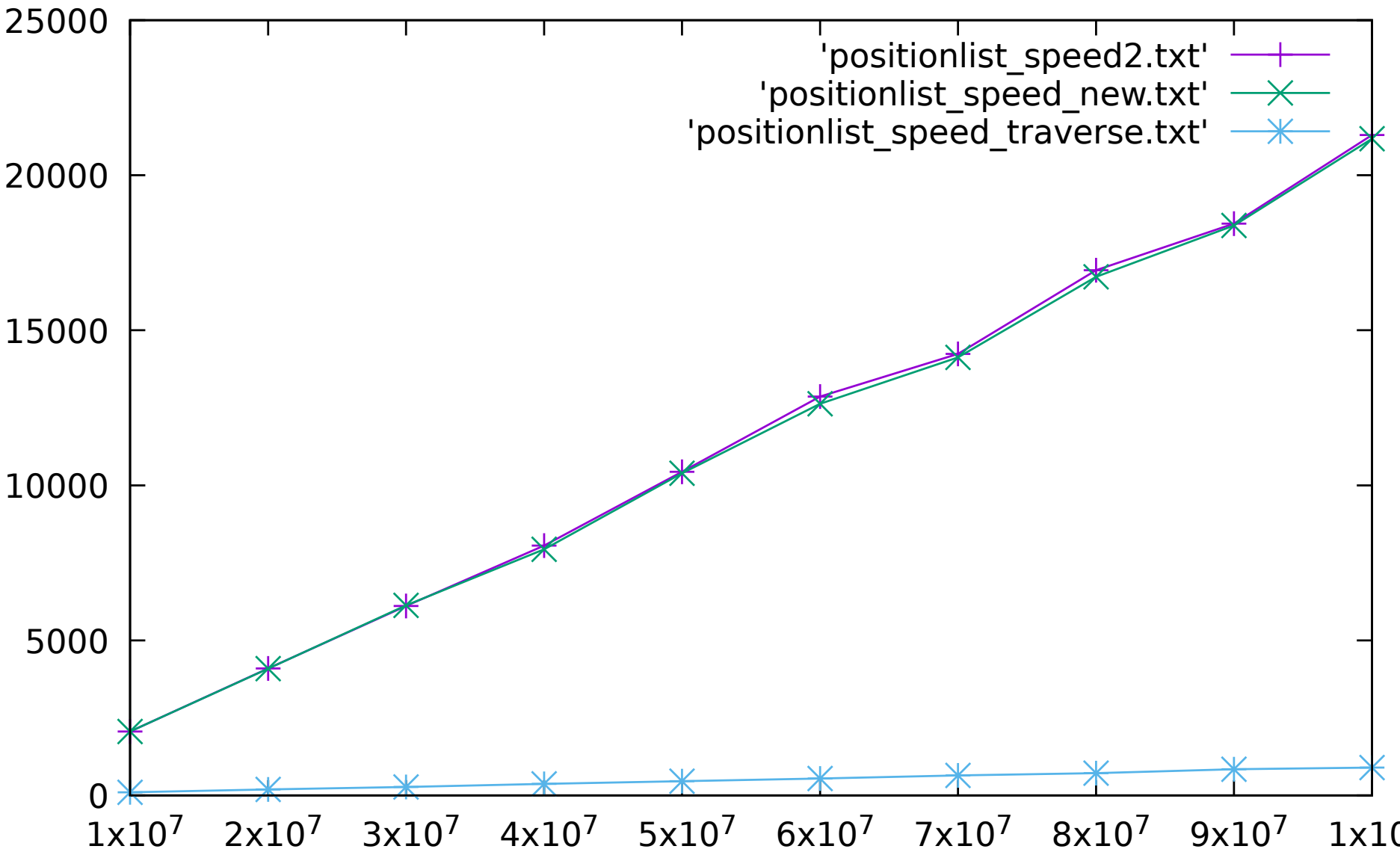
- Crear la lista:

```
for (int i=0; i<n; i++)
  l.add(i,i);
```

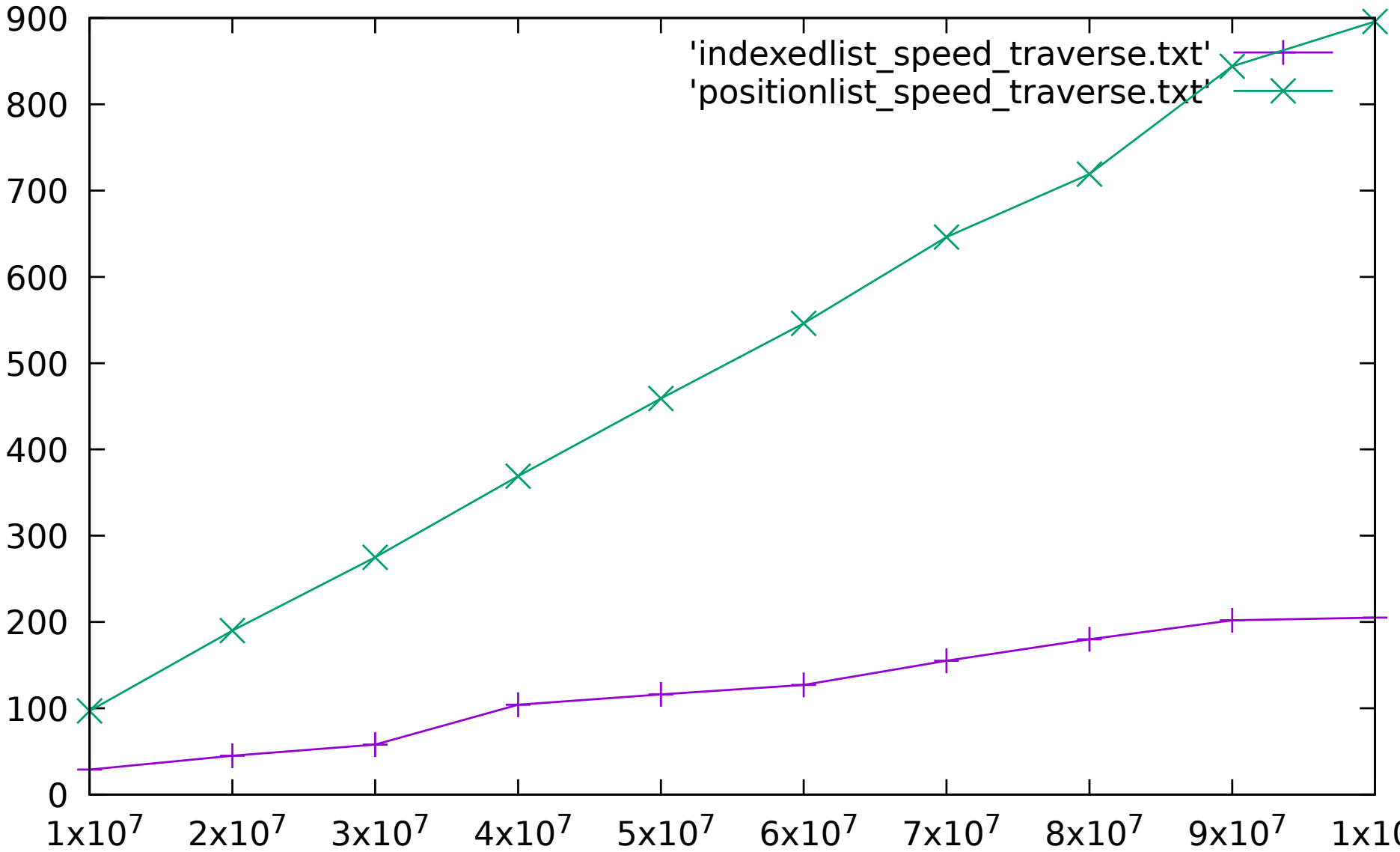
- Recorrer la lista:

```
for (int i=0; i<l.size(); i++)
  sum += l.get(i);
```

Domina el coste de `new` para PositionLists:



Recorrer la lista es más lento en PositionLists:



Conclusiones

- Pagamos un precio en memoria para usar `NodePositionList` comparado con `ArrayIndexedList`: 40 bytes en vez de 20 (por elemento).
- Domina el tiempo para crear la lista en ambas implementaciones (comparado con recorrer la lista).
- Es bastante más lento `NodePositionList` comparado con `ArrayIndexedList`. Ambas tienen complejidad lineal pero con constantes diferentes:

implementation	complexity function
`ArrayIndexedList`	$f(x) = 170 + 3.17 * 10^{-5} * x$
`NodePositionList`	$f(x) = -169 + 2.11 * 10^{-4} * x$

- Constante lineal positionlist / Constante lineal indexedlist = 6.6.

Ejercicio

Ejercicio

Dado un elemento, si no se tiene su posición entonces habría que buscarla. Añadir a la clase e implementar el método:

```
public Position<E> getPos(E e)
```

que debe devolver la primera posición en la lista que contiene el elemento `e`, o `null` si el elemento no está en la lista

Pregunta

¿Cuál sería la complejidad en el caso peor de dicho método?

Ejercicio

Practica re-implementando métodos: addAfter, addLast, remove, etc

Ejercicios (no muy fáciles)

Ejercicio

Implementar un método

`PositionList<E> l.splitList(Position<E> pos)`

que corta la lista `l` en dos, dejando los nodos hasta `pos` en `l` y copiando los nodos desde `pos` en adelante en una lista nueva

Ejercicio

Implementar un método

`l1.concatList(PositionList<E> l2)`

que añade la lista `l2` al final de la lista `l1`

Ejercicio

¿Cuales son las complejidades de los métodos?