Qu'est-ce qu'une position ?

Qu'est-ce qu'une liste positionnelle ?

Interfaces `Position` et `PositionalList`

Implémentation de `LinkedPositionalList`

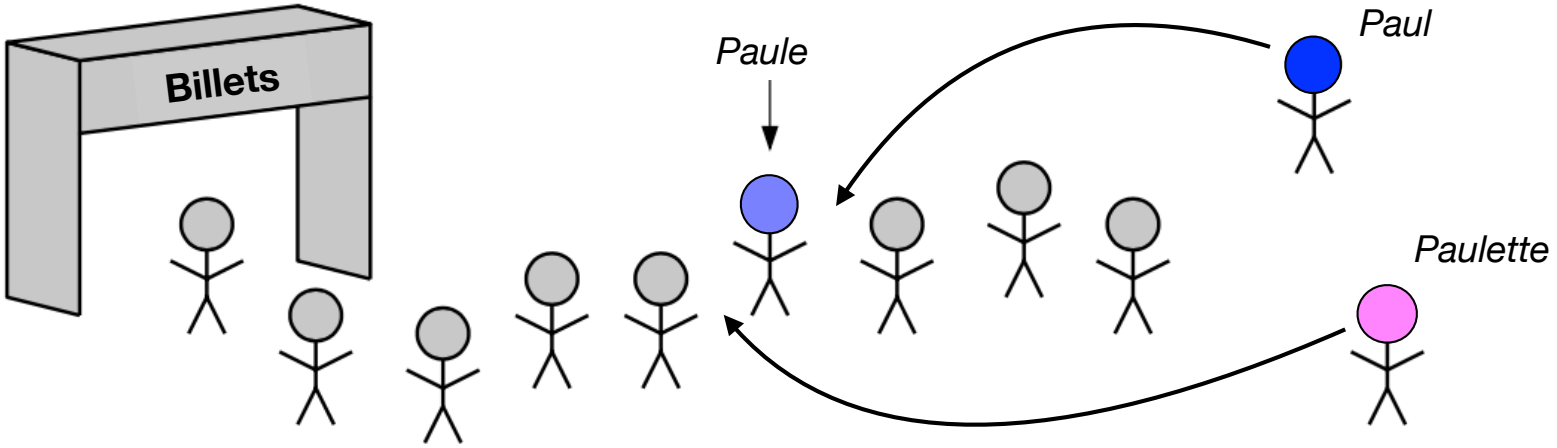
Trier une liste positionnelle

Les itérateurs d'éléments et de positions d'une liste positionnelle

Itérateur avec la méthode optionnelle `remove`

Gérer les fréquences d'accès aux éléments (`FavoritesList`)

Abstraction d'une `Position`




```
int paulePosition = maListe.indexOf( "Paule" );
```

	P	D	T	Q	C	Paule	S	H	N	Paul	Paulette
	0	1	2	3	4	**5**	6	7	8	9	10

```
maListe.add( paulePosition + 1, maListe.remove( "Paul" ) );
```

	P	D	T	Q	C	Paule	Paul	S	H	N	Paulette
	0	1	2	3	4	5	6	7	8	9	10

```
maListe.add( paulePosition, maListe.remove( "Paulette" ) );
```

	P	D	T	Q	C	Paulette	Paule	Paul	S	H	N
	0	1	2	3	4	5	6	7	8	9	10

L'idée d'une **`Position`** est d'être capable d'accéder à un élément dans une structure sans avoir à le rechercher (de nouveau) une fois qu'on l'a ! **Ceci est présentement impossible pour une liste chaînée** ! Et si on enlèvevait la méthode `indexOf` de `List` ?

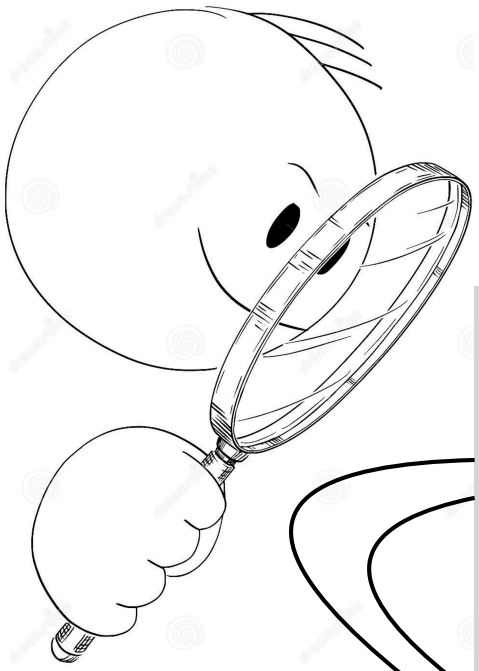
ADT `PositionalList`

- Une liste positionnelle stocke une collection d'éléments qui sont "interfacés" par des "***positions***"
- L'objet/abstraction **`Position`** ne supporte qu'une seule méthode pour accéder à son élément :
  - `E getElement()` qui retourne l'élément qui y est stockée
- Une **`Position`** permet d'accéder à un élément (avec un pointeur/référence/adresse, indice) dans la structure de données. Une **`Position`** peut donc être implémentée de différentes manières.
- *Le hic avec l'exemple est que si on avait ajouté Paulette avant Paul, le code ne marche pas à moins de "redemander" la position de Paule. On aurait eu les 3 amis dans le mauvais ordre: Paulette, Paul, Paule.*
- Une **`Position`** *p* associée à un élément *e* dans une liste ne change pas même si ce dernier se déplace dans la liste suite à des insertions/suppressions effectuées (sur ce dernier ou d'autres éléments dans la liste). Par exemple, la **`Position`** pour Paule dans l'exemple permettrait de lui accéder en tout temps, même après les insertions de Paul et Paulette et peu importe l'ordre des insertions on aurait obtenu le même résultat, soit Paulette avant Paule avant Paul. La **`Position`** de Paule aurait passé à 6 après l'insertion de Paulette.
- Même si on change l'élément *e*, *p* ne change pas ! C'est donc comme un indice dynamique et on ne peut pas l'implémenter avec un simple `int` représentant un indice de tableau !
- La seule manière de rendre une **Position** invalide est de retirer de la liste l'élément qui lui est associé.

Voir le code...

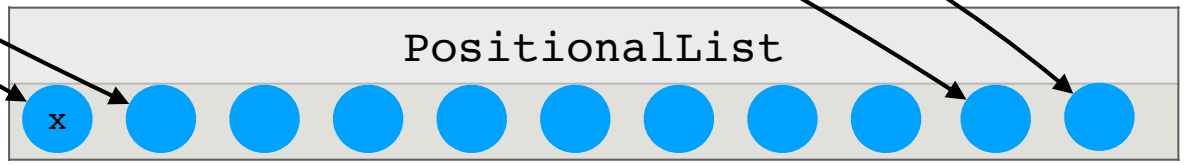
`Position.java`
`PositionalList.java`

L'utilisateur/programmeur d'application voit la liste à travers une interface de **`Position`** ; toutes les méthodes de mise à jour prennent une **`Position`** en argument



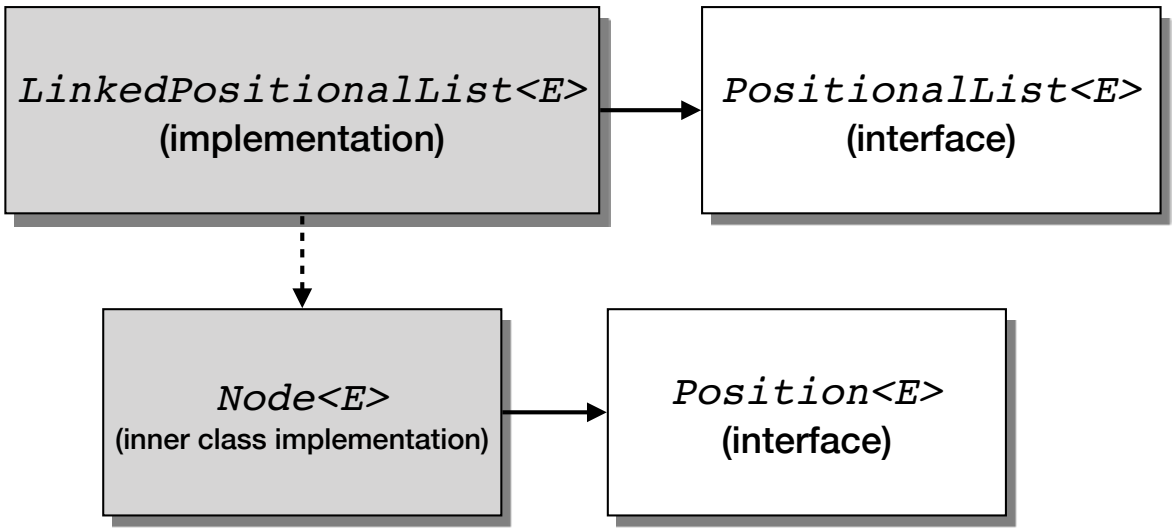
```
interface Position :
 E getElement();

Position p = l.addFirst( x );
Position q = l.after( p );
Position r = l.last();
Position s = l.before( r );
```

Operation	Return Value	PositionalList
addLast(8)	p	[8p]
first()	p	[8p]
addAfter(p, 5)	q	[8p,5q]
before(q)	p	[8p,5q]
addBefore(q, 3)	r	[8p,3r,5q]
r.getElement()	3	[8p,3r,5q]
after(p)	r	[8p,3r,5q]
before(p)	null	[8p,3r,5q]
addFirst(9)	s	[9s,8p,3r,5q]
remove(last())	5	[9s,8p,3r]
set(p, 7)	8	[9s,7p,3r]
remove(q)	"error"	[9s,7p,3r]



LinkedPositionalList

(Implémentation de l'ADT *PositionalList* avec une liste doublement chaînée)

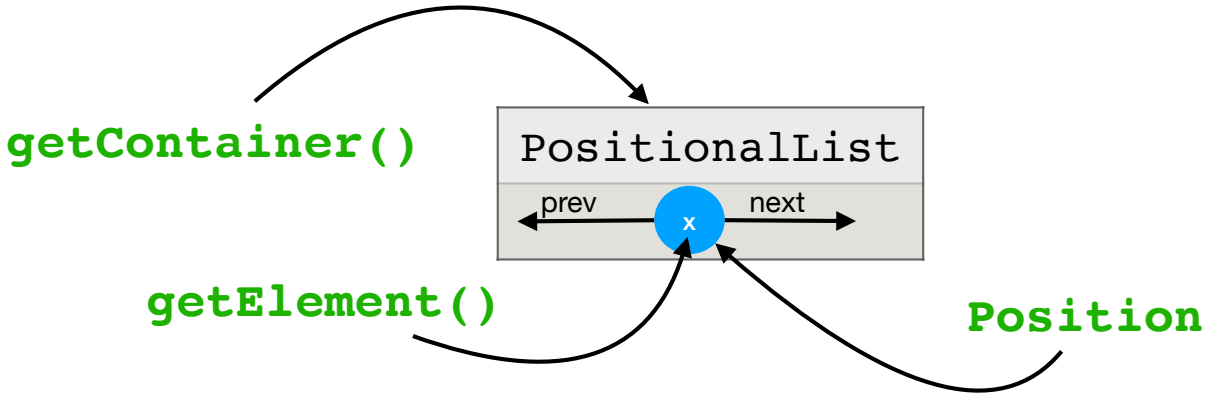


👉 La classe `Node` est imbriquée dans la classe `LinkedPositionalList`
La classe `Node` implémente `Position`

Node<E> implements Position<E>

Voir le code...

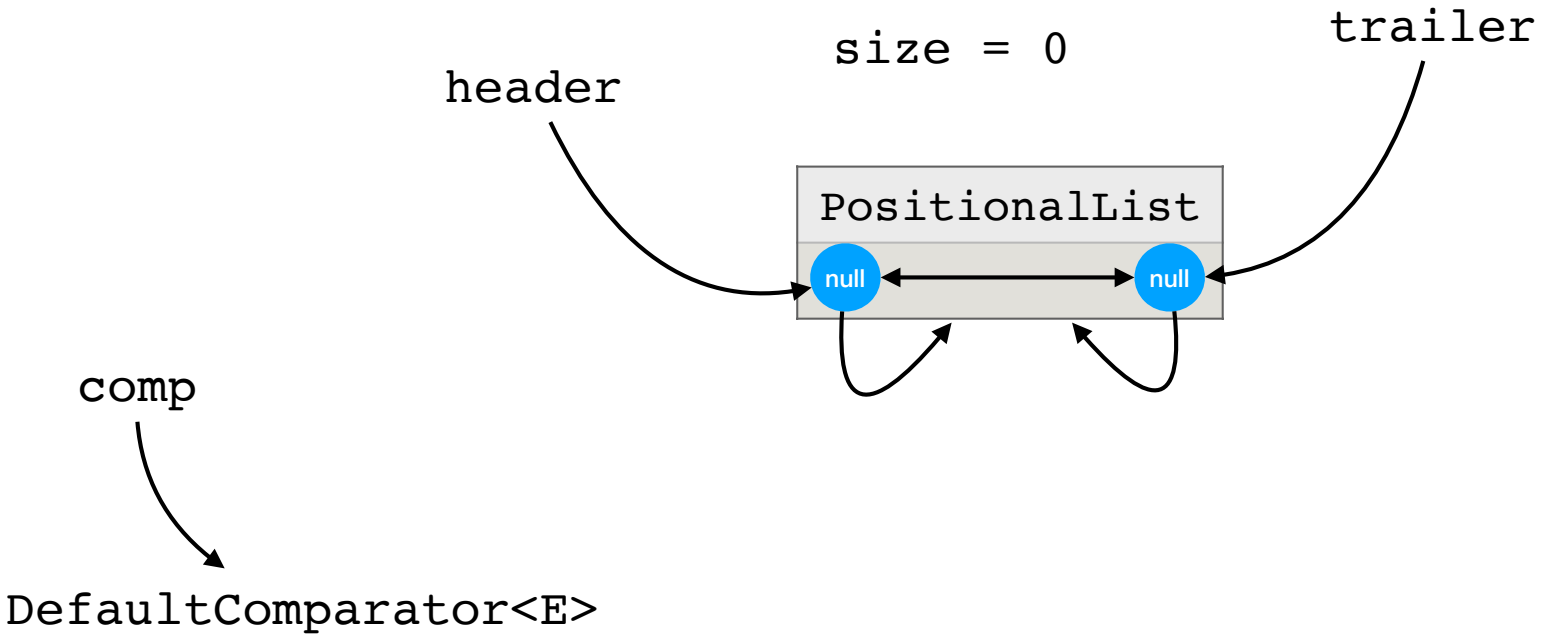
LinkedPositionalList.java



Attributs et état initial

Voir le code...

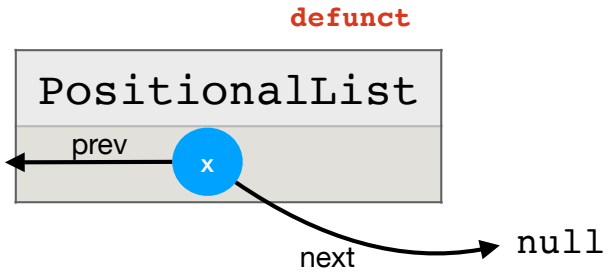
`LinkedPositionalList.java`

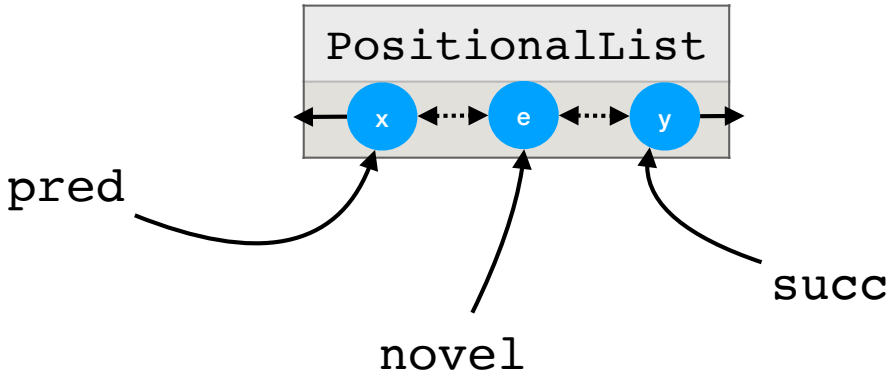


"defunct" `Position` et méthode `addBetween`

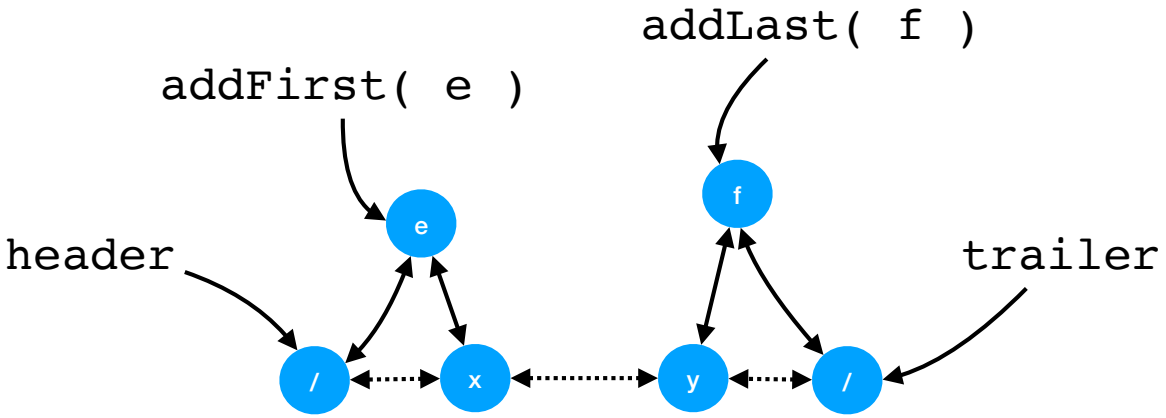
Voir le code...

`LinkedPositionalList.java`



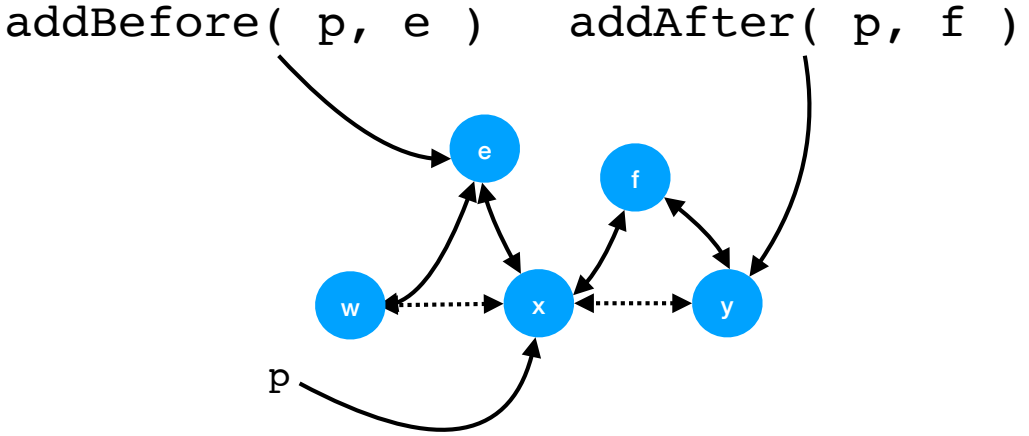


Les multiples méthodes add

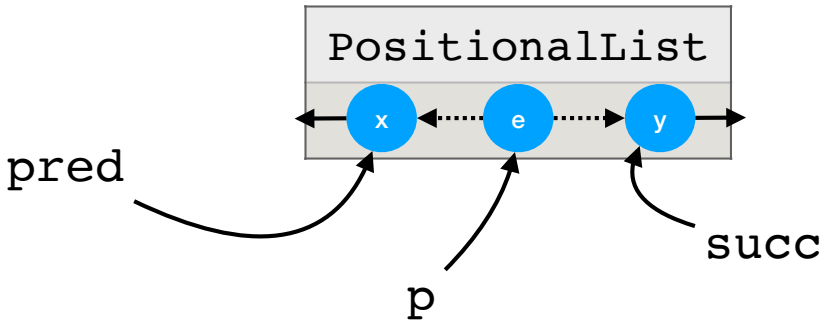


Voir le code...

LinkedPositionalList.java

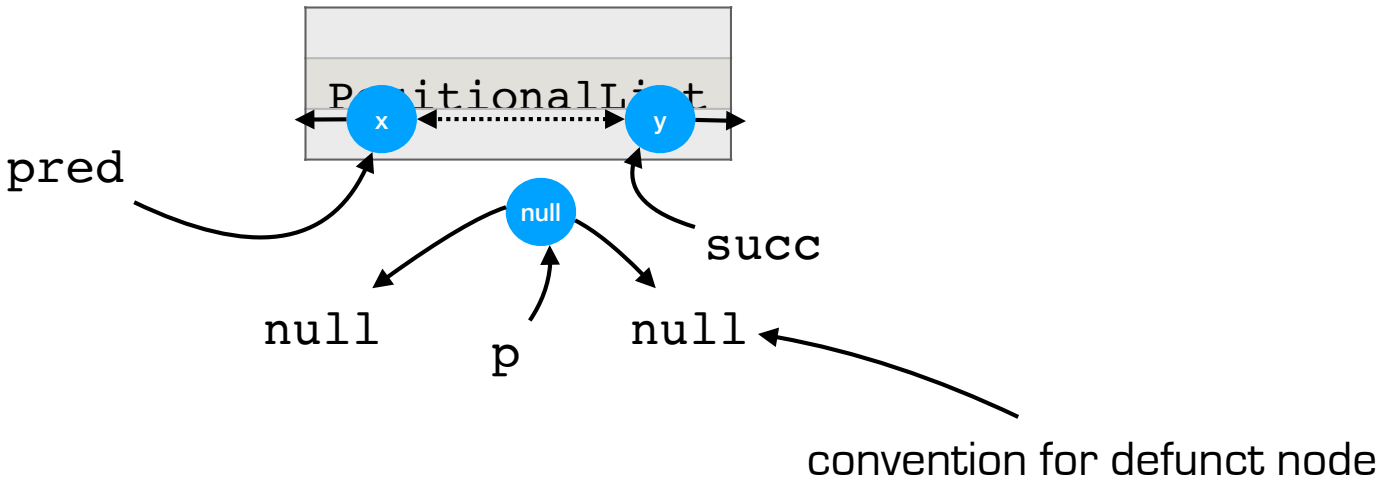


remove(p)

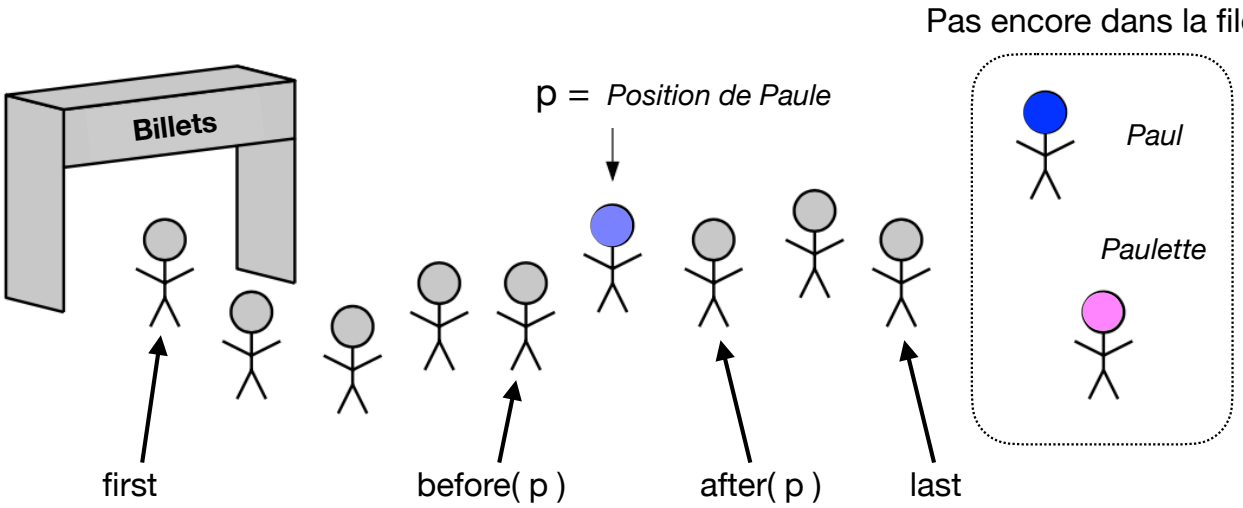


Voir le code...

LinkedPositionalList.java

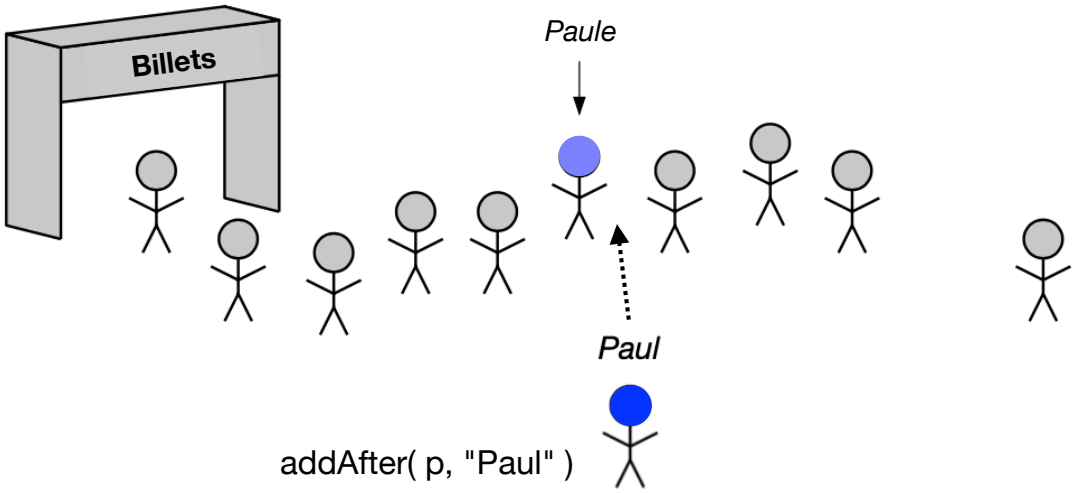


`Paul` rejoint `Paule` dans la liste

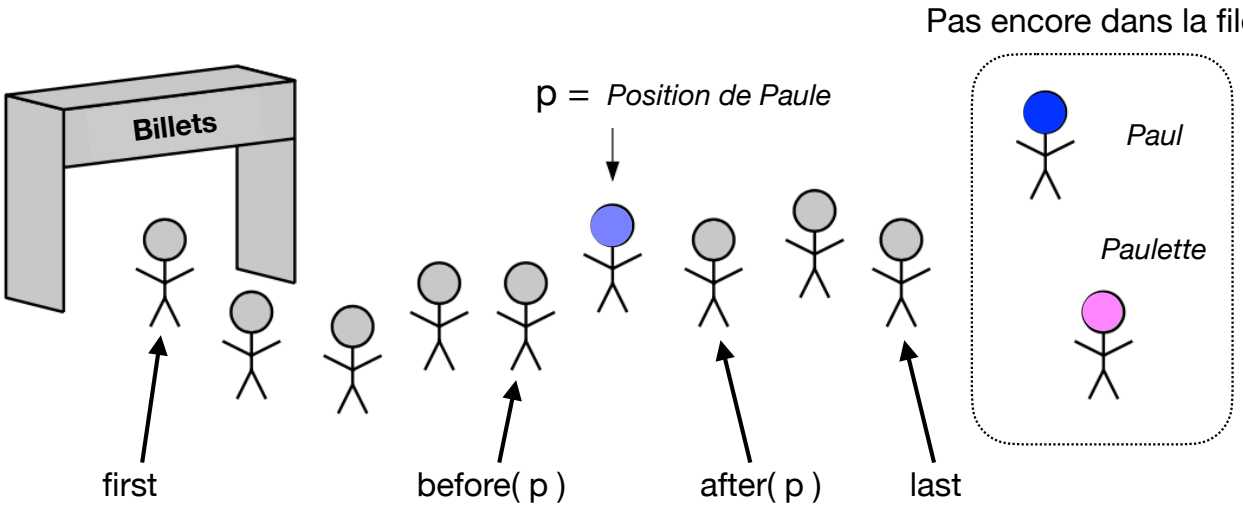


Voir le code...

`TestPosition.java`

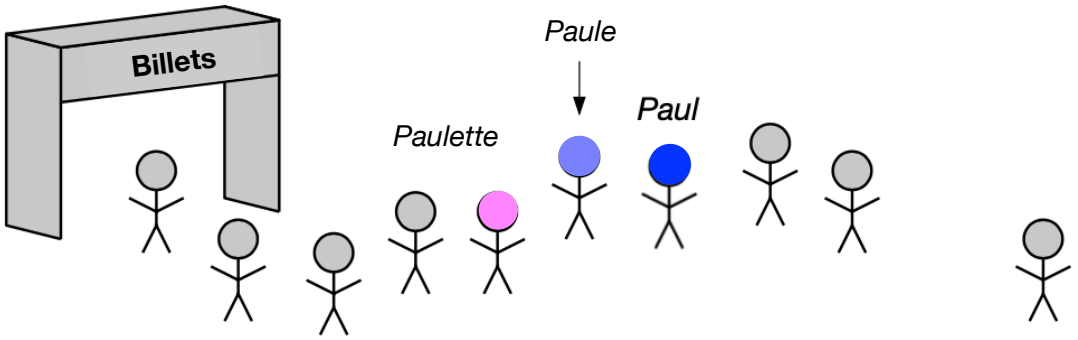


`Paul` et `Paulette` rejoignent `Paule` dans la liste



Voir le code...

`TestPosition.java`



addBefore(p, "Paulette")

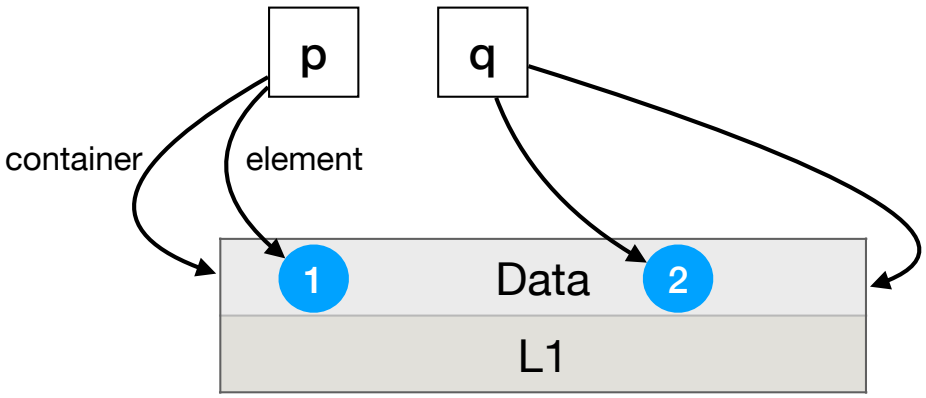
Exercice : Que se passe-t-il avec le code suivant ?

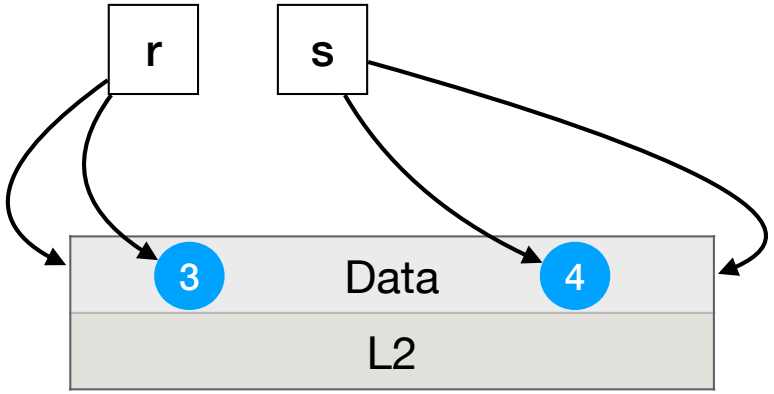
```
// mixing two positional lists
PositionalList<Integer> L1 = new LinkedPositionalList<>();
PositionalList<Integer> L2 = new LinkedPositionalList<>();
p = L1.addLast( 1 );
q = L1.addLast( 2 );
r = L2.addLast( 3 );
s = L2.addLast( 4 );
System.out.println( "L1: " + L1 );
System.out.println( "L2: " + L2 );
L1.remove( r );
System.out.println( "L1: " + L1 );
System.out.println( "L2: " + L2 );
```

Exercice : Que se passe-t-il avec le code suivant ?

```
// mixing two positional lists
PositionalList<Integer> L1 = new LinkedPositionalList<>();
PositionalList<Integer> L2 = new LinkedPositionalList<>();
p = L1.addLast( 1 );
q = L1.addLast( 2 );
r = L2.addLast( 3 );
s = L2.addLast( 4 );
System.out.println( "L1: " + L1 );
System.out.println( "L2: " + L2 );
L1.remove( r );
System.out.println( "L1: " + L1 );
System.out.println( "L2: " + L2 );
```

On a introduit la notion du "*container*" d'une `Position`, qui permet de détecter que `r` n'appartient pas à `L1`





On attrape l'erreur et on lance une exception pour "Invalid container"

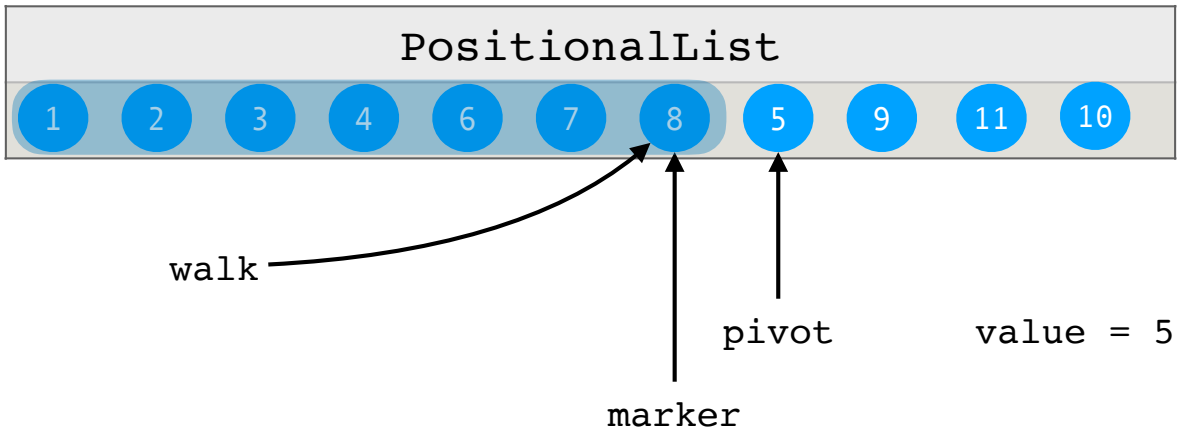
```
// mixing two positional lists
PositionalList<Integer> L1 = new LinkedPositionalList<>();
PositionalList<Integer> L2 = new LinkedPositionalList<>();
p = L1.addLast( 1 );
q = L1.addLast( 2 );
r = L2.addLast( 3 );
s = L2.addLast( 4 );
System.out.println( "L1: " + L1 );
System.out.println( "L2: " + L2 );
L1.remove( r );
System.out.println( "L1: " + L1 );
System.out.println( "L2: " + L2 );
```

```
[java] L1: [1,2]
[java] L2: [3,4]
[java] Exception in thread "main" java.lang.IllegalArgumentException: Invalid container
[java]  at ca.umontreal.adt.list.LinkedPositionalList.validate(LinkedPositionalList.java:58)
[java]  at ca.umontreal.adt.list.LinkedPositionalList.remove(LinkedPositionalList.java:168)
[java]  at ca.umontreal.adt.list.ListApp.main(ListApp.java:101)
[java] Java Result: 1
```

Trier un liste positionnelle

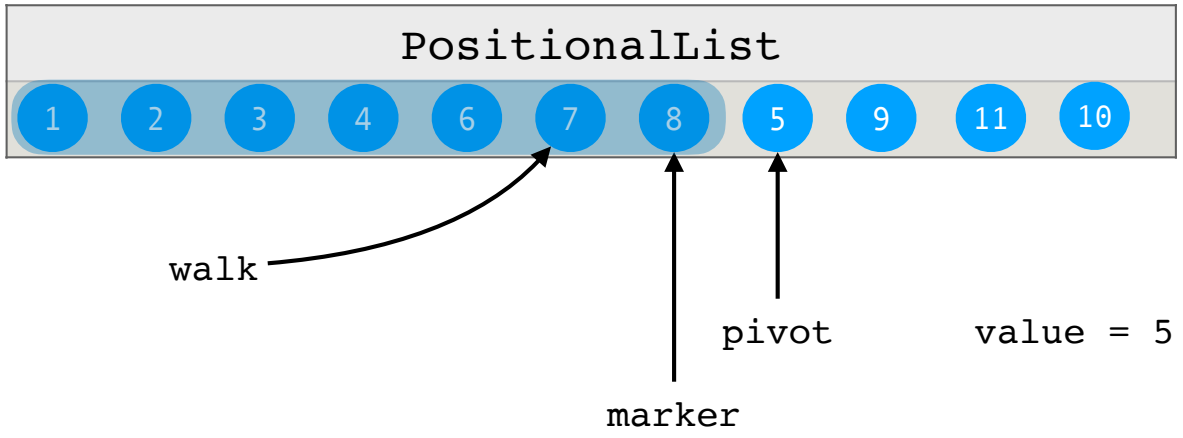
Nous développons une implémentation d'un tri par insertion qui fonctionne sur une `PositionalList`, où chaque élément est placé par rapport à une collection de nombres croissants d'éléments précédemment triés.

Nous prenons une variable, `marker`, qui représente la position la plus à droite de la partie actuellement triée de la liste. À chaque itération, nous considérons la position juste après le marqueur comme le pivot et nous considérons où doit être inséré l'élément du pivot par rapport à la partie déjà triée ; nous utilisons une autre variable, `walk`, pour se déplacer vers la gauche à partir du marqueur, tant qu'on trouve un élément précédent avec une valeur supérieure à celle du pivot.



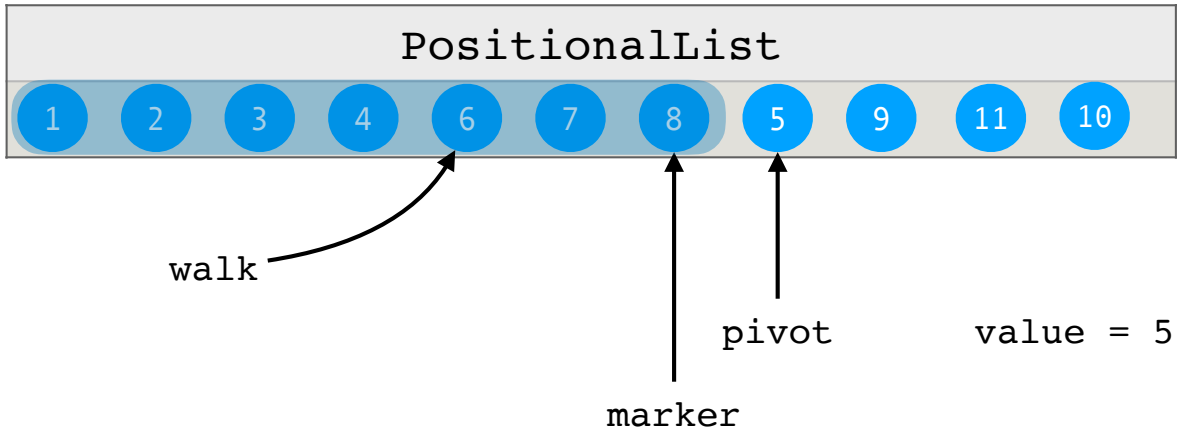
Trier un liste positionnelle

on recule `walk` jusqu'à ce que la **valeur avant** `walk` soit < `value`



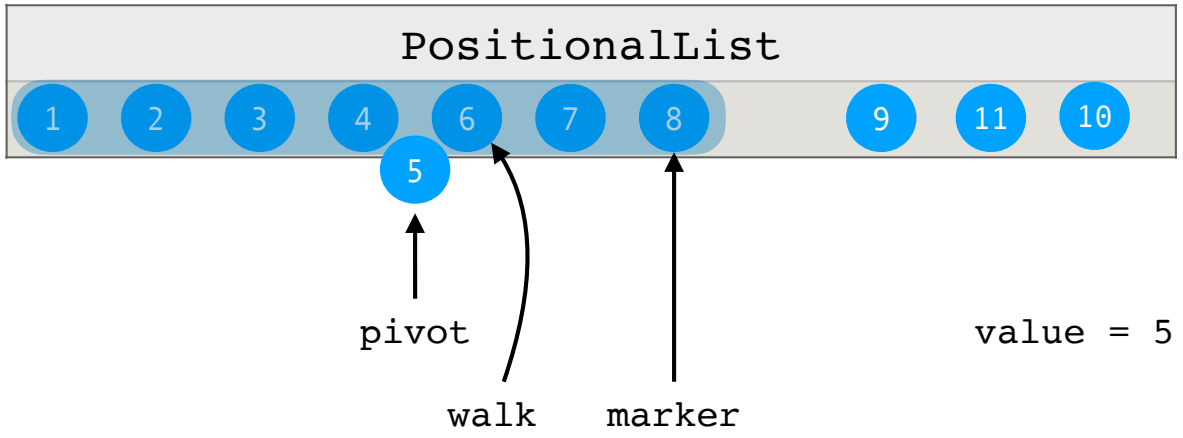
Trier un liste positionnelle

on recule `walk` jusqu'à ce que la **valeur avant** `walk` soit < `value`



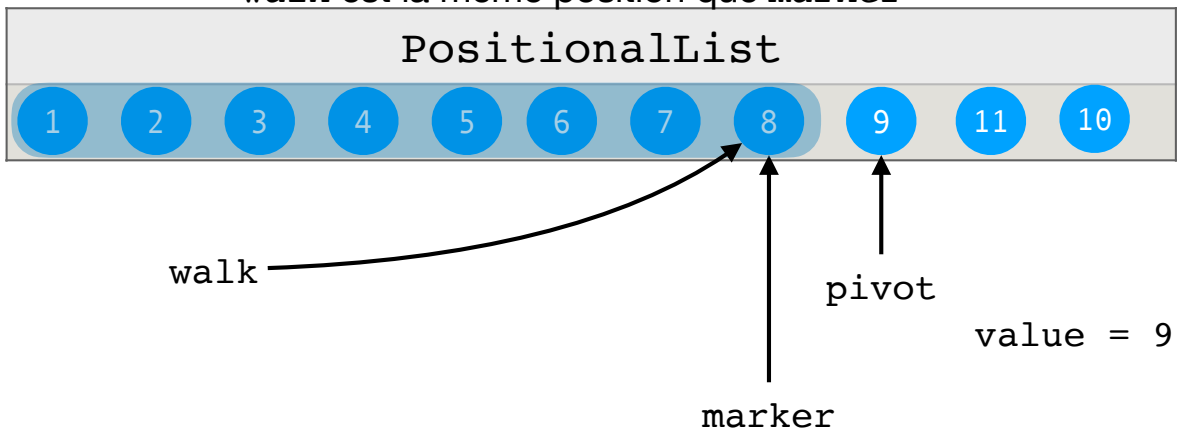
Trier un liste positionnelle

on arrête car la valeur 4 est < que la valeur 5;
dans ce cas, on déplace la position `pivot` devant `walk`;



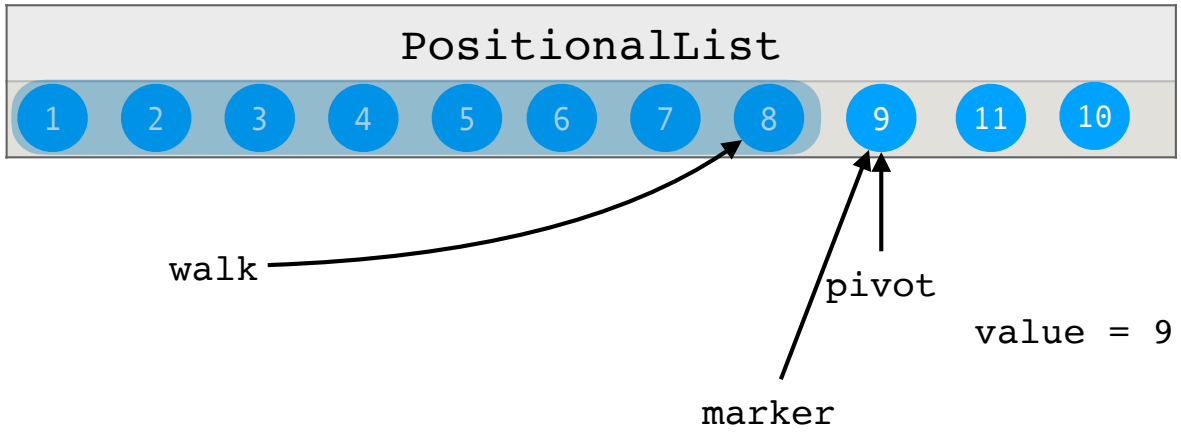
Trier un liste positionnelle

ça remet la liste dans l'état où `marker` est la position du dernier élément trié
`pivot` est la position après `marker`
`walk` est la même position que `marker`



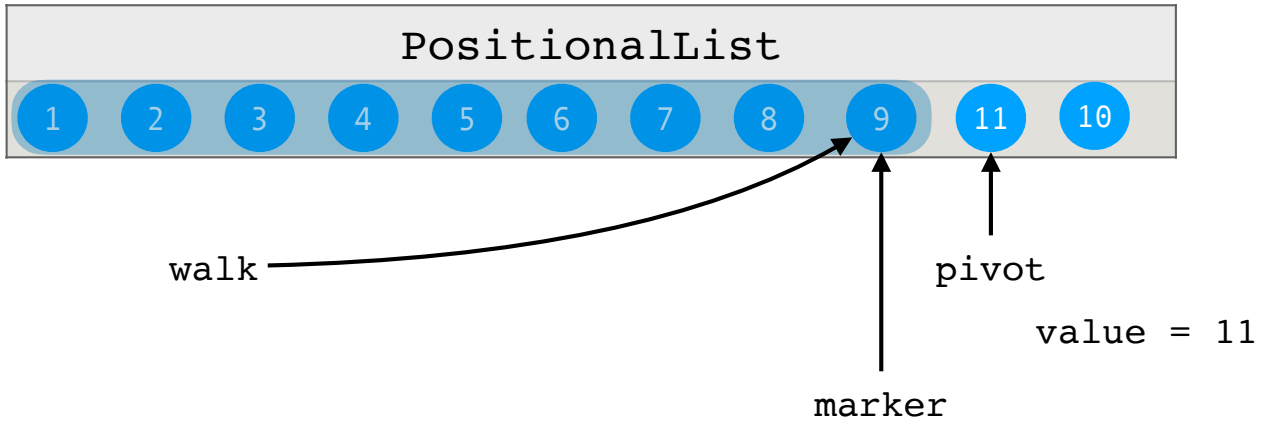
Trier un liste positionnelle

lorsque `value` est > que la valeur à la position `marker`, on avance `marker` à `pivot`



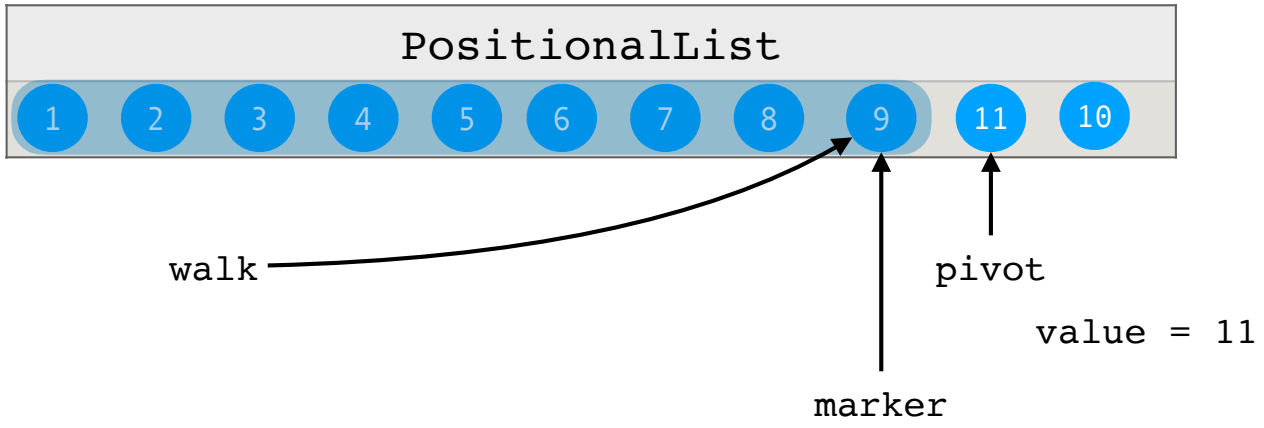
Trier un liste positionnelle

on revient dans la première boucle
`pivot` devient la position après `marker`



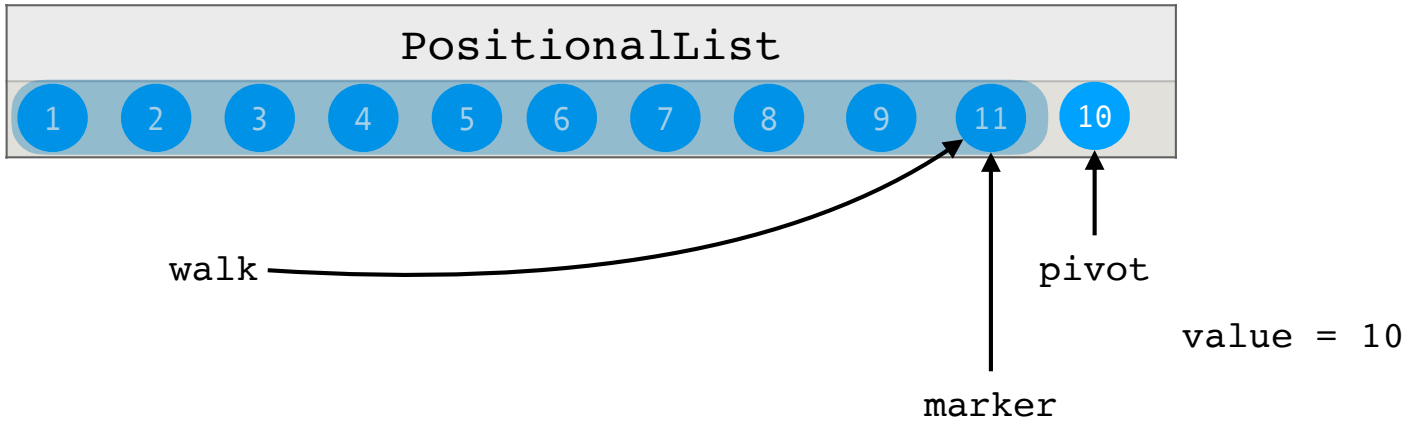
Trier un liste positionnelle

même scénario : `value` > la valeur à la position `marker`
on avance `marker` à `pivot`



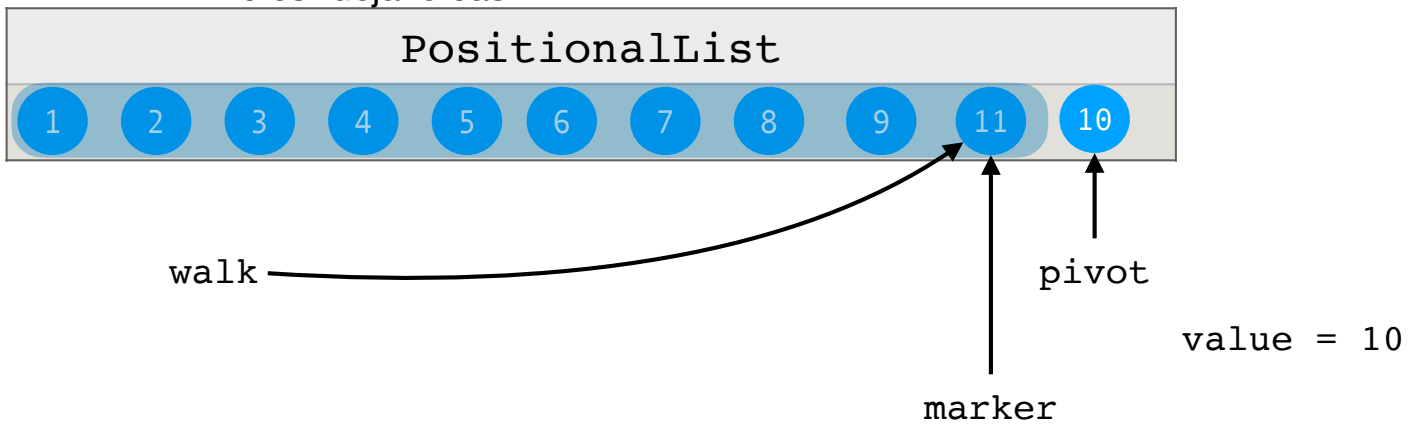
Trier un liste positionnelle

même scénario, `value` > valeur à la position `marker`
on avance `marker` à `pivot`

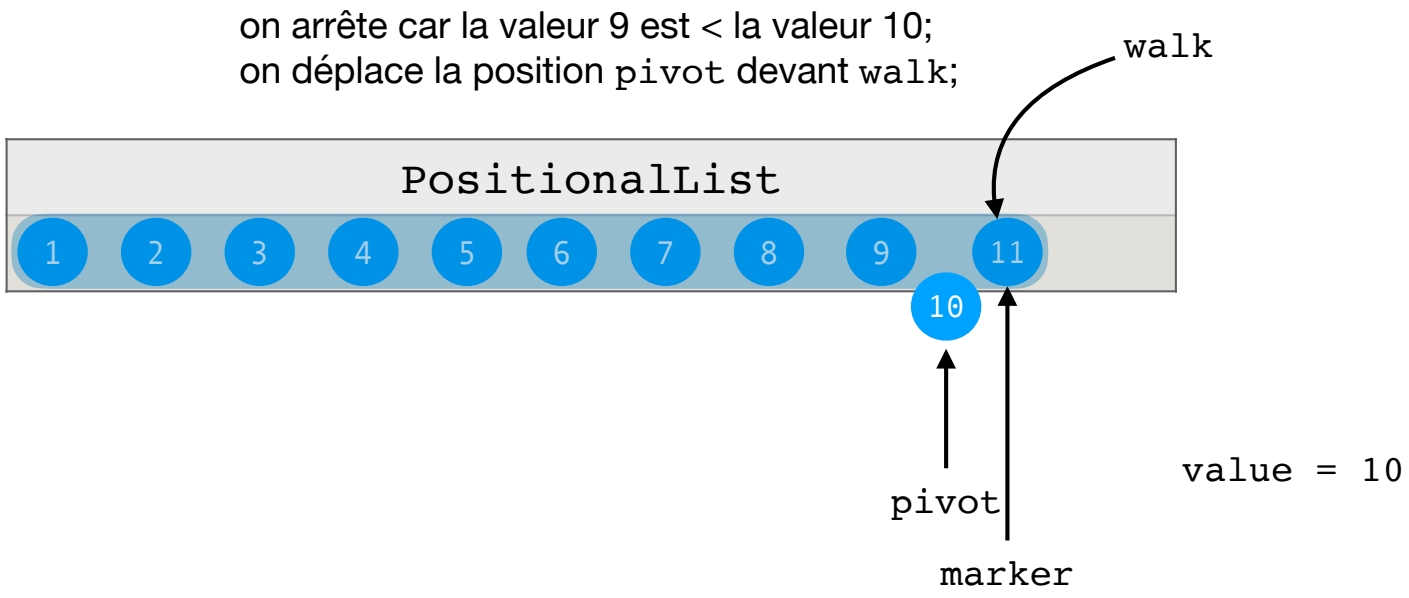


Trier un liste positionnelle

`value` < valeur à la position `marker`
on recule `walk` jusqu'à ce que la valeur avant `walk` soit < `value`
c'est déjà le cas !

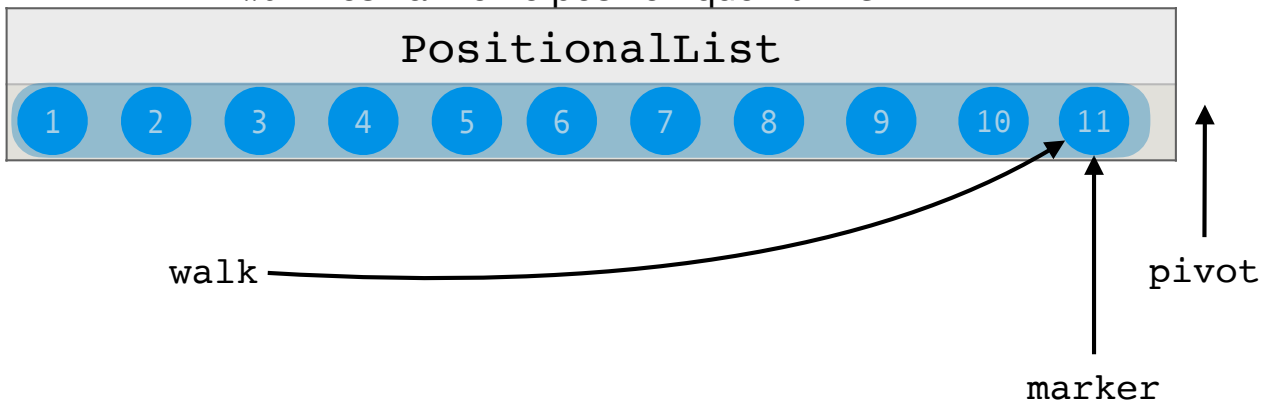


Trier un liste positionnelle



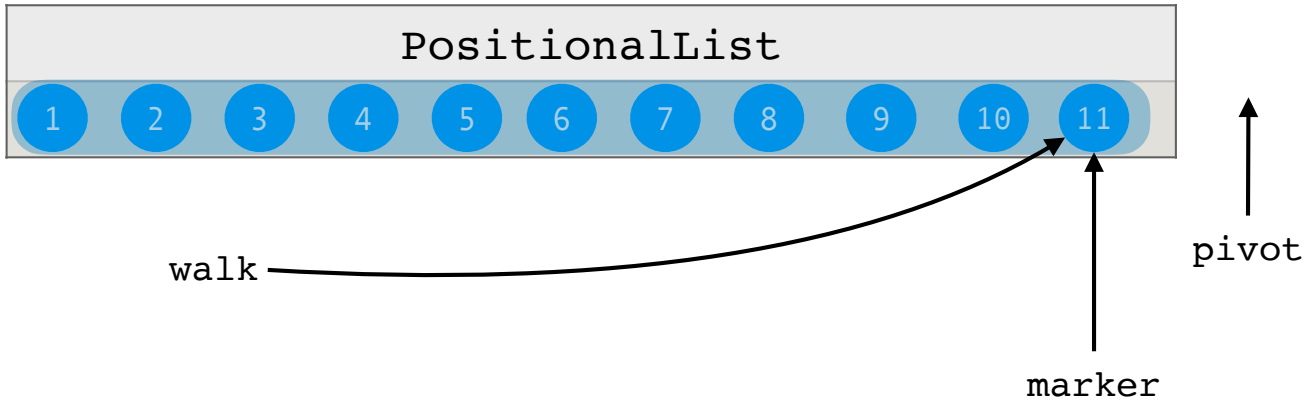
Trier un liste positionnelle

ça remet la liste dans l'état où `marker` est la position du dernier élément trié
`pivot` est la position après `marker`
`walk` est la même position que `marker`



Trier un liste positionnelle

`marker == list.last()` => on sort de la boucle externe;
on termine l'algorithme !



👉 `marker` devient la position du dernier élément lorsque tous les éléments de la liste sont bien placés !

Voir le code de la méthode `sort` ...

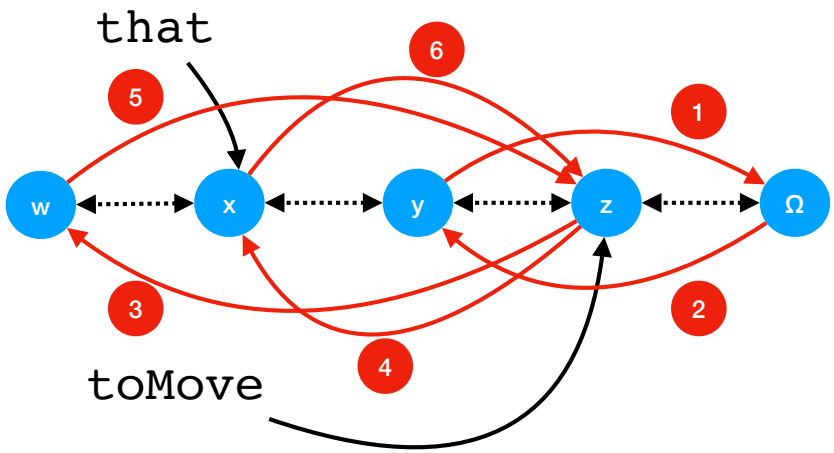
`LinkedPositionalList.java`

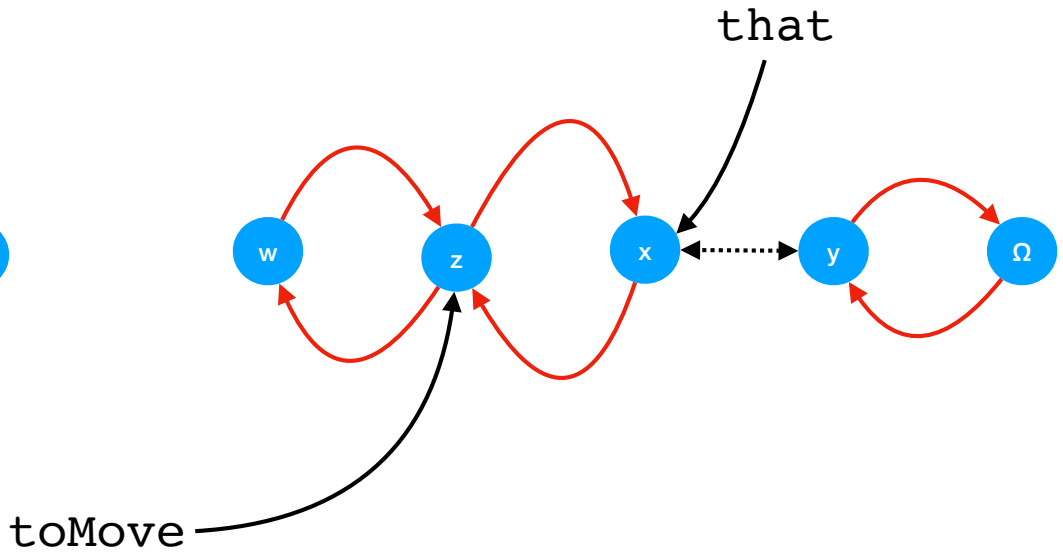
moveBefore(that, toMove)

```
// remove toMove by relinking its prev and next nodes
1  toMove.prev.next = toMove.next
2  toMove.next.prev = toMove.prev

// move toMove : 1) adjust toMove prev and next
3  toMove.prev = that.prev
4  toMove.next = that.next

// move toMove: 2) relink that node
5  that.prev.next = toMove
6  that.prev = toMove
```

Voir le code...

LinkedPositionalList.java





Pour rendre un ADT itérable, il suffit d'étendre avec `Iterable` (ça nécessite d'ajouter la méthode `iterator()`)

Voir le code...

`PositionalList.java`

1. déclarer `extends Iterable<E>`

```
public interface PositionalList<E> extends Iterable<E> {
    ...
    Iterator<E> iterator(); // return an iterator on list elements
}
```

2. implémenter la méthode `iterator()`

`Iterable` permet de faire une boucle `foreach`

```
PositionalList<Integer> PL = new LinkedPositionalList<>();
…

// foreach on an Iterable PositionalList
for( Integer i : PL ) System.out.println( i );
```

L'interface `Iterator` possède une méthode optionnelle : `remove`

boolean	**hasNext()** Returns `true` if the iteration has more elements.
E	**next()** Returns the next element in the iteration.
default void	**remove()** Removes from the underlying collection the last element returned by this iterator (optional operation).

```
// test the element iterator of PositionalList
Iterator<Integer> itPL = PL.iterator();
while( itPL.hasNext() ) {
    Integer current = itPL.next();
    System.out.print( current );
    if( current > 5 ) {
        itPL.remove();
        System.out.println( " removed" );
    }
    else System.out.println( " kept" );
}
System.out.println( PL );
```

```
[java] 9 removed
[java] 7 removed
[java] 3 kept
[java] [3]
```

👉 On ne peut pas invoquer `remove` dans un `foreach`

```
// test foreach on the Iterable PositionalList
for( Integer i : PL )
    System.out.println( i );
```

```
[java] 9
[java] 7
[java] 3
```

L'interface `Iterator` exige l'implémentation de `hasNext()` et `next()`

Dans le cas d'une structure positionnelle, on veut itérer sur les positions.
On définit une classe *`PositionIterator`* qui implémente *`Iterator<Position<E>>`*

Voir le code de la classe `PositionIterator` et de la méthode `positions()`

`LinkedPositionalList.java`

Mais on veut aussi itérer sur les éléments

- La classe **`PositionIterator`** implémente un itérateur de positions, **`Iterator<Position<E>>`**, qui implémente **`hasNext`**, **`next`** et (optionnellement) **`remove`**.
- Pour la classe **`ElementIterator`** qui doit implémenter un itérateur d'élément, **`Iterator<E>`**, on fait simplement réutiliser **`PositionIterator`**, qui fait déjà la job ! On utilise une instance de **`PositionIterator`** et on retourne les éléments via la méthode **`getElement`** de l'interface **`Position`**.

  Voir le code ... les classes **`PositionIterator`** et **`ElementIterator`** dans **`LinkedPositionalList`**

- La méthode **`positions()`** doit retourner une instance d'une classe itérable sur les positions. Cela nécessite l'implémentation d'une classe itérable sur les positions, **`PositionIterable`**, qui ne fait que réaliser l'interface **`Iterable<Position<E>>`**, càd implémenter la méthode **`itertor()`**,qui retourne un **`Iterator<Position<E>>`**, qu'on a déjà dans la classe **`PositionIterator`**.

  Voir le code ... la classe **`PositionIterable`** et la méthode **`positions()`** dans **`LinkedPositionalList`**

Maintenir les fréquences d'accès d'une liste de favoris

La liste positionnelle est utile dans un certain nombre d'applications :

- un **programme qui simule un jeu de cartes** pourrait modéliser la main de chaque personne sous forme de liste positionnelle. Étant donné que la plupart des gens conservent ensemble des cartes de la même couleur, l'insertion et le retrait de cartes de la main d'une personne pourraient se faire en utilisant les méthodes de la liste positionnelle, les positions étant déterminées par un ordre naturel des couleurs ;
- un simple **éditeur de texte** intègre la notion d'insertion et de suppression de position, car de tels éditeurs effectuent généralement toutes les mises à jour relatives à un curseur, qui représente la position actuelle dans la liste des caractères du texte en cours d'édition.

Ici, nous allons regarder une application qui maintient une collection d'éléments et garde une trace du nombre de fois où chacun est accédé (sa fréquence d'accès).

Des exemples de tels scénarios incluent :

- un **navigateur Web** qui garde une trace des pages les plus consultées d'un utilisateur/fournisseur ;
- une **collection de Track** de musique qui maintient une liste des chansons les plus fréquemment jouées ("jukebox").

Nous allons implémenter cette dernière application avec une liste de favoris (FavoritesList) qui prend en charge les méthodes size et isEmpty ainsi que :

`Position<E> add( E e )` : insère l'élément `e` à la fin de la liste avec `count` initial de 0, retourne sa `Position`

`E access( Position<E> p )` : incrémente le count et retourne l'élément à la `Position p`

`E remove( Position<E> p )` : supprime et retourne l'élément à la `Position p`, s'il existe

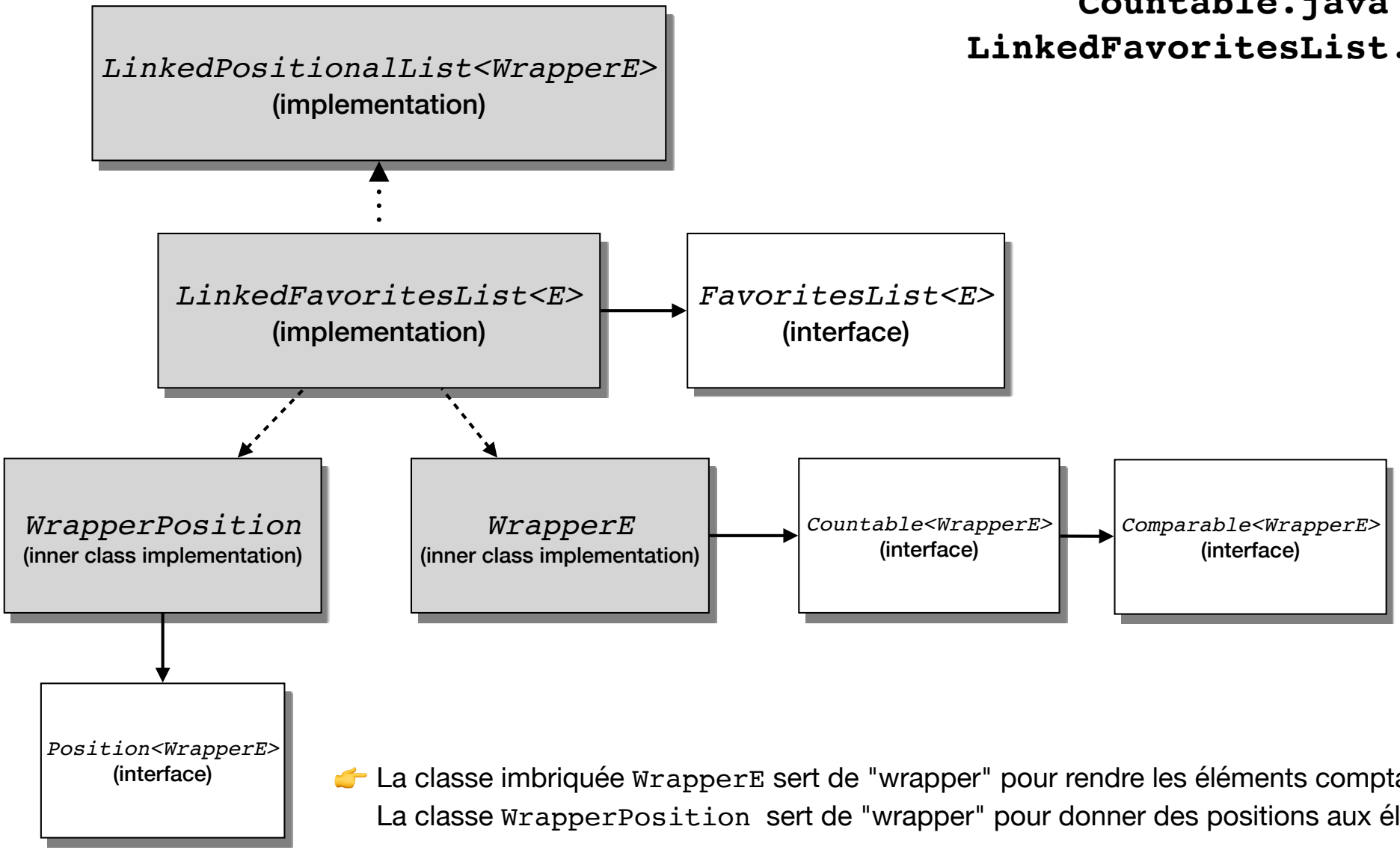
`Iterable<E> getFavorites( int k )` : retourne une collection itérable des k éléments accédés le plus souvent (le top-k)

Voir le code...

`FavoritesList.java`

Voir le code...

`Countable.java`
`LinkedFavoritesList.java`



👉 La classe imbriquée `WrapperE` sert de "wrapper" pour rendre les éléments comptables.
La classe `WrapperPosition` sert de "wrapper" pour donner des positions aux éléments.

On garde la liste triée avec la méthode `moveUp`

Un élément dans la liste vient d'avoir son `count` incrémenté à 9 (noir).

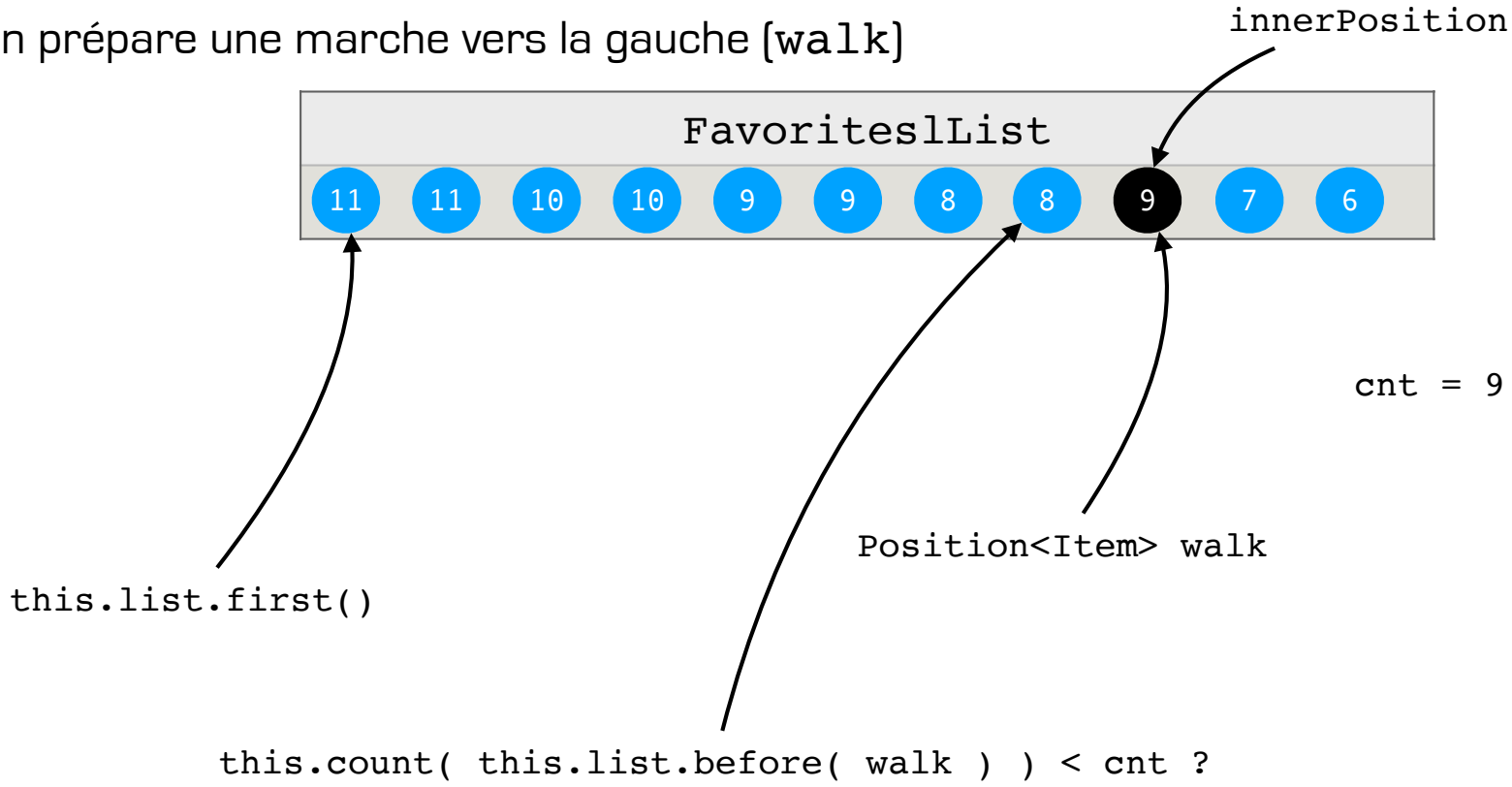
On copie la valeur de son `count` dans `cnt`

On le marque avec `innerPosition`

On prépare une marche vers la gauche (`walk`)

Voir le code...

`LinkedPositionalList.java`

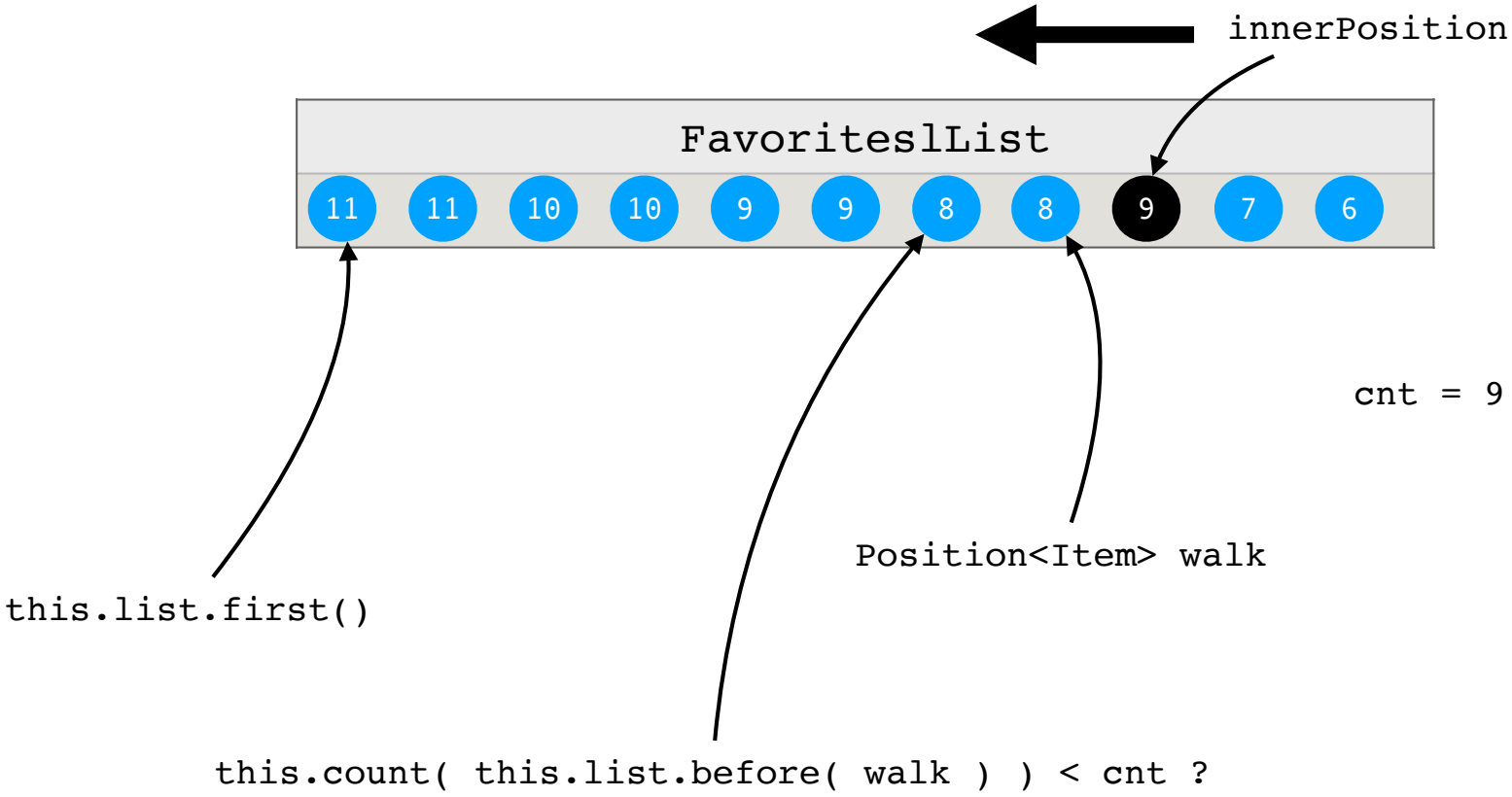


On garde la liste triée avec la méthode `moveUp`

On marche vers la gauche (`walk`) tant que cnt est plus grand que le count de l'item `walk`

Voir le code...

`LinkedPositionalList.java`

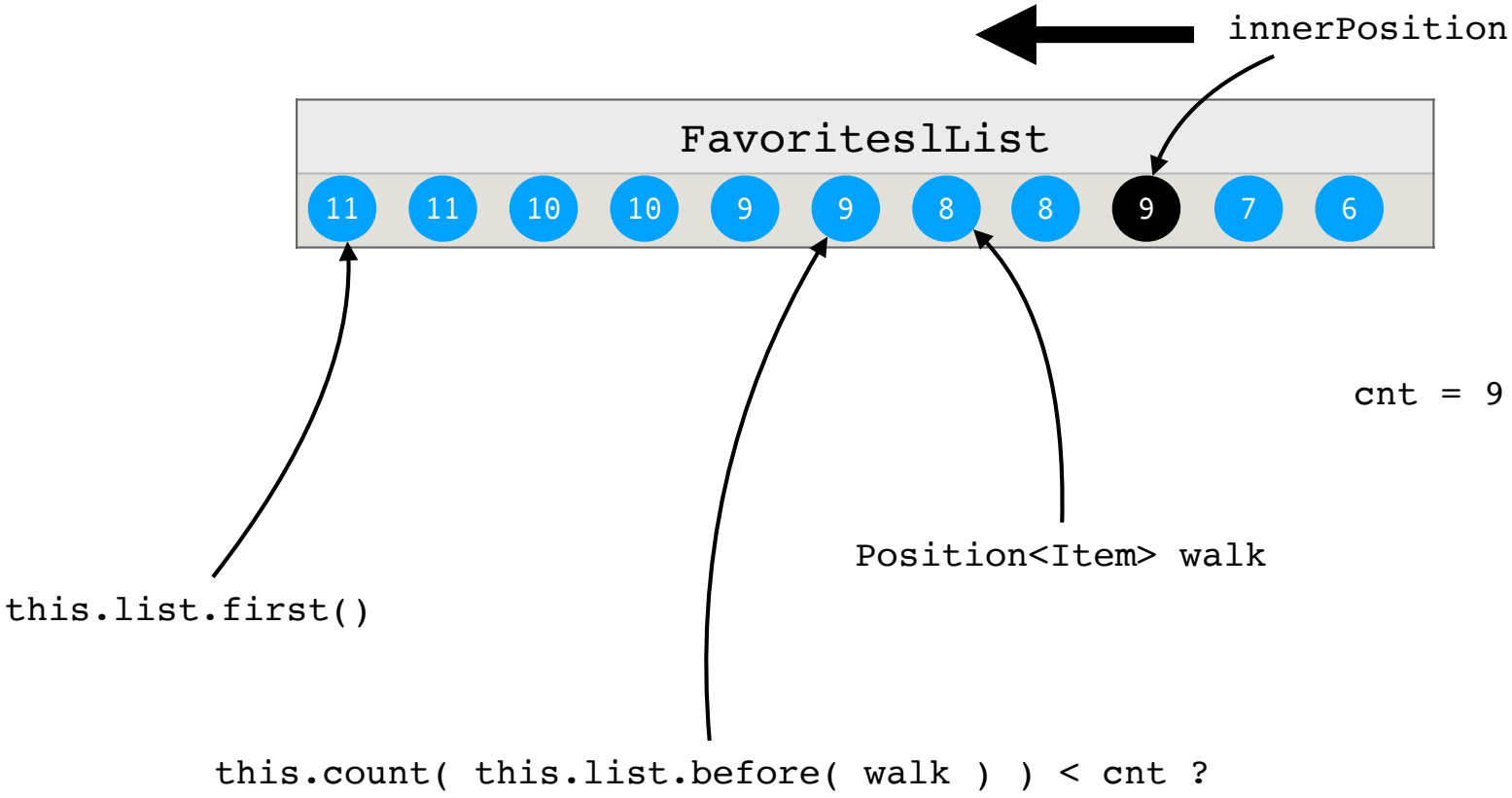


On garde la liste triée avec la méthode `moveUp`

On marche vers la gauche (`walk`) tant que `cnt` est plus grand que le `count` de l'item `walk`

Voir le code...

`LinkedPositionalList.java`

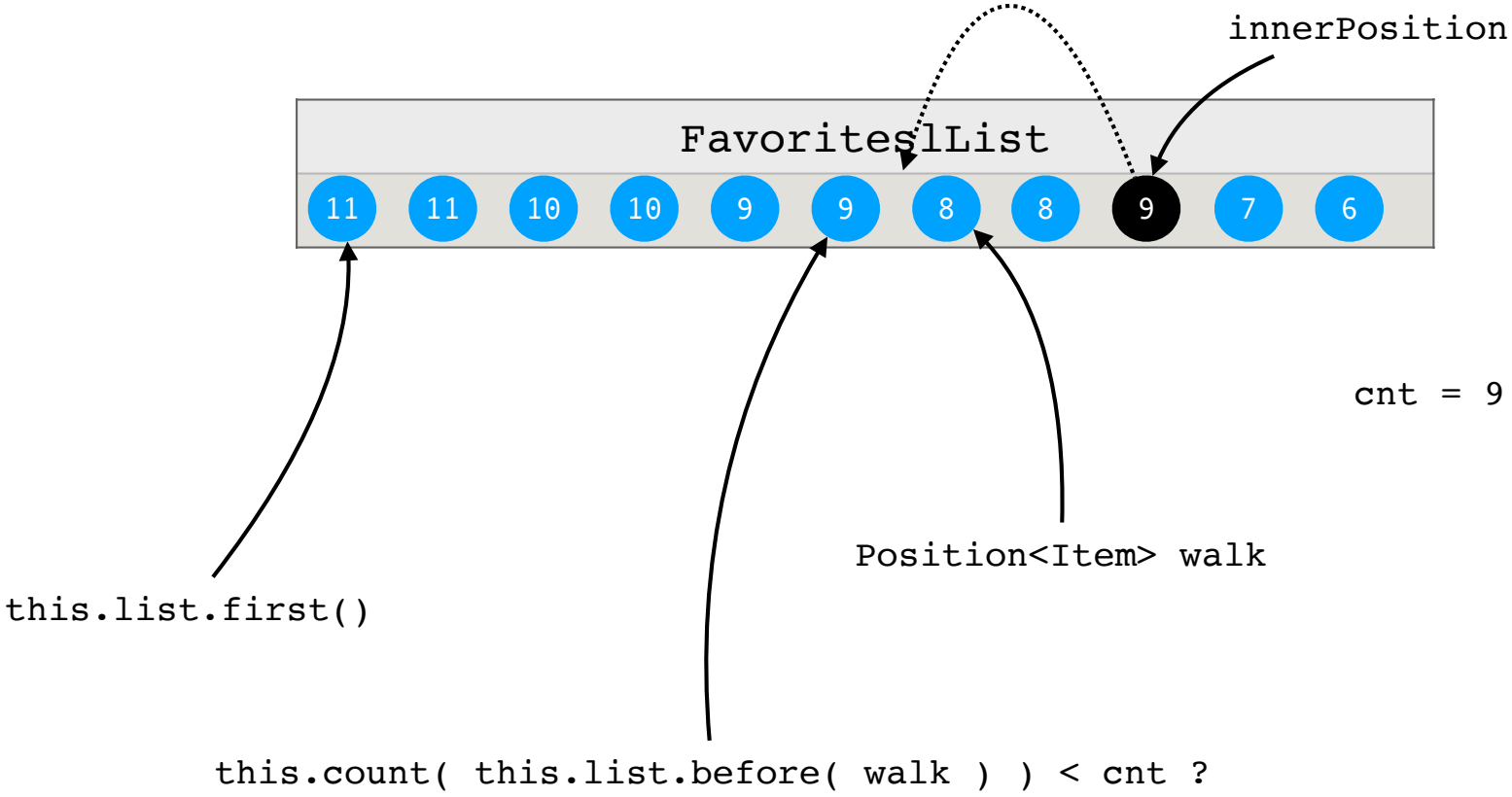


On garde la liste triée avec la méthode `moveUp`

Quand `cnt` n'est plus > que le `count` de l'item devant `walk`, on déplace l'item de la `Position innerPosition` devant `walk`

Voir le code...

`LinkedPositionalList.java`

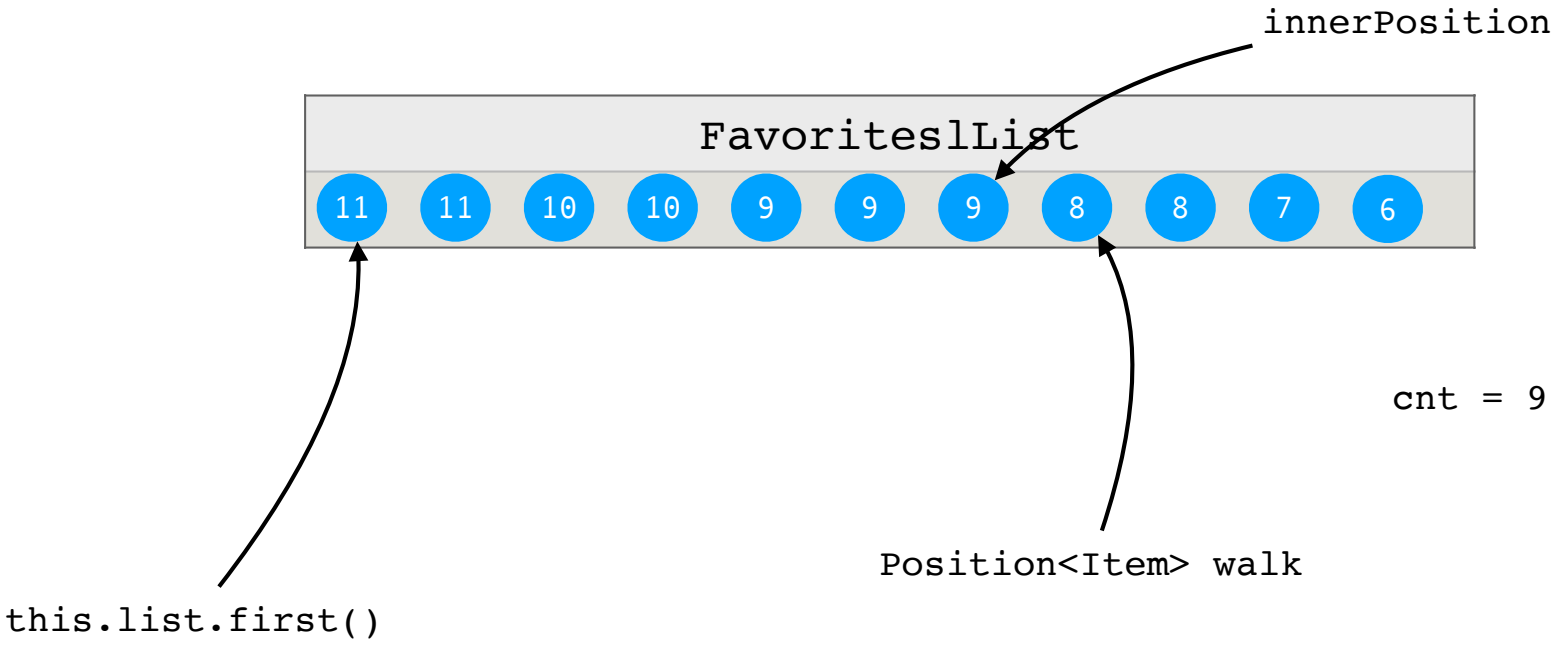


On garde la liste triée avec la méthode `moveUp`

C'est fini !

Voir le code...

`LinkedPositionalList.java`



Garder la liste triée coûte cher, $O(n^2)$!

```
for n: 100,000 m: 10,000,000 k: 5 it takes 23.5 seconds for a sorted (moveUp) strategy
for n: 100,000 m: 10,000,000 it takes 24.9 seconds to sort
```

Que peut-on faire ?

1) On ne maintient plus les items de la liste triés, ce qui enlève le `moveUp()` dans $O(n)$ à chaque accès. On "override" `moveUp()` avec une l'opération `moveFirst()` dans O(1) de sorte à ce que les items les plus accéder se retrouve quand même au début de la liste (heuristique).

2) On "override" `getFavorites( k )` :

a. On copie les entrées de la liste de favoris dans une liste temporaire, `tmp` ;

b. On scanne `tmp` k fois. À chaque scan, on trouve l'entrée avec le plus grand nombre d'accès, on l'ajoute aux résultats et on le supprime de `tmp`.

Cette stratégie de bouger-au-front (“Move-To-Front”) s’exécute en $O(kn)$.

Voir le code...

`FavoritesListMTF.java`
`FavoritesMuzikApp.java`

```
for n: 100,000 m: 10,000,000 k: 5 it takes 0.8 seconds for the MTF strategy
```

Remarques conclusives

- On a vu les concepts de position et de liste positionnelle
- Une position permet d'effectuer des opérations en O(1), qui sont normalement en O(n) si on doit trouver cette position avant de les effectuer
- On a définit un ADT pour des listes positionnelles
- On a implémenté l'ADT avec une liste doublement chaînée
- On a discuté et proposé une solution au problème de "container" d'une position
- On a discuté le tri d'une liste positionnelle
- On a montré comment rendre itérable une liste positionnelle, sur ses éléments mais aussi sur ses positions
- On a regardé la méthode optionnelle `remove` d'un itérateur
- On a définit un ADT pour des listes de favoris, qu'on a implémenté avec une liste positionnelle.
- On a regardé 2 stratégies pour maintenir les comptes d'accès aux éléments