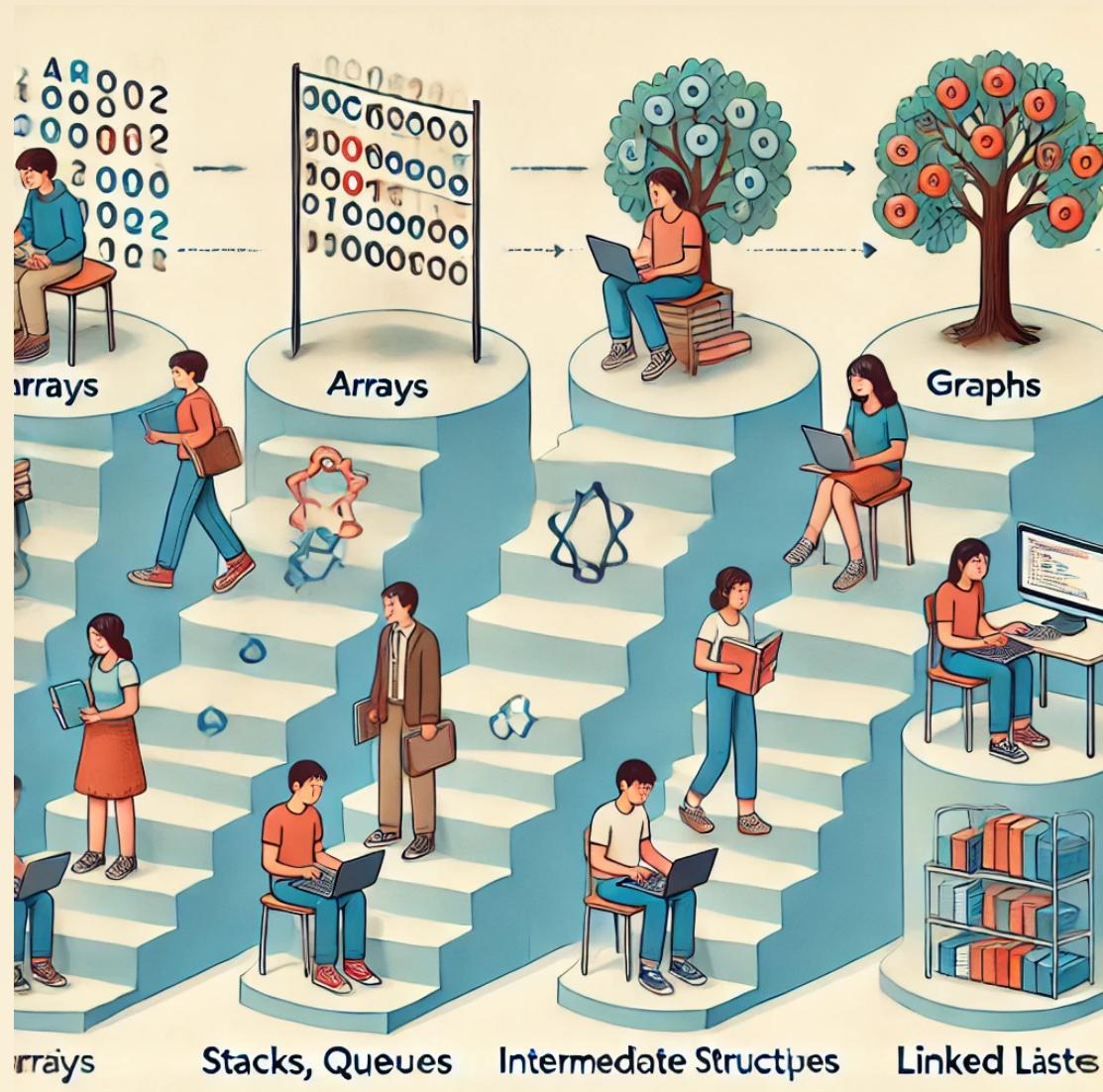


# Introduction: Mise à niveau

(révision pour ceux et celles qui ont fait IFT1025 - Programmation 2 avec moi)



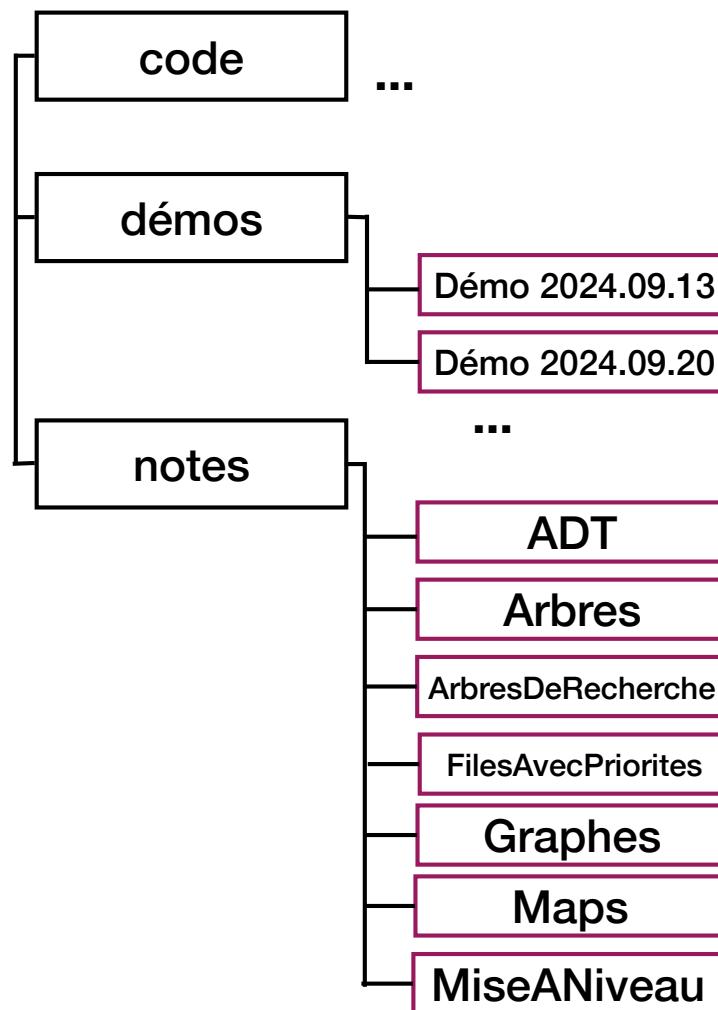
# Objectifs

1. Présenter l'organisation du répertoire de travail (code)
2. S'assurer que tous soient au même niveau en termes des concepts Java à venir dans le cours, en particulier les interfaces, types paramétriques, itérateurs, exceptions
3. Présenter un exemple simple montrant les rouages d'un projet de développement et l'introduction d'un nouveau type abstrait de données (ADT) et de structures physiques pouvant l'implémenter
4. Réviser les tableaux et les listes chaînées en Java
5. *Comparer les temps d'exécution de Java et Python sur un ADT simple*
6. Rafraîchir les notions et outils d'analyse de complexité computationnelle
7. Rafraîchir l'utilisation de la récursivité en Java

# Plan

1. Organisation du code (Apache Maven et Ant)
2. Tableaux et listes : scénario avec séquences d'objets
3. Outils d'analyse
4. Récursivité

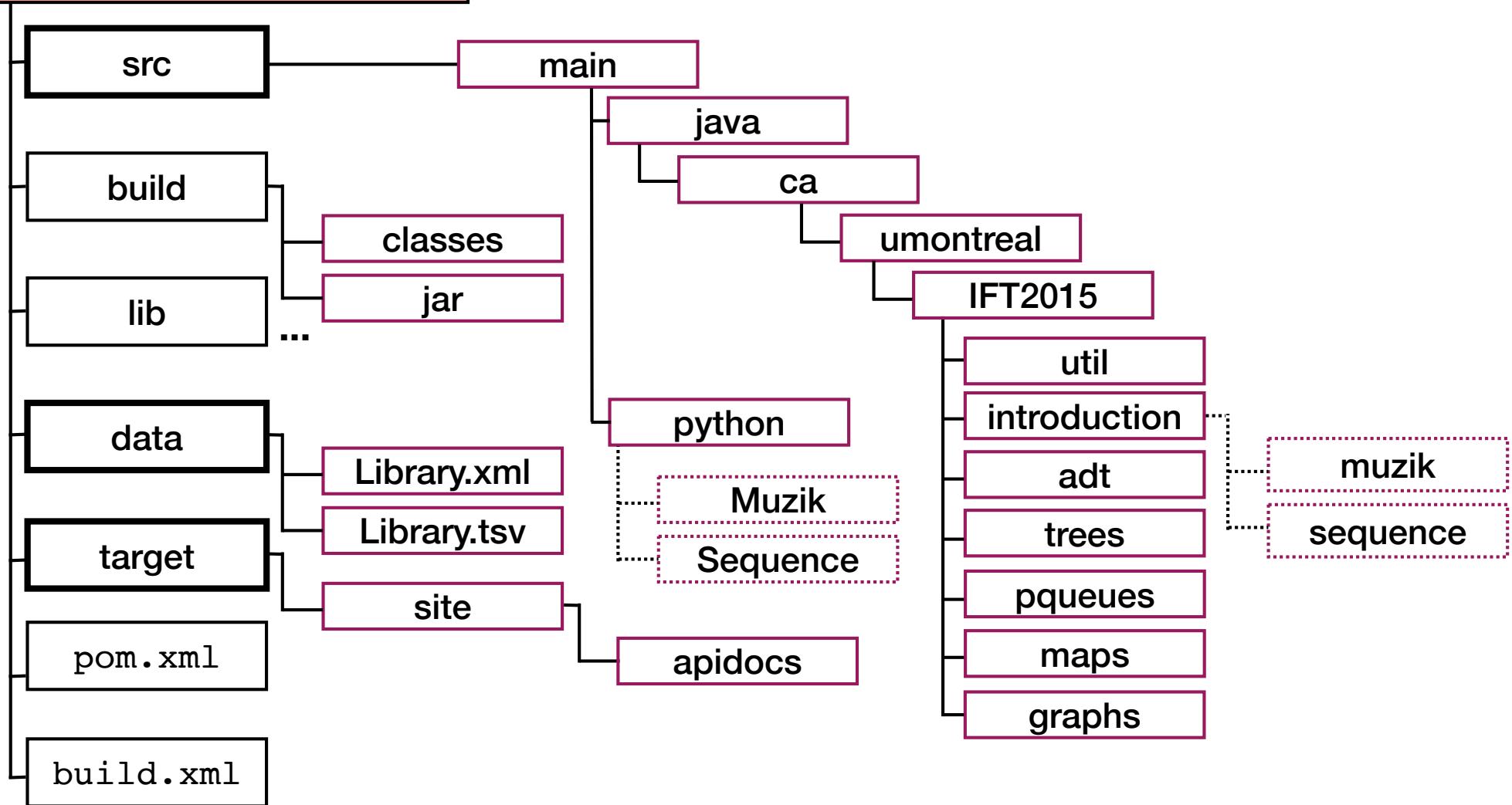
# 1. Structure du répertoire IFT2015



# Structure du répertoire Apache Maven (avec Ant) pour le code

<https://maven.apache.org>

## Maison du projet (code)





## Introduction

**Maven**, a [Yiddish word](#) meaning *accumulator of knowledge*, began as an attempt to simplify the build processes in the Jakarta Turbine project. There were several projects, each with their own Ant build files, that were all slightly different. JARs were checked into CVS. We wanted a standard way to build the projects, a clear definition of what the project consisted of, an easy way to publish project information, and a way to share JARs across several projects.

The result is a tool that can now be used for building and managing any Java-based project. We hope that we have created something that will make the day-to-day work of Java developers easier and generally help with the comprehension of any Java-based project.

Un outil pour unifier et standardiser le développement de plusieurs projets, chacun avec ses propres fichiers. Offre un moyen standard de construire des projets, définir clairement ce en quoi consiste le projet, publier des informations sur le projet et partager les fichiers JAR entre plusieurs projets ; permet de construire et gérer n'importe quel projet basé sur Java.



## Apache Ant™

Apache Ant is a Java library and command-line tool whose mission is to drive processes described in build files as targets and extension points dependent upon each other. The main known usage of Ant is the build of Java applications. Ant supplies a number of built-in tasks allowing to compile, assemble, test and run Java applications. Ant can also be used effectively to build non Java applications, for instance C or C++ applications. More generally, Ant can be used to pilot any type of process which can be described in terms of targets and tasks.

Ant is written in Java. Users of Ant can develop their own "antlibs" containing Ant tasks and types, and are offered a large number of ready-made commercial or open-source "antlibs".

Ant is extremely flexible and does not impose coding conventions or directory layouts to the Java projects which adopt it as a build tool.

Software development projects looking for a solution combining build tool and dependency management can use Ant in combination with [Apache Ivy](#).

The Apache Ant project is part of the [Apache Software Foundation](#).

Outil de ligne de commande dont la mission est de piloter les processus décrits dans les fichiers de construction en tant que cibles dépendantes les unes des autres.

## 2. Applications avec séquences de pistes musicales



# Qu'est-ce qu'une piste musicale, un morceau de musique, une "track" ?

Une piste musicale peut être traitée comme une **abstraction**, et définir une **interface** en Java est une bonne pratique de conception. En programmation orientée objet, une **abstraction** consiste à définir les caractéristiques essentielles d'un objet sans inclure les détails d'implémentation. Une interface est donc un excellent moyen de mettre en œuvre ce concept en Java.

## Une interface pour une piste musicale :

### 1. Sépare les responsabilités (interface vs implémentation)

- L'**interface** se concentre sur **quoi** fait une piste musicale (ex. : obtenir l'artiste, le nom, le genre), sans préciser **comment** cela est fait.
- L'**implémentation** (la classe qui implémente l'interface) gère les détails concrets de **comment** représenter et manipuler une piste musicale.

### 2. Donne de la flexibilité et de l'extensibilité

- En définissant une interface **TrackADT**, on peut créer plusieurs implémentations différentes d'une piste musicale (par exemple, **Track**, **LocalTrack**, **StreamingTrack**, **PlayingTrack**, **DownloadedTrack**), toutes respectant le même contrat.
- Cela rend le système plus flexible, car on peut introduire de nouveaux types de pistes sans modifier le code existant.

### 3. Permet le polymorphisme

- Avec une interface, on peut manipuler des objets de manière polymorphe. Par exemple, vous pouvez avoir une collection d'objets **TrackADT**, qui peut contenir n'importe quel type de piste (que ce soit une **PlayingTrack**, une **StreamingTrack** ou une autre implémentation). Cela permet de traiter différents types de pistes de manière uniforme.

### 4. Donne de la scalabilité

- Le système est plus facile à étendre. Si on introduit plus tard de nouveaux types de pistes (par ex. : **iTunesTrack** ou **LiveTrack**), ils pourront simplement implémenter l'interface **TrackADT**, ce qui permet au système de les accueillir sans changement majeur du code existant.

### 5. Donne de la cohérence

- Chaque implémentation de piste suit les mêmes règles (les méthodes définies dans l'interface), assurant ainsi une cohérence entre les différents types de pistes.

package →

```
package ca.umontreal.IFT2015.introduction.muzik;
```

Auteur →

```
/**  
Created by François Major on 2024.09.04
```

Notice légale →

```
Permission is hereby granted, free of charge, to any person obtaining a copy of this Software and  
associated documentation files, to deal in the Software without restriction, including without  
limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell  
copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to  
the following conditions:
```

```
The following copyright notice and this permission notice shall be included in all copies or  
substantial portions of the Software: "MajorLab Software: Copyright 1994-2024 Université de  
Montréal, François Major's Laboratory".
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT  
NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND  
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES  
OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN  
CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

```
*/
```

Définition →

```
/**  
* TrackADT is the interface for muzik tracks  
*  
* @author      François Major  
* @version     1.0  
* @since       1.0, 2024.09.04  
*/
```

Versions

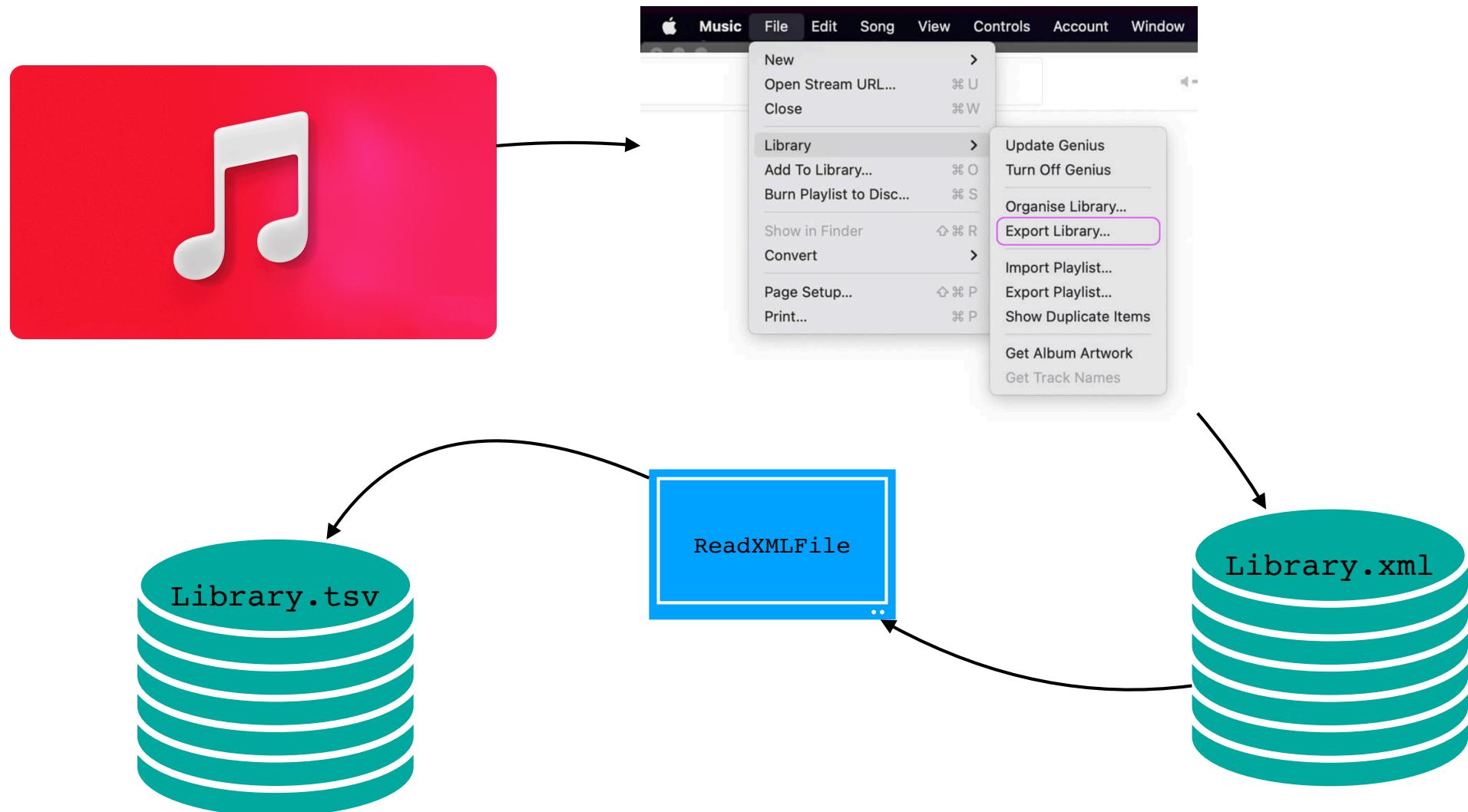
Code →

```
public interface TrackADT extends Comparable<TrackADT> {  
    // Getters  
    public String getArtist();  
    public String getName();  
    public String getAlbum();  
    public int    getBPM();  
    public long   getTotalTime();  
    public Genre  getGenre();  
    public int    getYear();  
    public int    getTrackID();  
    // String setters  
    public void setArtist(  String artist );  
    public void setName(   String name );  
    public void setAlbum(   String album );  
    public void setBPM(    String bpm );  
    public void setTotalTime( String totalTime );  
    public void setGenre(   String genre );  
    public void setYear(   String year );  
    public void setTrackID( String trackID );  
}
```

## Code à voir...

Track.java  
Genre.java

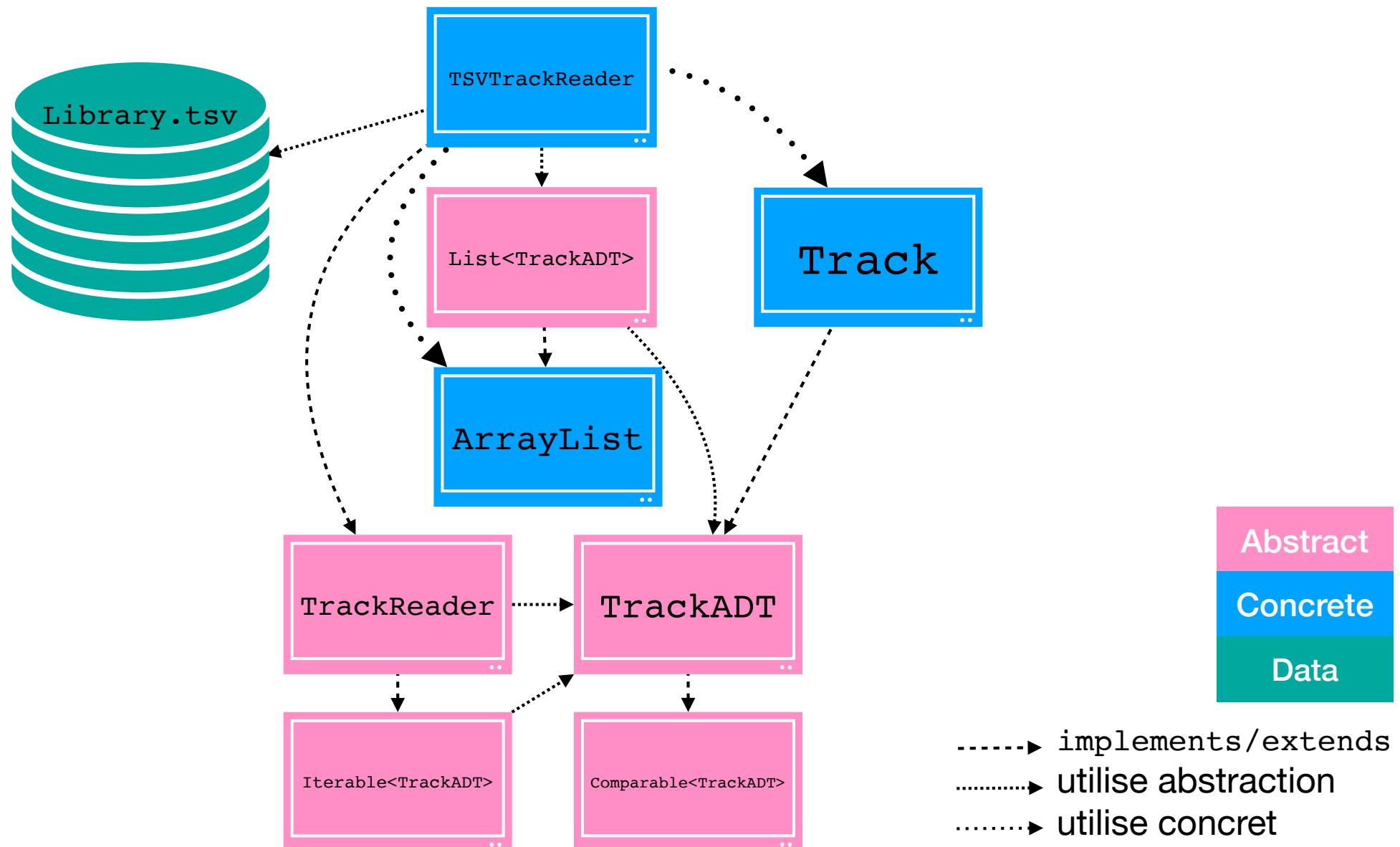
Les applications vont piger dans un fichier de pistes : `Library.tsv`



# "Snapshot" de code/data/Library.tsv (34,041 pistes)

	Artist	Name	Album	BPM	Total Time	Genre	Year	Track ID
13555	<b>Red Hot Chili Peppers</b>	Knock Me Down - 12A	What Hits!?	125	222171	Alternative	1989	64683
13556	<b>Red Hot Chili Peppers</b>	Under The Bridge - 12B	What Hits!?	86	265613	Alternative	1991	64686
13557	<b>Red Hot Chili Peppers</b>	Show Me Your Soul - 8A/7A	What Hits!?	122	261250	Alternative	1992	64689
13558	<b>Red Hot Chili Peppers</b>	If You Want Me To Stay - 4A	What Hits!?	101	245211	Alternative	1985	64692
13559	<b>Red Hot Chili Peppers</b>	Hollywood - 6A	What Hits!?	89	301479	Alternative	1985	64695
13560	<b>Red Hot Chili Peppers</b>	Jungle Man - 11A	What Hits!?	103	247405	Alternative	1985	64698
13561	<b>Red Hot Chili Peppers</b>	The Brothers Cup - 7A	What Hits!?	107	207177	Alternative	1985	64701
13562	<b>Red Hot Chili Peppers</b>	Taste The Pain - 9A	What Hits!?	105	268617	Alternative	1992	64704
13563	<b>Red Hot Chili Peppers</b>	Catholic School Girls Rule - 8A	What Hits!?	98	115565	Alternative	1985	64707
13564	<b>Red Hot Chili Peppers</b>	Johnny Kick A Hole In The Sky - 9A	What Hits!?	121	310125	Alternative	1992	64710
13565	<b>Rednek</b>	Game Over - Calvertron Remix - 4A	Game Over	110	270654	Maya Dubstep	2011	64713
13566	<b>Rednex</b>	Cotton Eye Joe - 11B - 7	Sex & Violins	132	194011	Maya Dance	1994	64716
13567	<b>Redsky</b>	Takin Over You feat. Lisa Law - Ben Macklin Remix - 8A	Takin Over You	129	508029	House	2006	64719
13568	<b>Reel Big Fish</b>	Take On Me - 11B - 7	Our Live Album Is Better Than Your Live Album (Live)	102	207693	Maya Rock	2006	64722
13569	<b>Reina</b>	They Say It's Gonna Rain - Original Mix - 4A	This Is Reina	132	243644	Trance	2004	64725
13570	<b>Remady</b>	Give Me a Sign (Radio Edit) [feat. Manu-L] - 8A - 7	Give Me a Sign	128	193902	Maya Dance	2010	64728
13571	<b>Remady</b>	No Superstar (Full Vocal Mix) - 9A	No Superstar	128	319165	Maya Dance	2009	64731
13572	<b>Renaud</b>	Le Père Noël Noir - 7A	The Very Meilleur	136	205531	Pop	1999	64734
13573	<b>Revolver</b>	Get Around Town - 9A - 6	Music For a While (Special Edition)	192	140499	Maya Swing	2010	64737
13574	<b>Revolver</b>	Wind Song - 2A/2B - 7	Wind Song - EP	142	200080	Maya Alternative	2011	64740
13575	<b>Rhythm Plate</b>	Don't Care About You - 11A	Don't Care About You	120	424045			64743
13576	<b>Ricardo Reyna</b>	Balia Conmigo - Original Mix - 7A	Summer Time	128	410749	Latino House	2006	64746
13577	<b>Rice &amp; Beans Orchestra</b>	You've Got Magic	Cross Over (Digitally Remastered)	124	229866	Dance	1995	64749
13578	<b>Richard F, Ralphie Romance</b>	Say What! - Original Mix - 12A	Say What!	130	380081	House	2005	64752
13579	<b>Richard Grey</b>	Lady (Federico Scavo Remix) - 6A	Lady (Remix)	127	488097	Maya House	2011	64755
13580	<b>Richard Grey, Erick Morillo, Jose Nunez</b>	Life Goes On feat. Shawnee Taylor - Richard F Mix - 3A	Life Goes On (Part Two)	130	480130	House	2007	64758
13581	<b>Richard Grey, New Yorker Soul</b>	Fallin - Robbie Rivera Juicy Mix - 9A	Fallin	130	440737	House	2006	64761
13582	<b>Richard Les Crees</b>	All The Love I Own - 11A	All The Love I Own	125	340192			64764
13583	<b>Richard Petit</b>	Ou la la	YUL	115	204373	Maya Quebec	2008	64767
13584	<b>Richard Petit</b>	L'étoile	YUL	118	245413	Maya Quebec	2008	64770
13585	<b>Richard Petit</b>	Ou la la	YUL (Édition Deluxe)		209576	Maya Quebec	2008	64773
13586	<b>Richard Petit</b>	L'étoile	YUL (Édition Deluxe)		253921	Maya Quebec	2008	64775
13587	<b>Richard Séguin</b>	Gentil gentil - 9A/11B	Double vie	65	223059	Musique française	1987	64777
13588	<b>Richard Séguin</b>	J'te cherche - 10B	Double vie	125	240979	Musique française	1987	64780

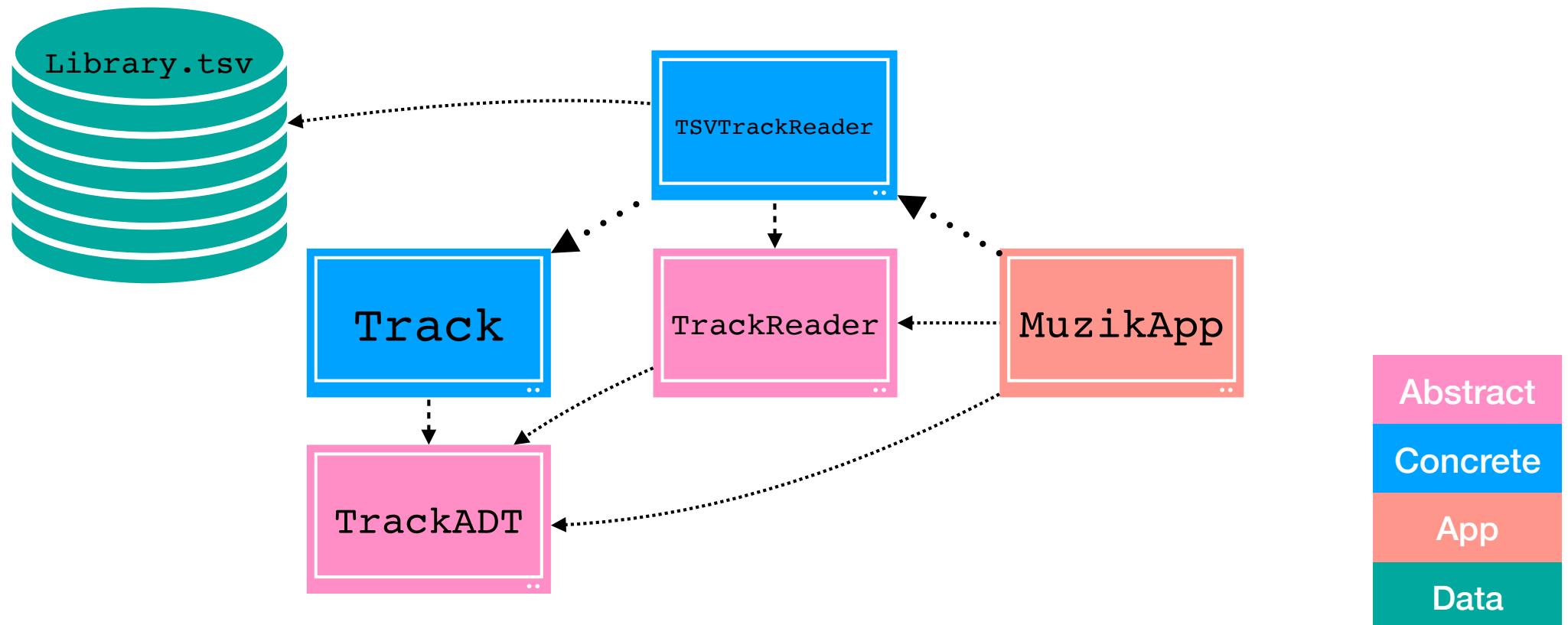
## TSVTrackReader est un TrackReader pour fichier TSV



## Code à voir...

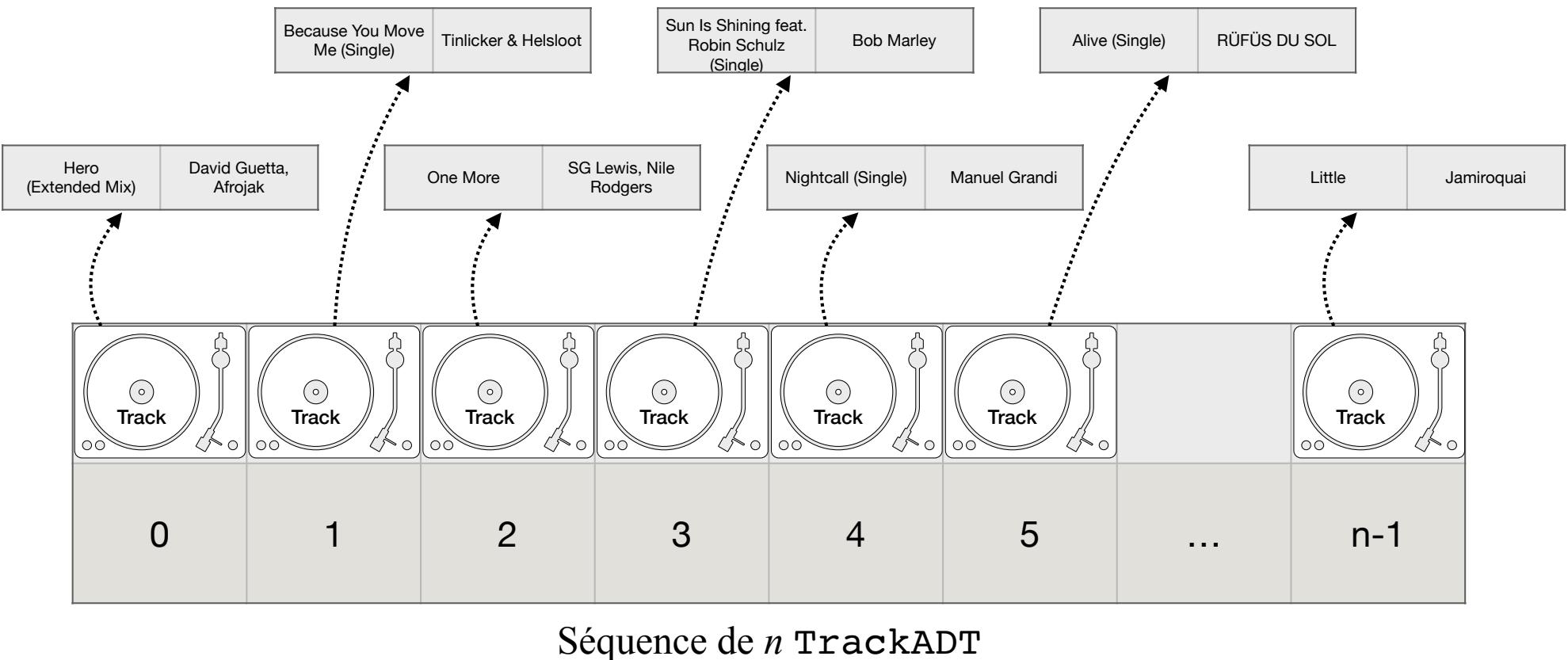
`TrackReader.java`  
`TSVTrackReader.java`

# MuzikApp



..... → implements  
..... → utilise abstrait  
..... → utilise concret

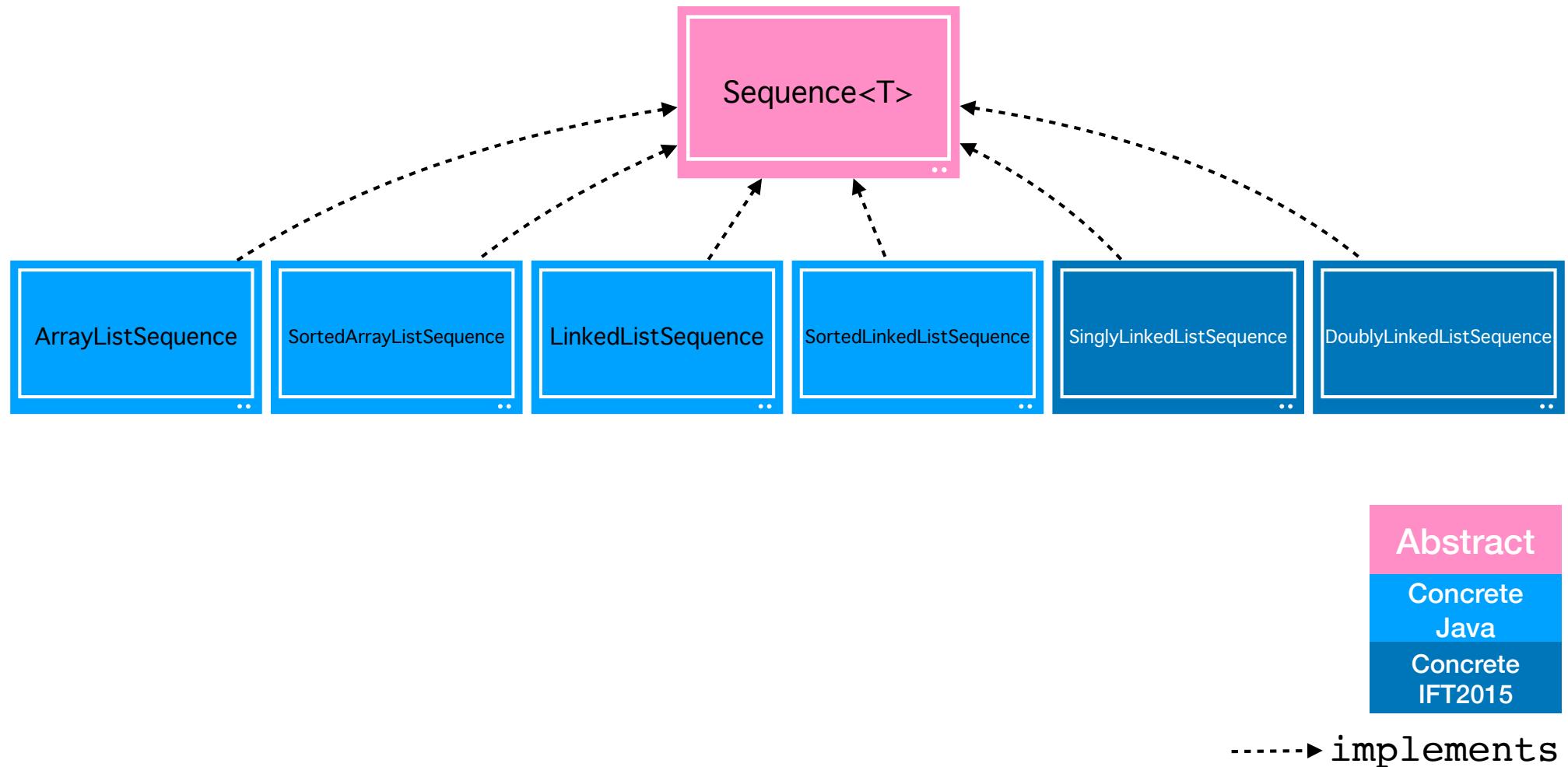
On veut des applications qui utilisent des **séquences** de "tracks"



# Qu'est-ce qu'une séquence ?

```
/**  
 * Sequence is an interface for simple sequence management operations (ADT)  
 * It manipulates Comparable elements  
 *  
 * @author      Francois Major  
 * @version     %I%, %G%  
 * @since       1.0  
 */  
public interface Sequence<T extends Comparable<T>> extends Iterable<T> {  
    public int      size();           // returns the number of elements in sequence  
    public void     add( T t );      // add element t  
    public void     delete( T t );   // delete element t  
    public T        get( int i );    // returns the element at index i  
    public int      index( T t );   // returns the index of the first occurrence of t, or -1 if absent  
}
```

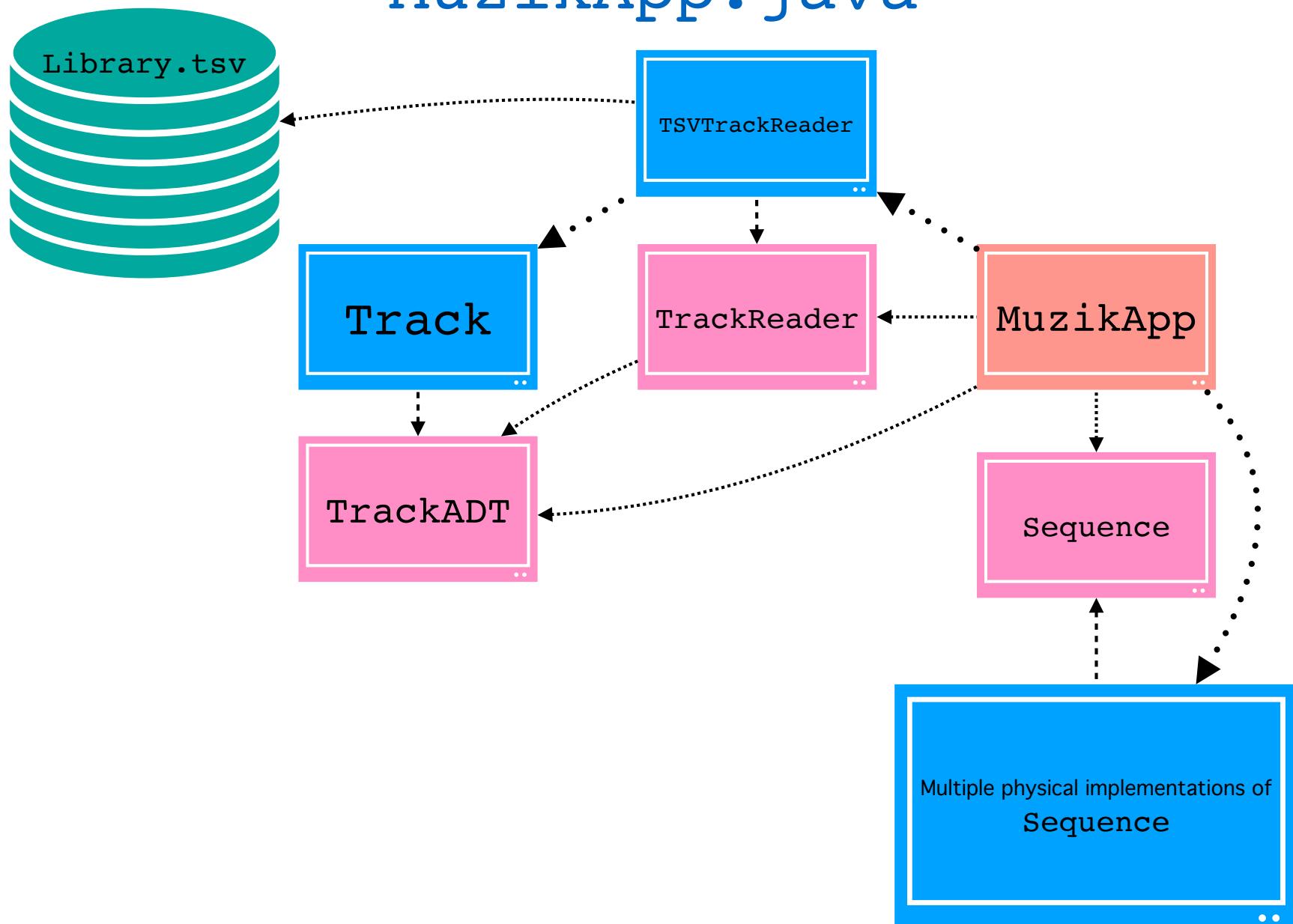
# Comment implémenter Sequence<T>



## Code à voir...

ArrayListSequence.java  
SortedArrayListSequence.java  
LinkedListSequence.java  
SortedLinkedListSequence.java  
DoublyLinkedListSequence.java

# MuzikApp.java



# Comparaison

		ArrayList	ArrayList (sorted)	LinkedList	LinkedList (sorted)	SinglyLinkedList	DoublyLinkedList
Java	add	8 $O(1)$	136 $O(\log n + n)$	5 $O(1)$	7,755 $O(n)$	8 $O(1)$	5 $O(1)$
	delete	167 $O(2n)$	117 $O(\log n + n)$	1,097 $O(2n)$	2,763 $O(2n)$	1,375 $O(2n)$	1,374 $O(2n)$

**Tableau comparatif des opérations `add` et `delete` pour 34,133 "tracks".**

Les temps sont indiqués en millisecondes.

Les `Track` sont triées par ordre lexicographique des titres => `compareTo` coûte cher !

Allons voir ce que ça donne avec la liste Python



## Code à voir...

Track.py  
TSVTrackReader.py  
MuzikApp.py  
Sequence.py  
ListSequence.py  
SortedListSequence.py

# Comparaison

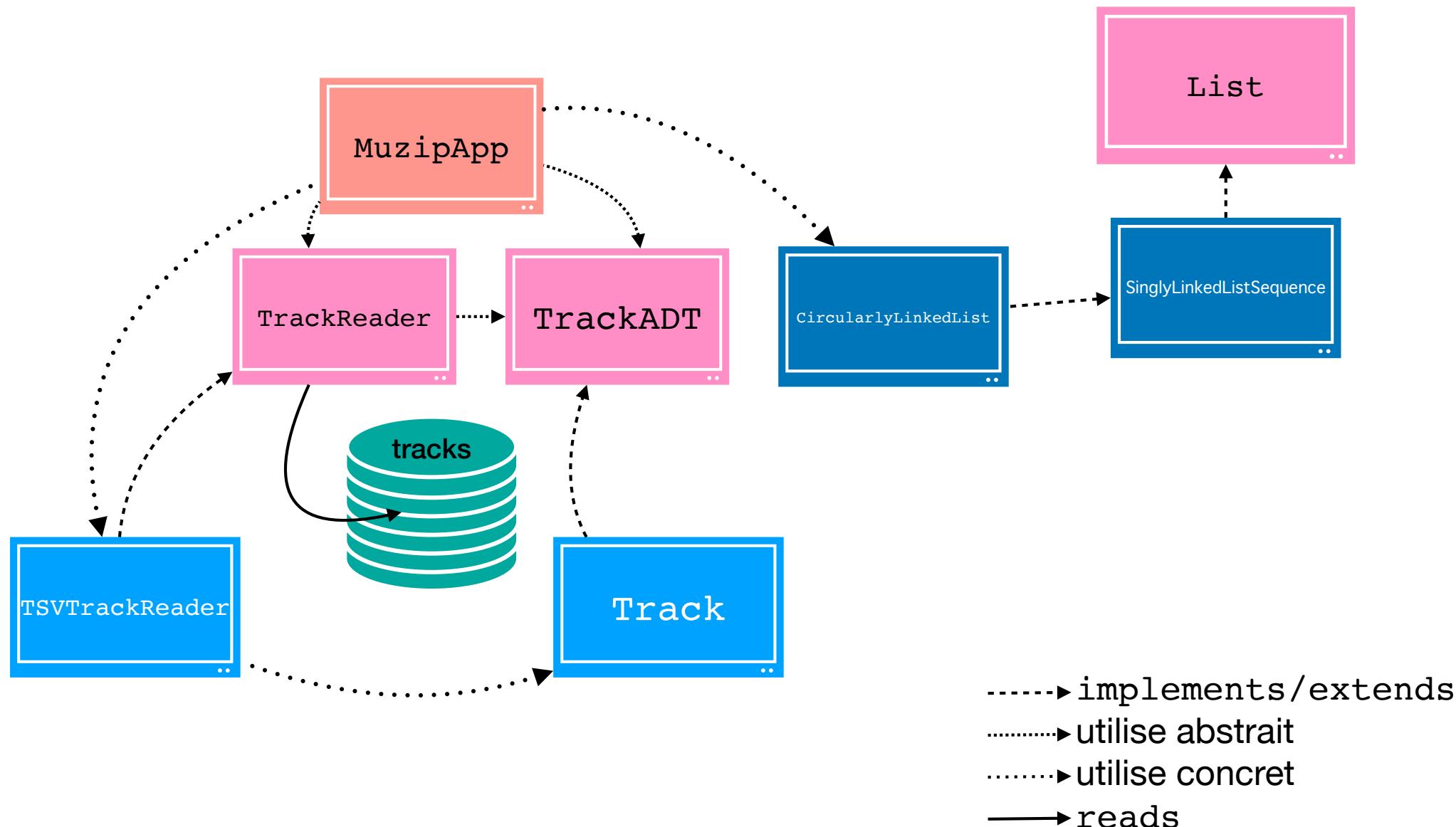
		ArrayList	ArrayList (sorted)	LinkedList	LinkedList (sorted)	SinglyLinkedList	DoublyLinkedList
Java	add	8	136	5	7,755	8	5
		$O(1)$	$O(\log n + n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$
	delete	167	117	1,097	2,763	1,375	1,374
		$O(2n)$	$O(\log n + n)$	$O(2n)$	$O(2n)$	$O(2n)$	$O(2n)$
Python	add	19	517	N/A	N/A	N/I	N/I
		34,617	366	N/A	N/A	N/I	N/I

**Tableau comparatif des opérations `add` et `delete` pour 34,133 "tracks".**

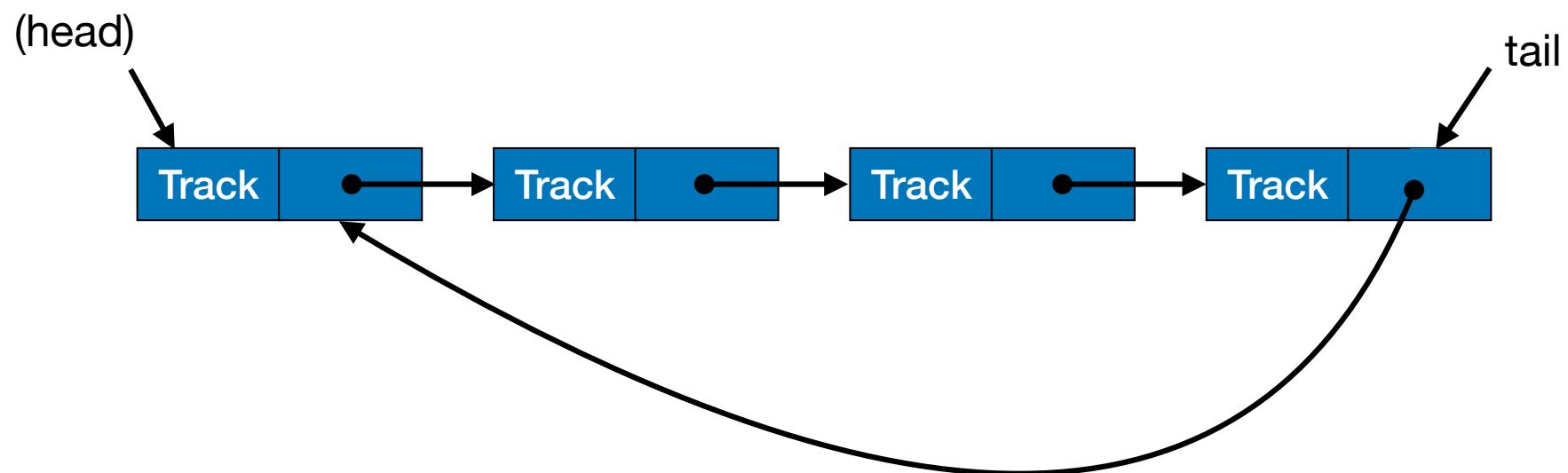
Les temps sont indiqués en millisecondes.

Les `Track` sont triées par ordre lexicographique des titres => `compareTo` coûte cher !

# Jouer une “playlist” (à répétition)

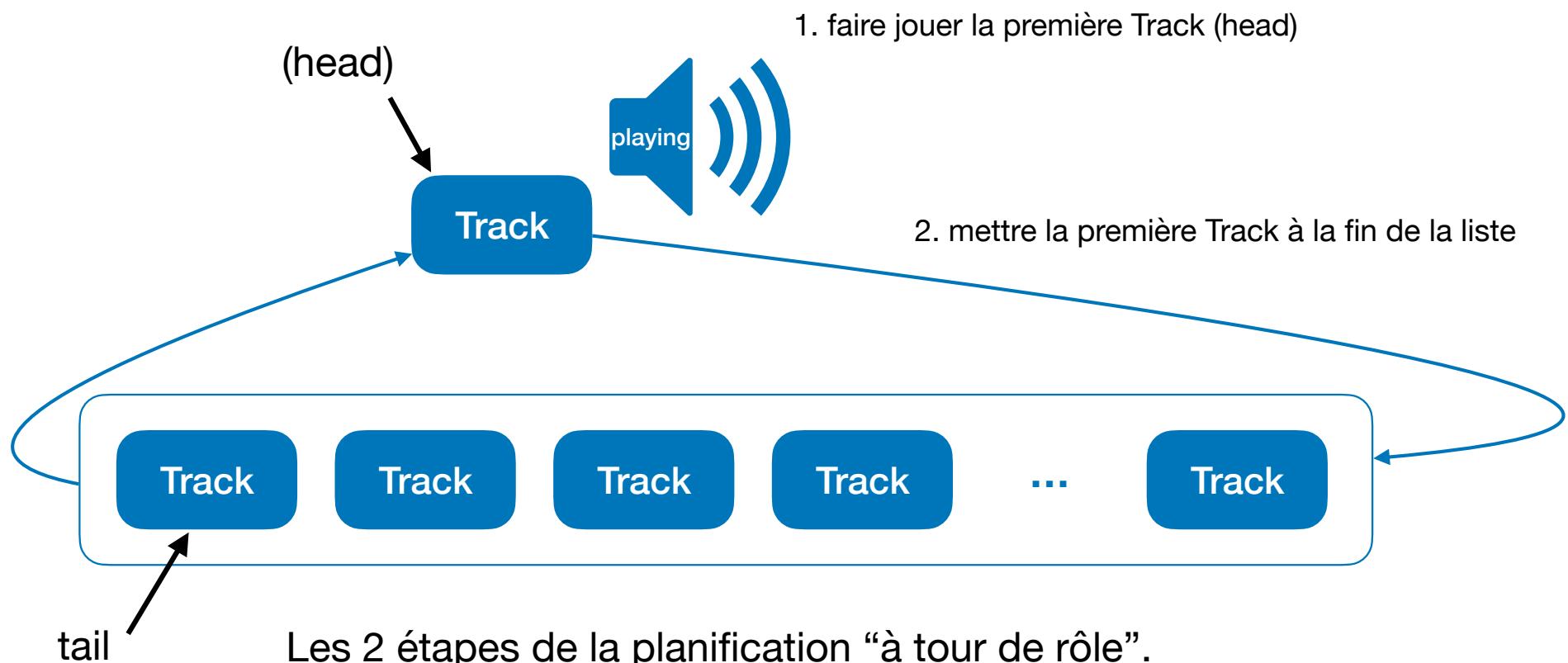


On implémente cette stratégie “à tour de rôle” (round robin) à l'aide d'une liste circulaire (circular list)

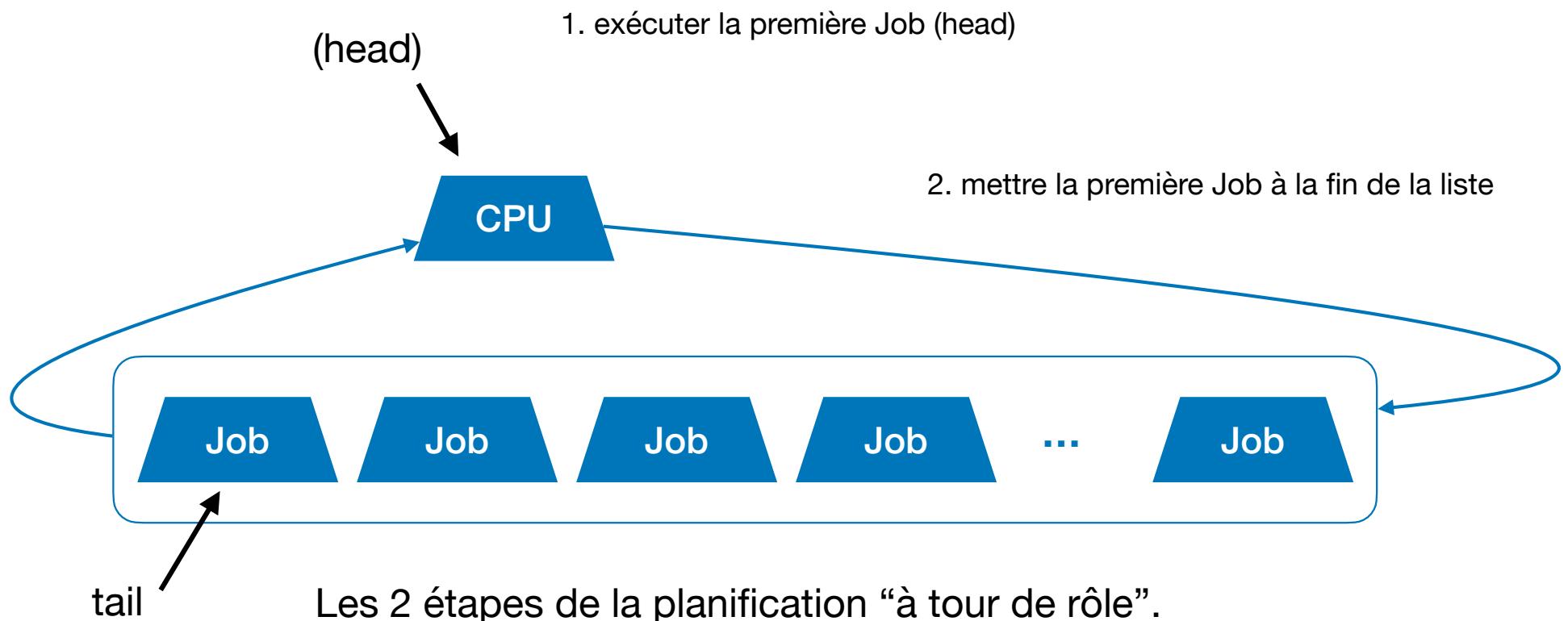


head est défini comme tail.next.

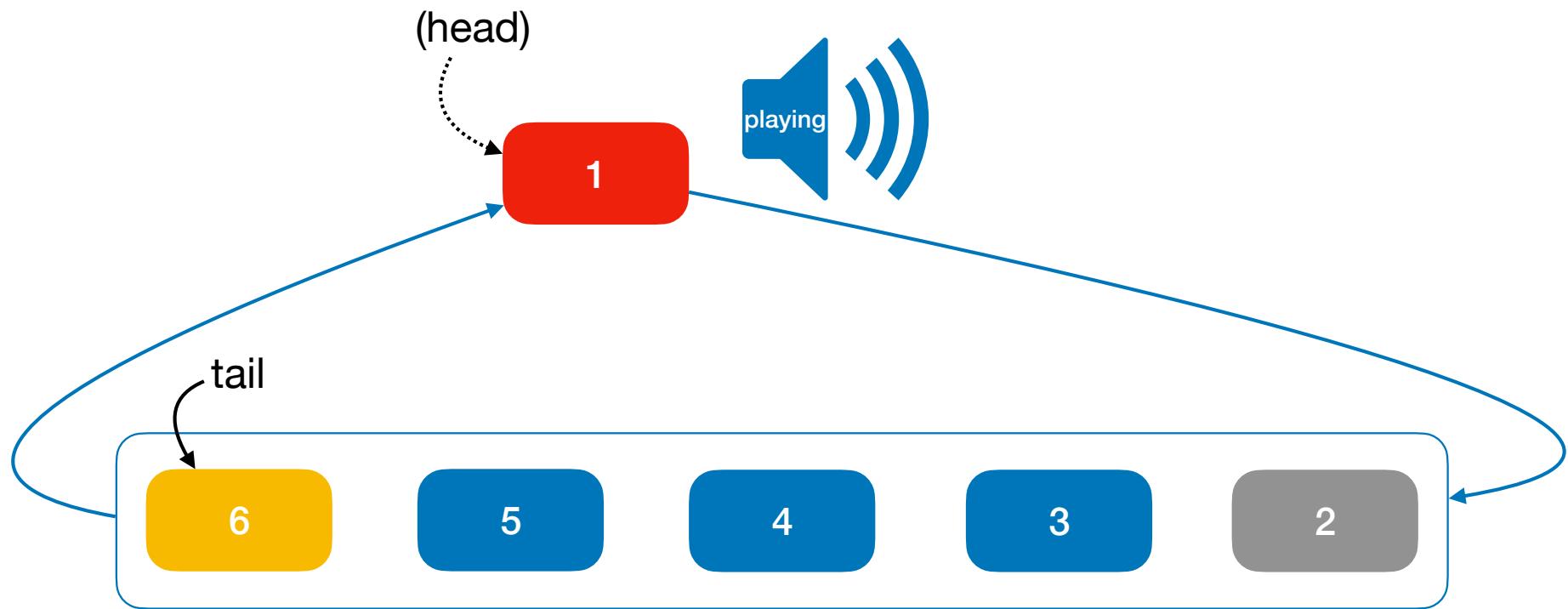
# Jouer une “playlist” à répétition



# Donner du CPU à des “jobs” à tour de rôle

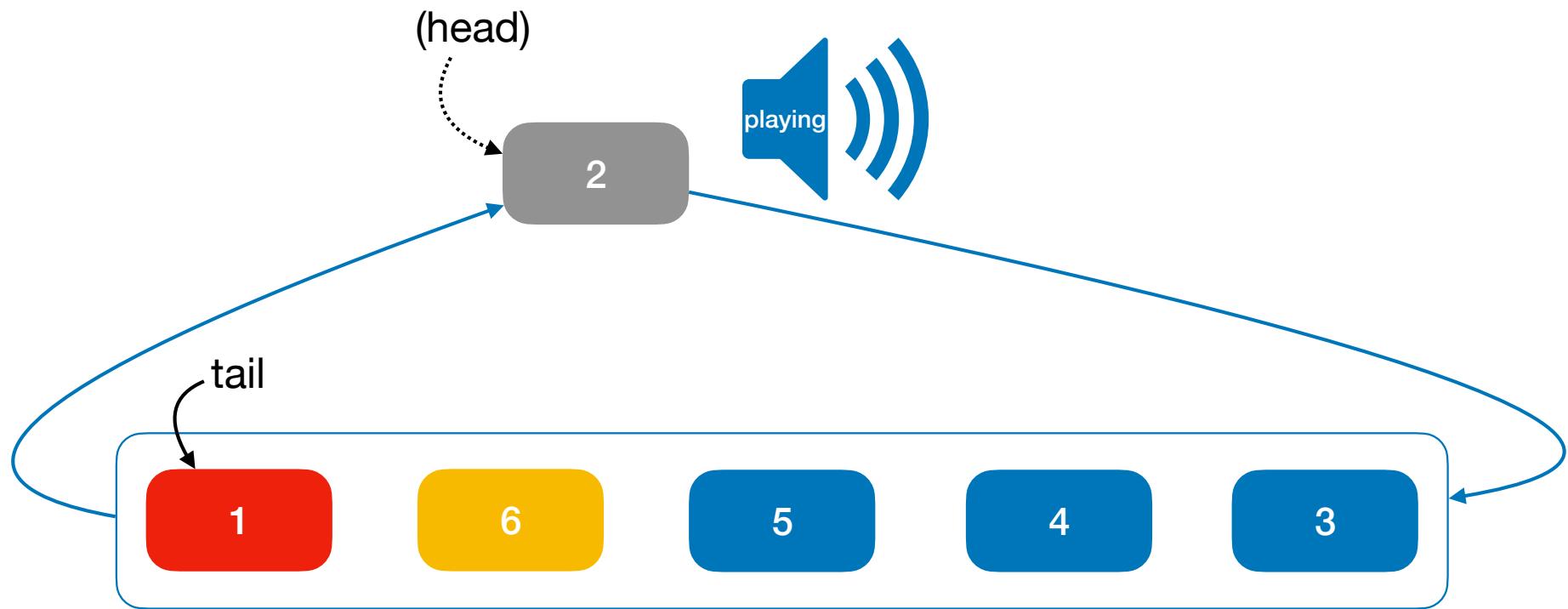


# Passer à la prochaine Track correspond à faire une rotation



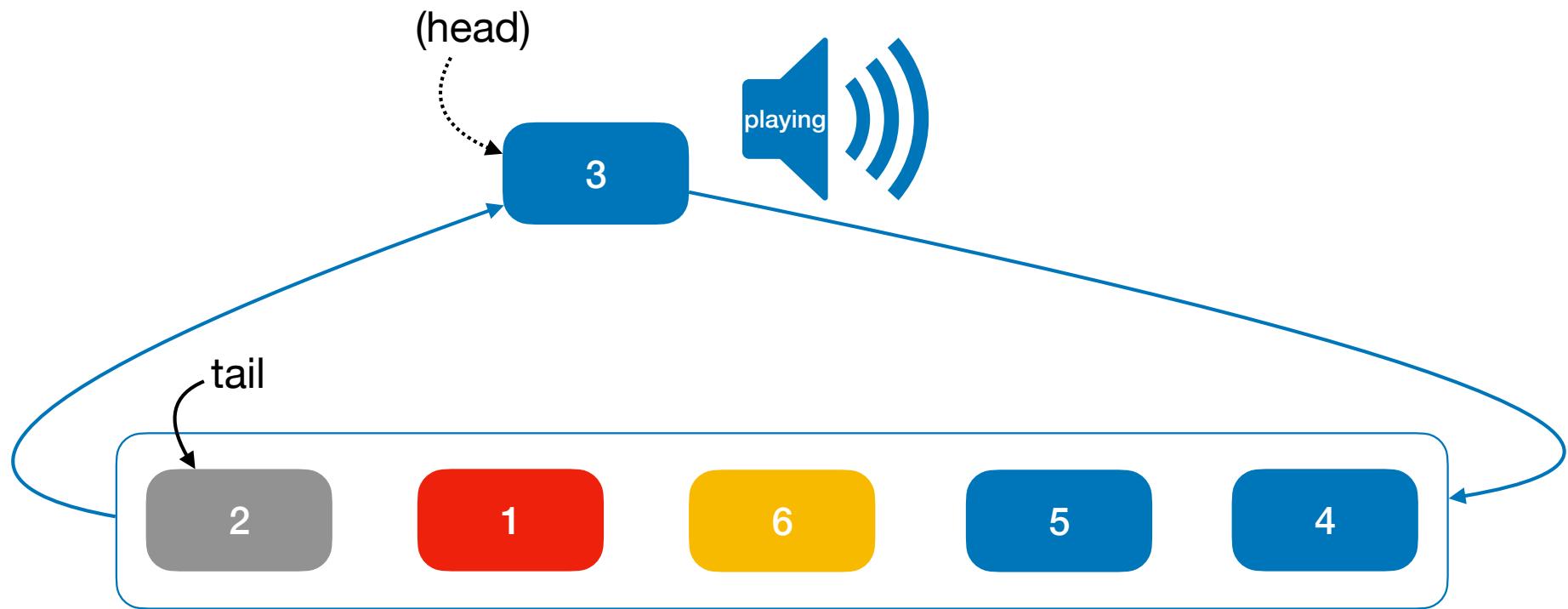
L'opération de rotation dans une liste circulaire.

# Passer à la prochaine Track correspond à faire une rotation



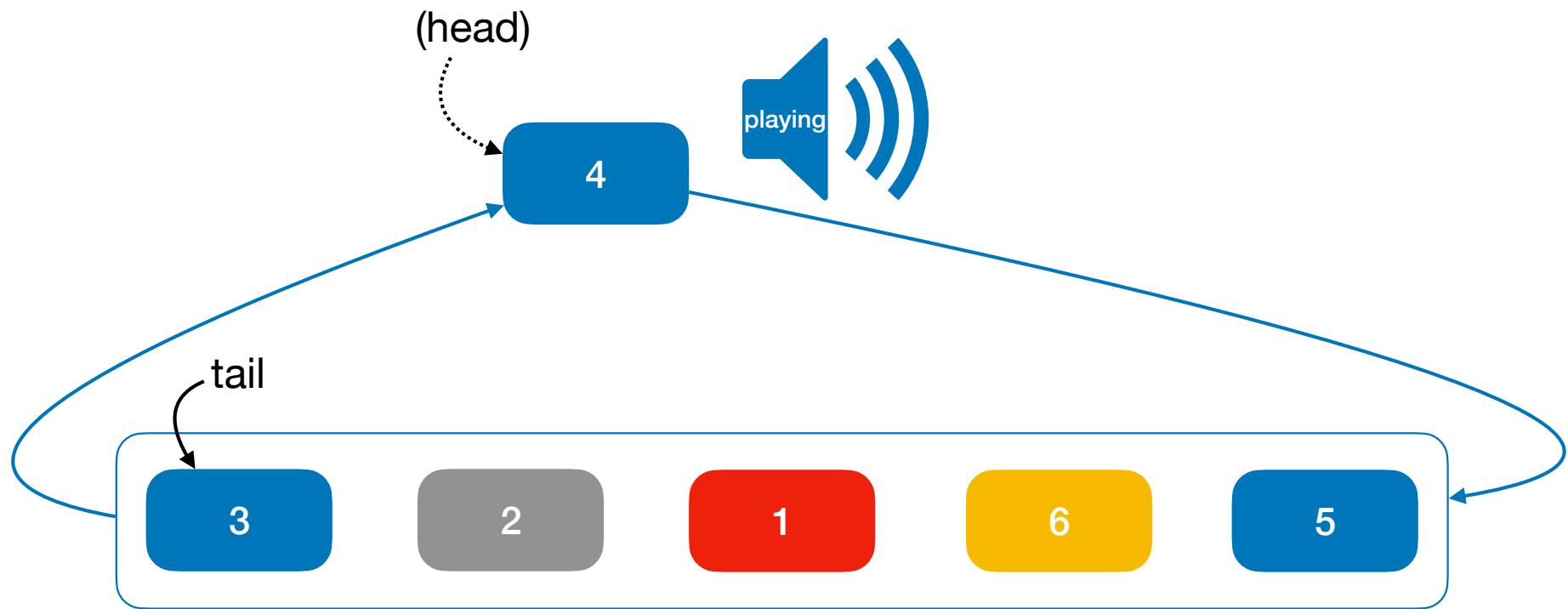
L'opération de rotation dans une liste circulaire.

# Passer à la prochaine Track correspond à faire une rotation



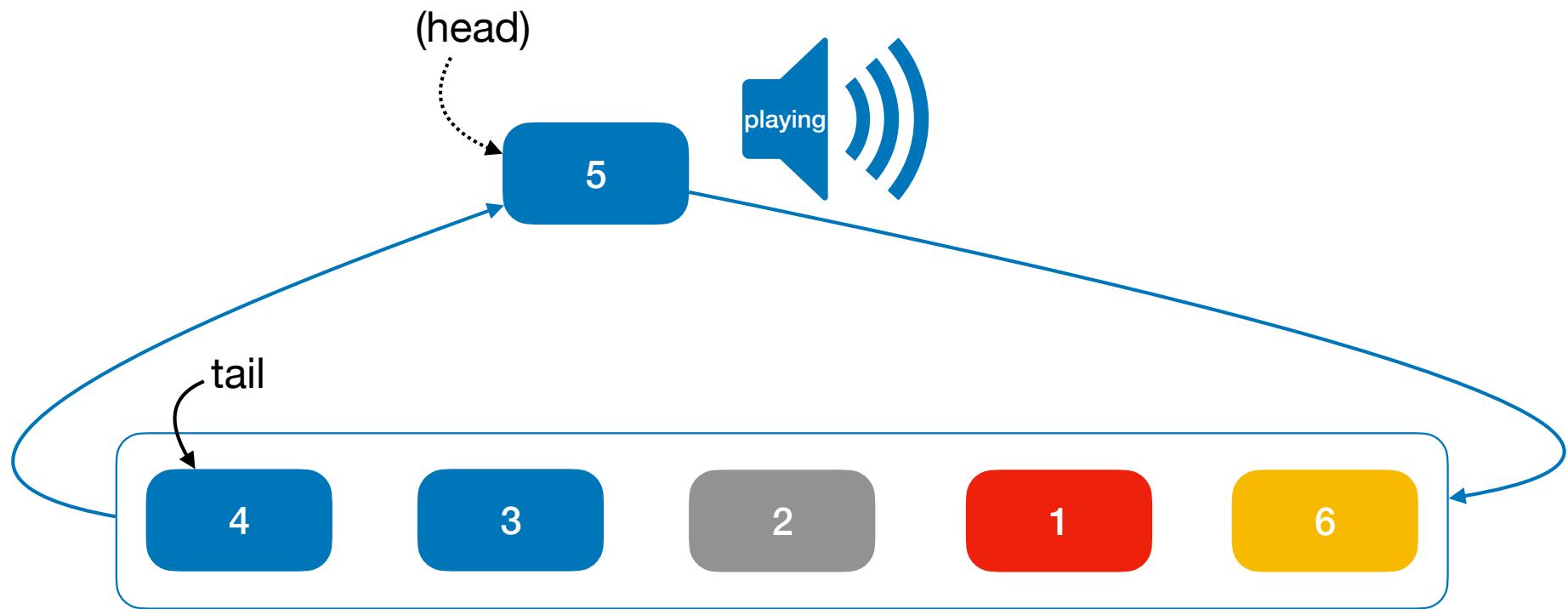
L'opération de rotation dans une liste circulaire.

# Passer à la prochaine Track correspond à faire une rotation



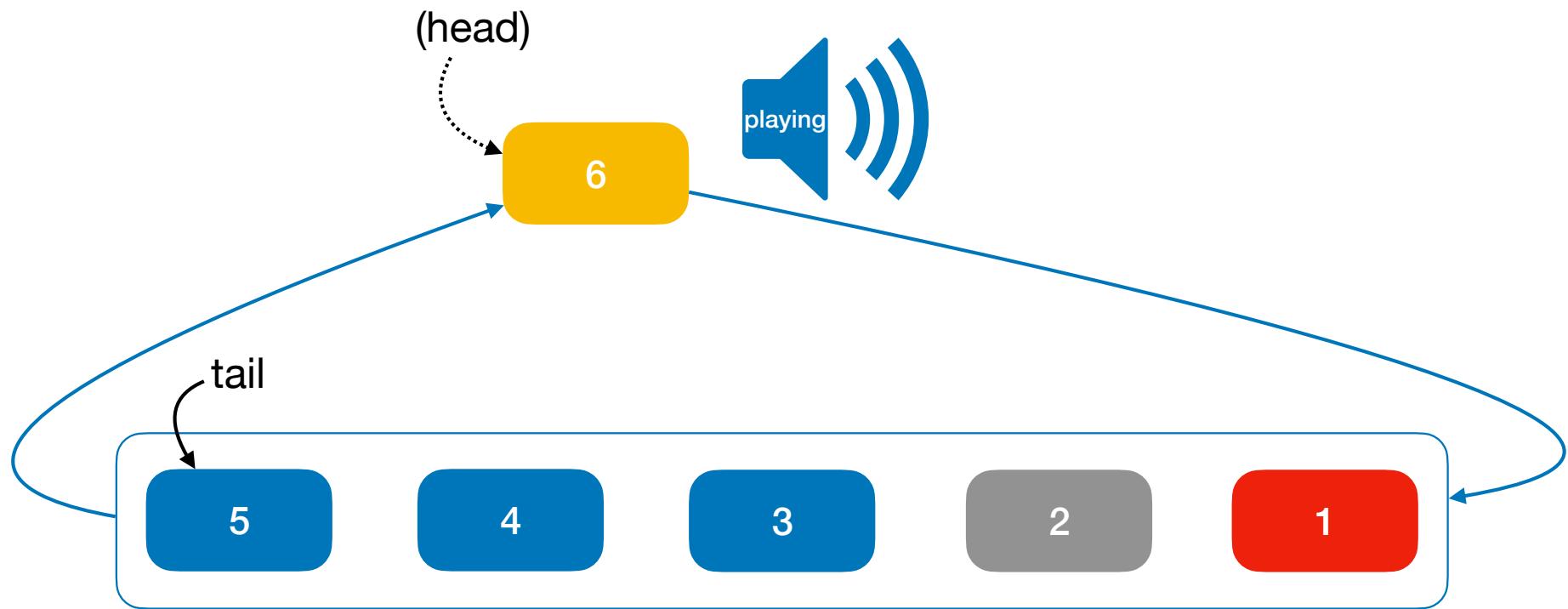
L'opération de rotation dans une liste circulaire.

# Passer à la prochaine Track correspond à faire une rotation



L'opération de rotation dans une liste circulaire.

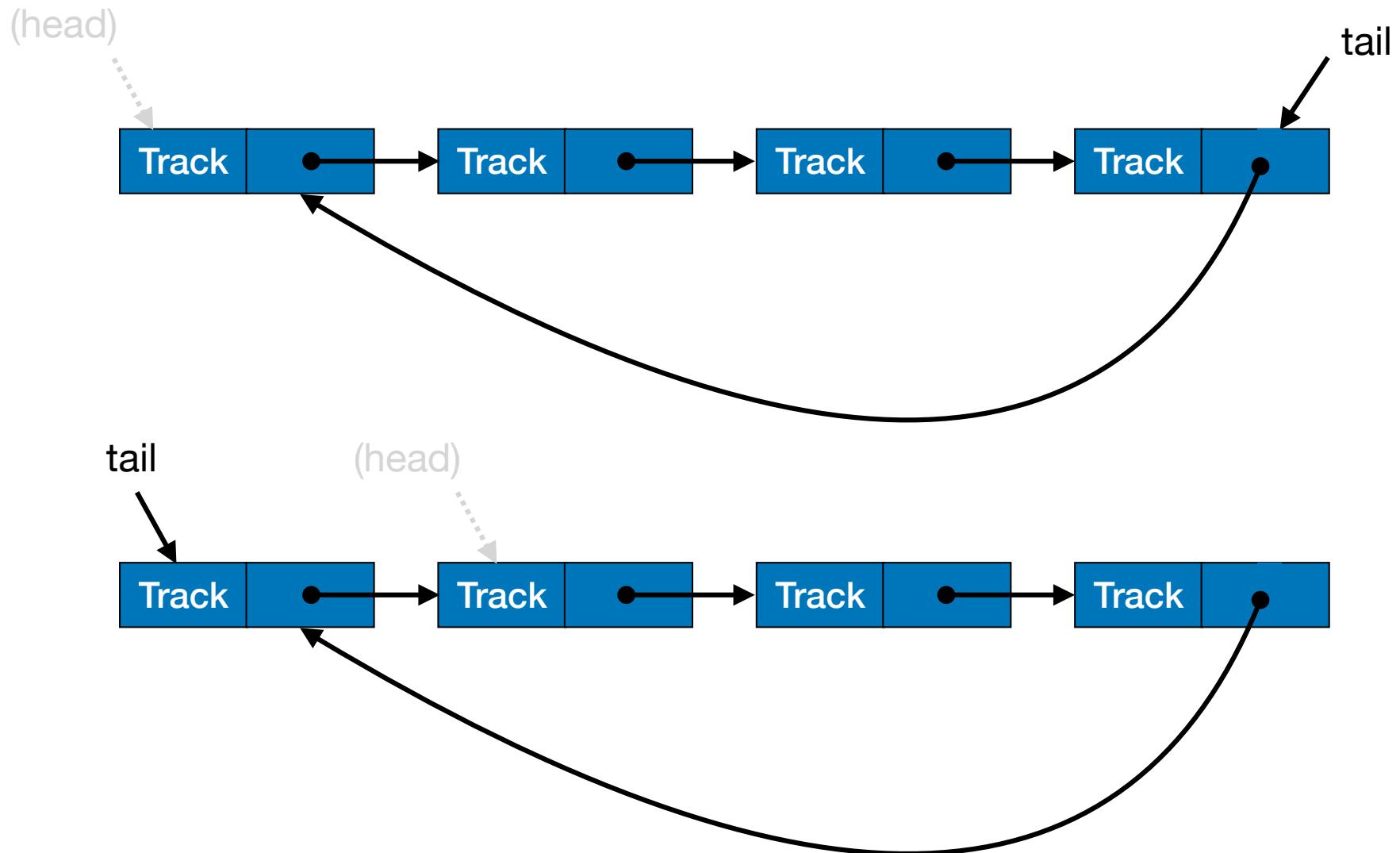
# Passer à la prochaine Track correspond à faire une rotation



L'opération de rotation dans une liste circulaire.

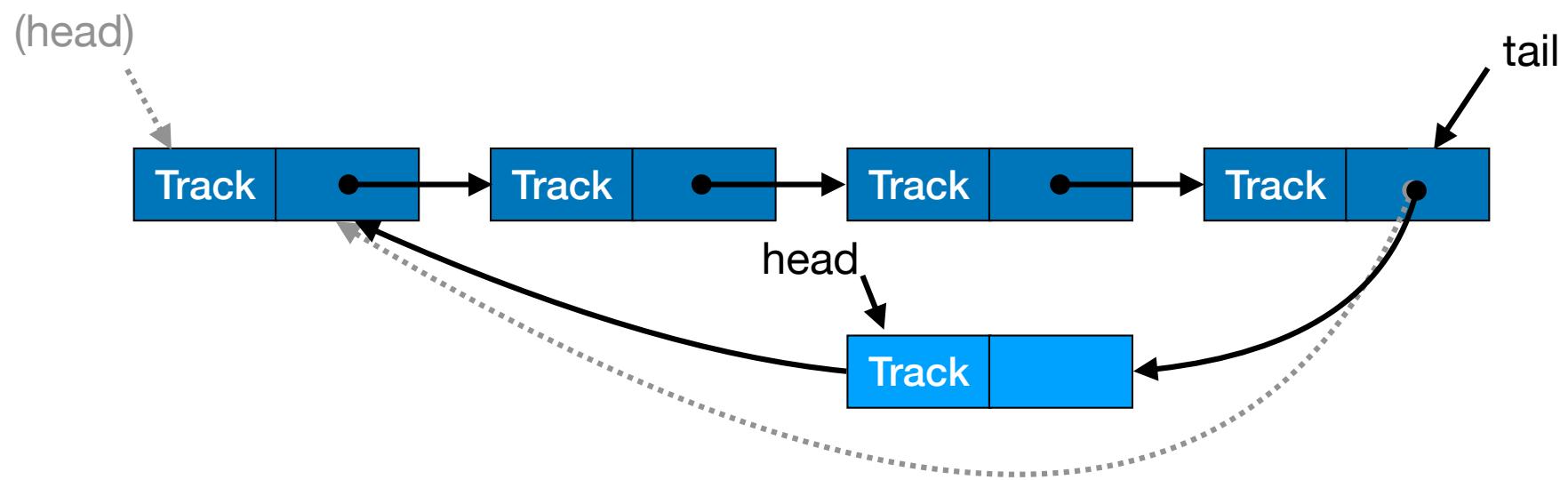
# Rotation d'une “playlist”

rotate()



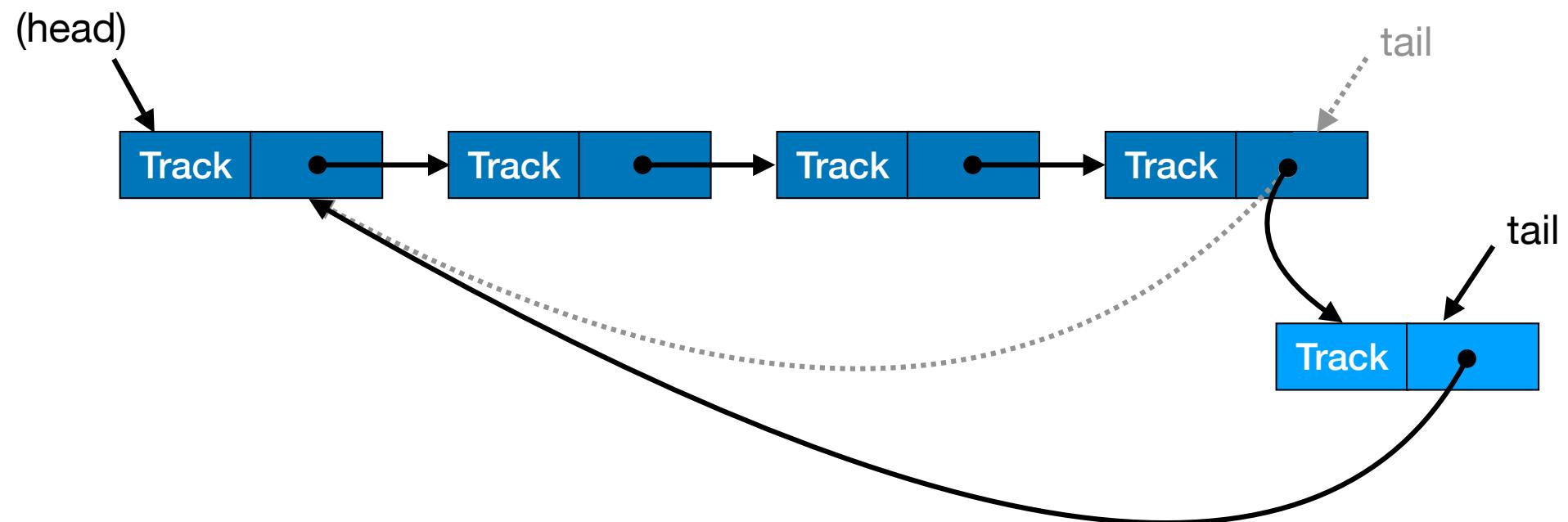
# Ajouter une Track à une “playlist”

`addFirst( Track t )`



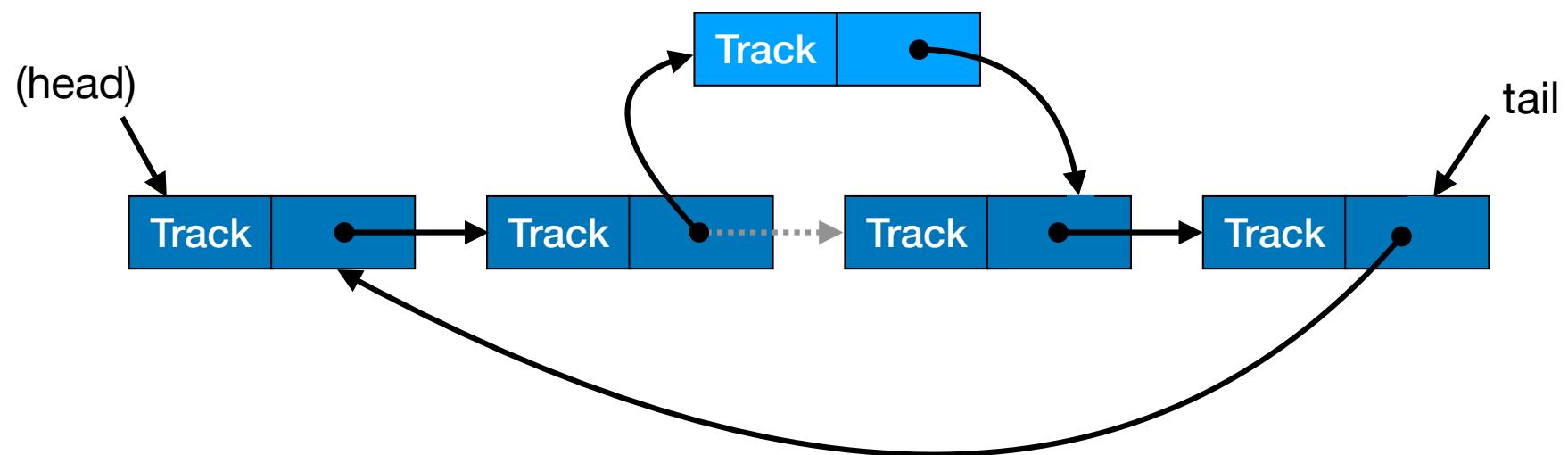
# Ajouter une Track à une “playlist”

`addLast( Track t )`



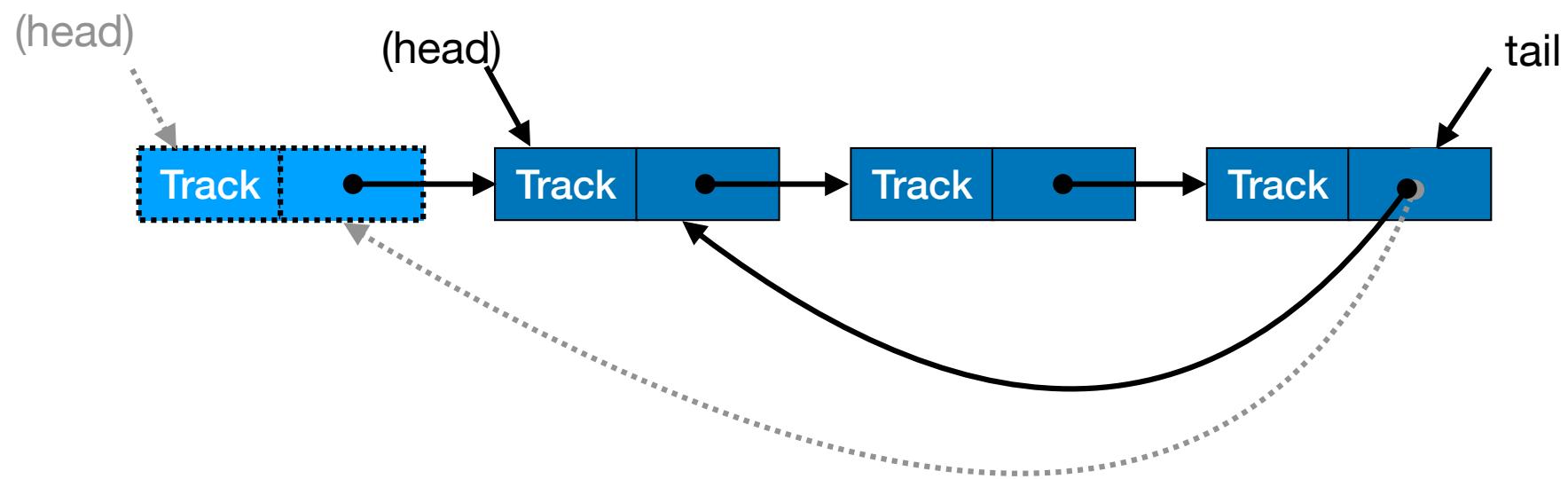
# Ajouter une Track à une “playlist”

`add( int i, Track t )`



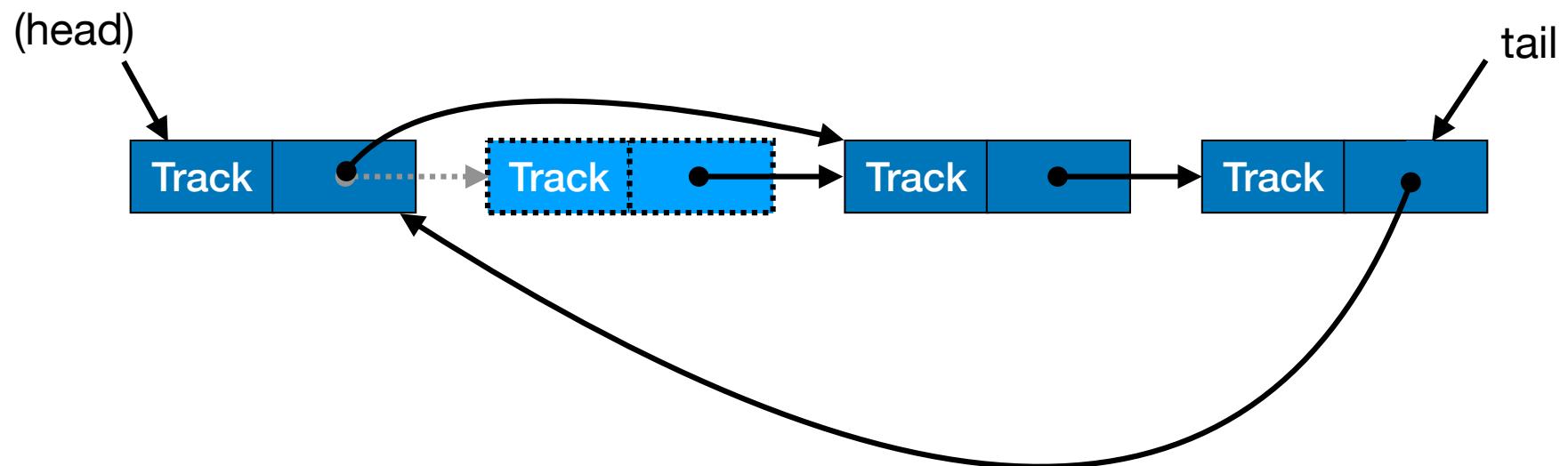
# Retirer une Track d'une "playlist"

`removeFirst()`

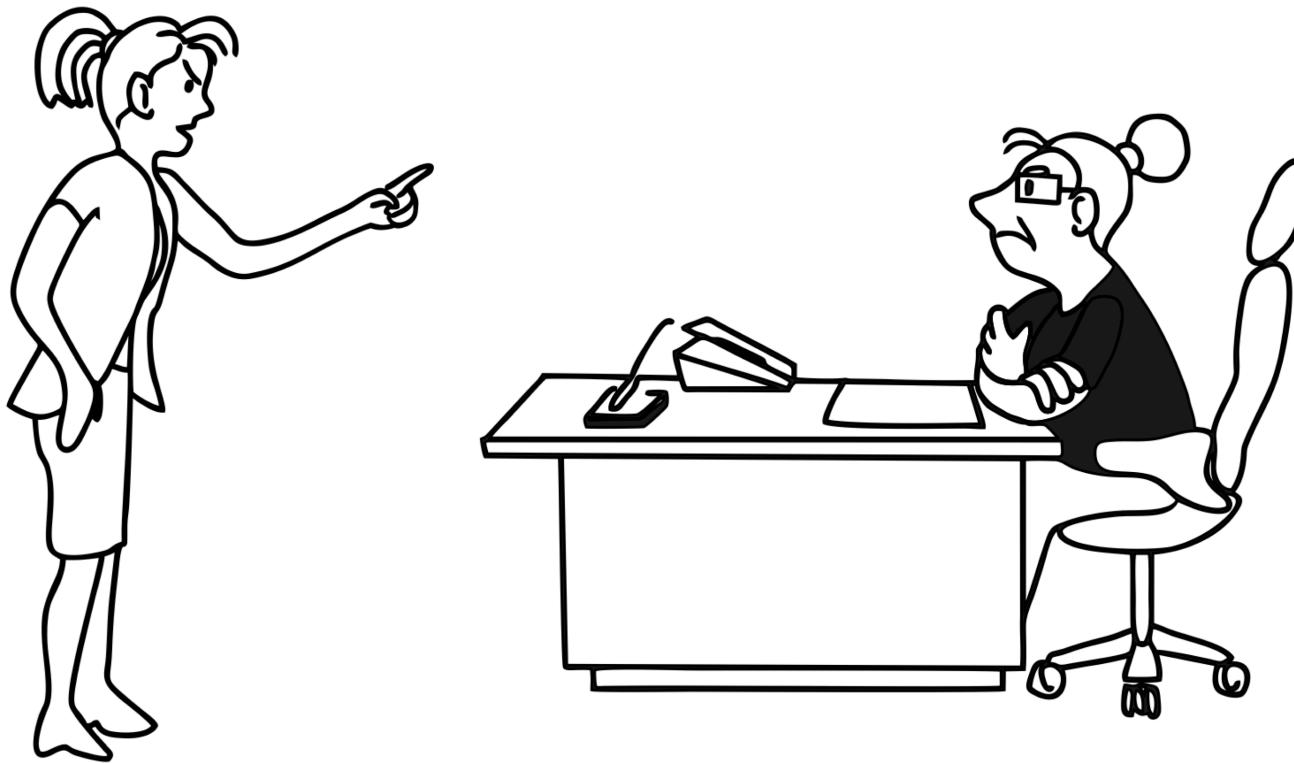


# Retirer une Track d'une "playlist"

```
remove( int i )
```



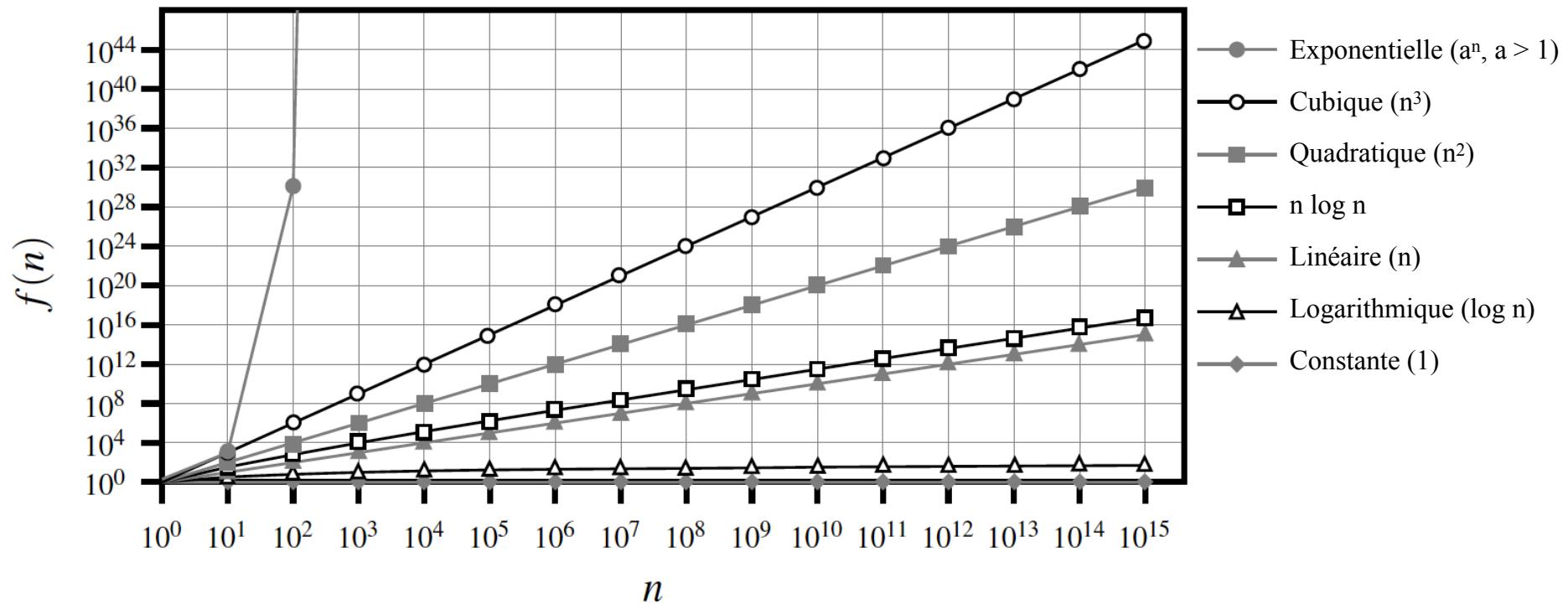
### 3. Outils d'analyse



**“I can’t find an efficient algorithm, because no such algorithm is possible!”**

**“Je ne peux pas trouver un algorithme efficace, parce qu’aucun n’est possible !”**

# Les 7 fonctions importantes



Les taux de croissance pour les 7 fonctions fondamentales utilisées dans l'analyse algorithmique. La base  $a = 2$  est utilisée pour la fonction exponentielle. Les fonctions sont tracées sur un graphique log-log pour comparer les taux de croissance principalement en tant que pentes. La fonction exponentielle se développe trop rapidement pour afficher toutes ses valeurs sur le graphique.

# Temps d'exécution souhaitables

Idéalement, on voudrait que toutes les opérations sur nos structures de données s'exécutent dans des temps proportionnels aux fonctions “constante” ou “logarithmique”, et que nos algorithmes s'exécutent en temps linéaire ou en  $n \log n$ .

Les algorithmes avec des temps d'exécution quadratiques ou cubiques sont moins pratiques, et les algorithmes avec des temps d'exécution exponentiels sont irréalisables sauf pour des jeux d'entrées très petites.

On estime la fonction de croissance d'un algorithme à partir de son code ou pseudo-code en comptant le nombre d'opérations primitives.

# Qu'est-ce qu'une opération primitive

- Les calculs de base exécutés par un algorithme et identifiables dans le code ou pseudo-code.
  - Définition précise non importante car les constantes propres associées à chacune sont éliminées dans le calcul final
  - On assume que ces opérations prennent un temps constant. Par exemple, additionner 2 nombres est souvent plus rapide que multiplier 2 nombres, mais on calcule 1 pour ces 2 opérations.
- Examples:
    - Évaluer une expression
    - Assigner une variable
    - Indexer dans un tableau
    - Appeler une méthode
    - Terminer une méthode

# Compter les opérations primitives

En inspectant le pseudocode, on peut déterminer le nombre maximum d'opérations primitives exécutées par un algorithme, en fonction de la taille d'entrée,  $n$  :

```

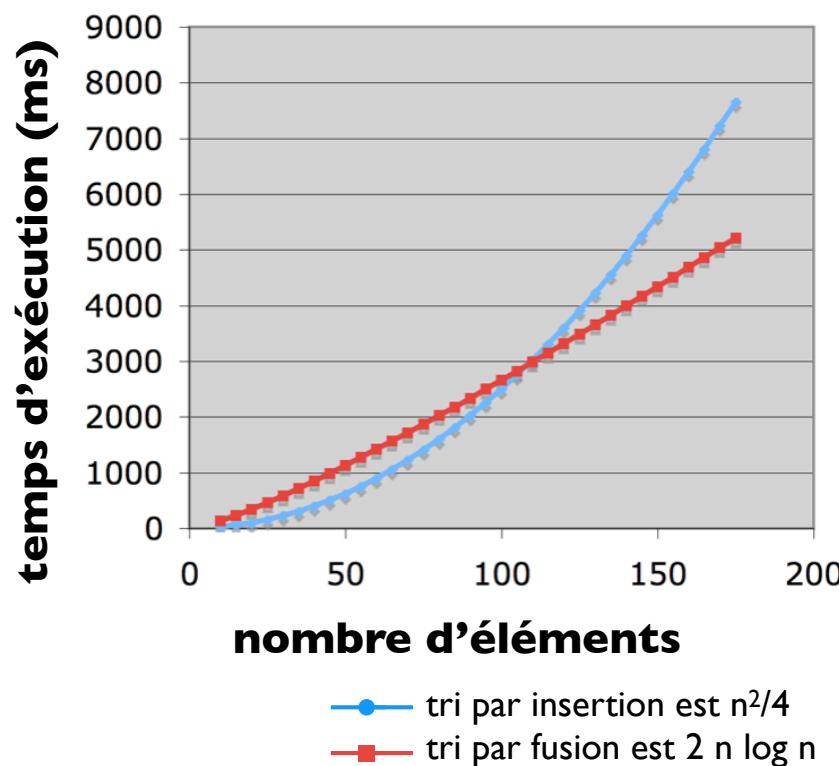
1 def trouve_max( data ):
2     """Retourne l'élément maximum d'une liste Python non vide."""
3     max = data[0]      #valeur initiale à battre
4     for _ in data:    #pour tous les éléments
5         if _ > max: #si il est plus grand que max
6             max = _   #on a trouvé un nouveau max
7     return max       #quand la boucle termine, max est l'élément maximum

```

2 opérations  
2 opérations  
2n opérations  
3n opérations  
0 à 2n opérations  
2 opérations

- **trouve\_max** exécute  $7n + 6$  opérations primitives dans le pire cas et  $5n + 6$  dans le meilleur des cas. Définissons :
  - $a$  = Temps d'exécution de l'opération primitive la plus rapide
  - $b$  = Temps d'exécution de l'opération primitive la plus lente
- Prenons  $T(n)$  comme étant le temps du pire cas de **trouve\_max**. Alors
 
$$a(5n + 6) \leq T(n) \leq b(7n + 6)$$
- Le temps d'exécution,  $T(n)$ , est borné par deux fonctions linéaires =>  $O(n)$

# Comparaison de 2 algorithmes



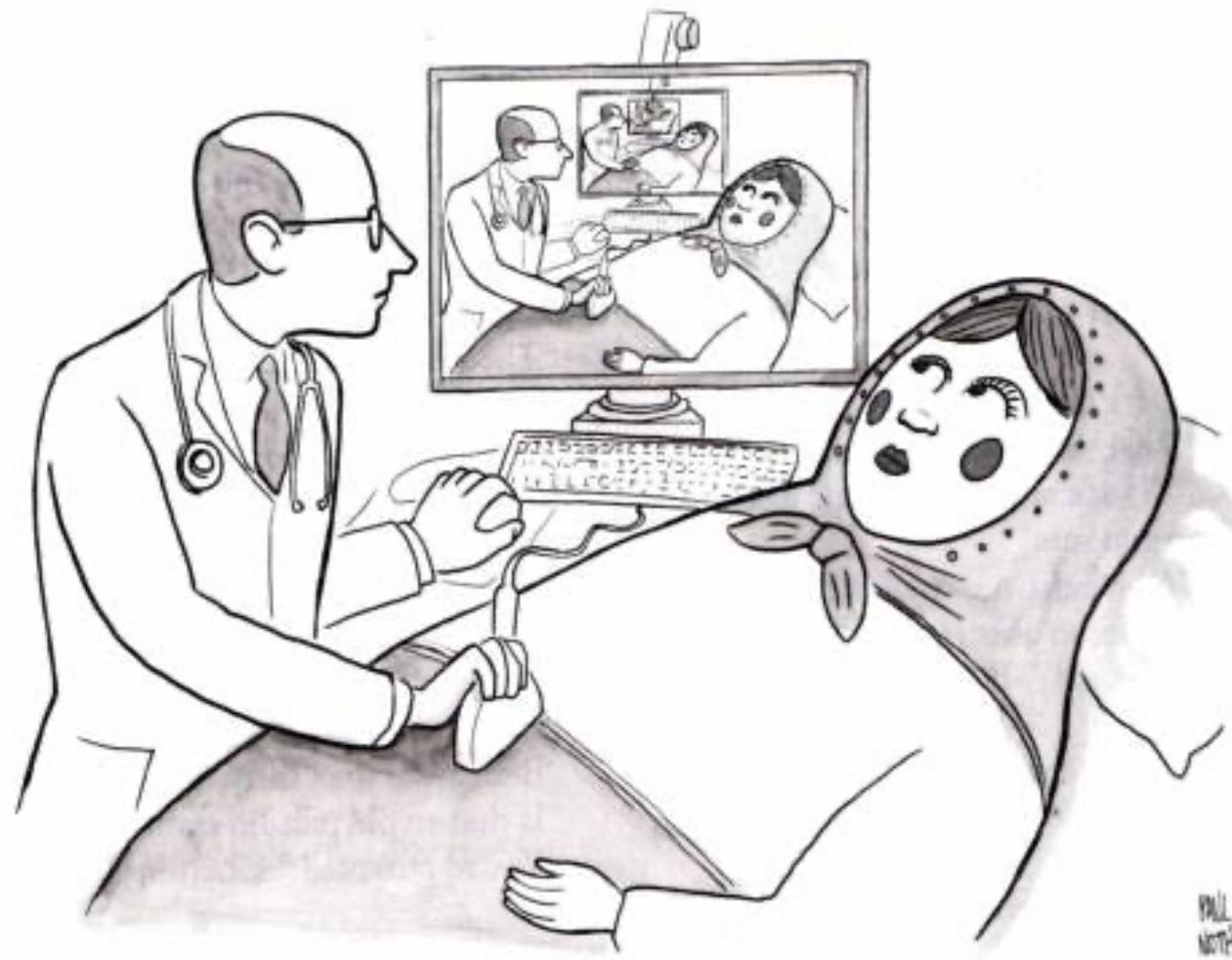
Trier un million d'éléments par insertion prend environ **70 heures** alors que par fusion prend **40 secondes**.

Sur une machine 100 x plus rapide, on aura **40 minutes** versus **0.5 seconde**.

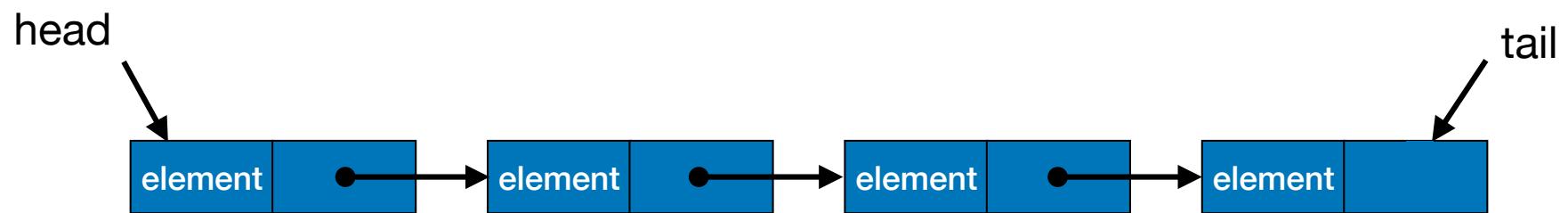
## Taux de croissance, complexité des opérations et structures de données

- Le hardware ou l'environnement du système affecte  $T(n)$  par un facteur constant (les constantes  $a$  et  $b$  dans l'exemple), mais ne change pas le taux de croissance,  $T(n)$
- Le taux de croissance du temps d'exécution,  $T(n)$ , est une propriété intrinsèque de l'algorithme `trouve_max`
- Chaque algorithme possède une propriété d'exécution et d'utilisation mémoire qui lui sont propres et qui sont définies selon l'univers dans lequel il est exécuté
- Les propriétés d'exécution et d'occupation mémoire des meilleurs algorithmes pour solutionner des problèmes peuvent servir à déterminer leur complexité et ainsi à les classer par difficultés de résolution
- La complexité de résolution d'un problème lui est inhérente. En considérant un grand nombre d'instances, on ne peut pas trouver un élément dans une séquence non triée en moins que  $O(n)$  ; si la séquence est triée, on peut le faire en  $O(\log n)$  ; on ne peut pas trier un ensemble d'éléments en les comparant 2 à 2 en moins que  $O(n \log n)$ .
- Une opération sur une structure de données est un problème à résoudre et possède sa propre complexité. Cette complexité change en fonction de la structure de données.

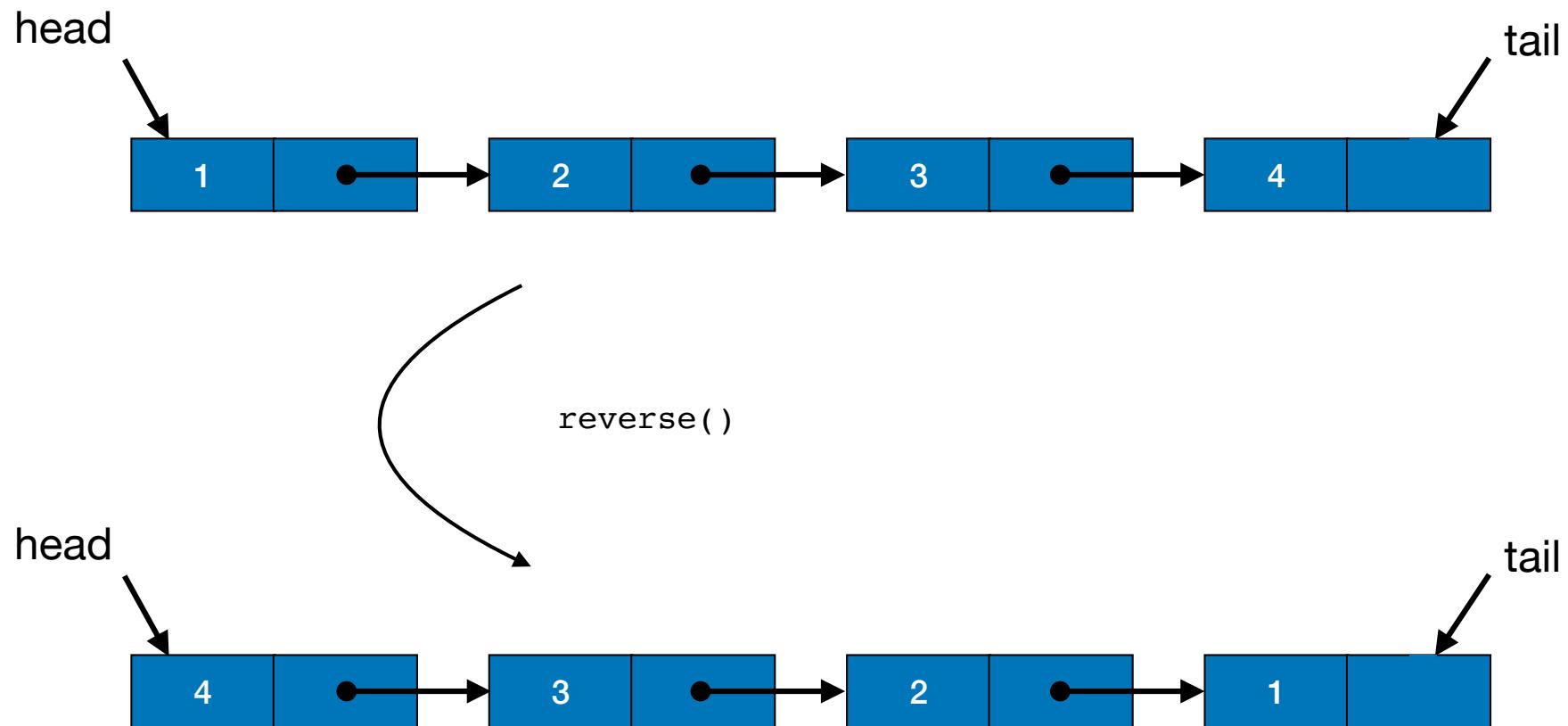
## 4. Récursivité



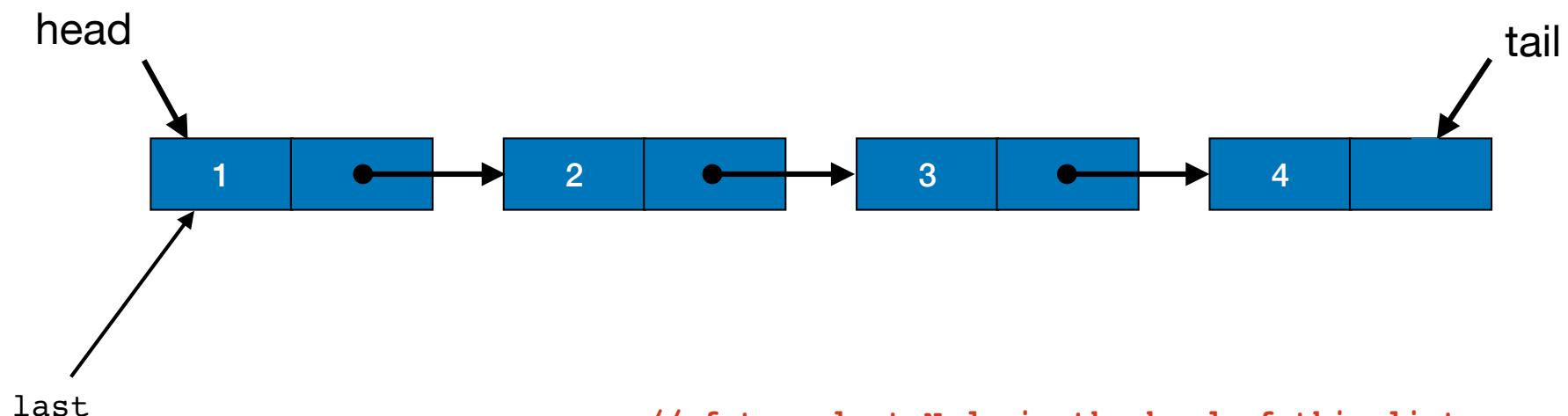
# SinglyLinkedList



## SinglyLinkedList<Integer> recursiveReverse()

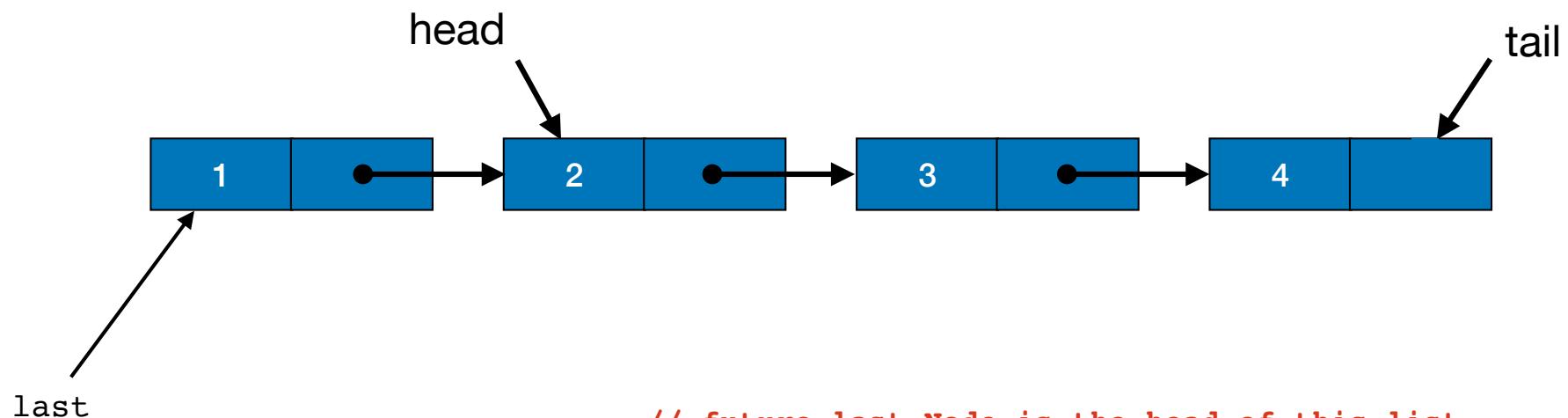


## SinglyLinkedList<Integer> recursiveReverse()



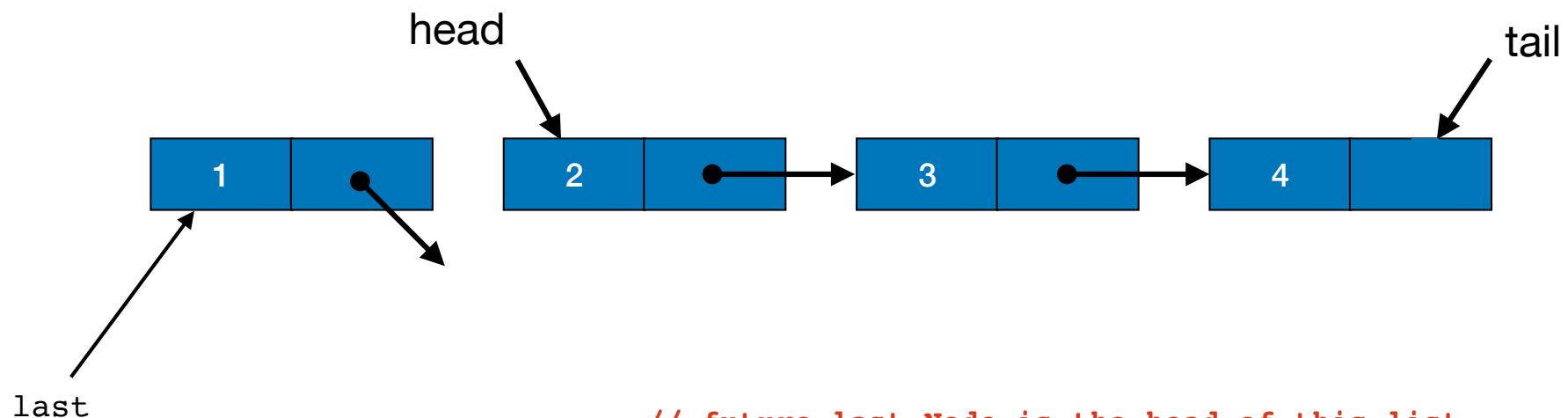
```
// future last Node is the head of this list
Node<E> last = this.getHead();
// remove the head of this list
this.head = last.getNext();
this.size--;
// next of the future last Node is null
last.next = null;
```

## SinglyLinkedList<Integer> recursiveReverse()



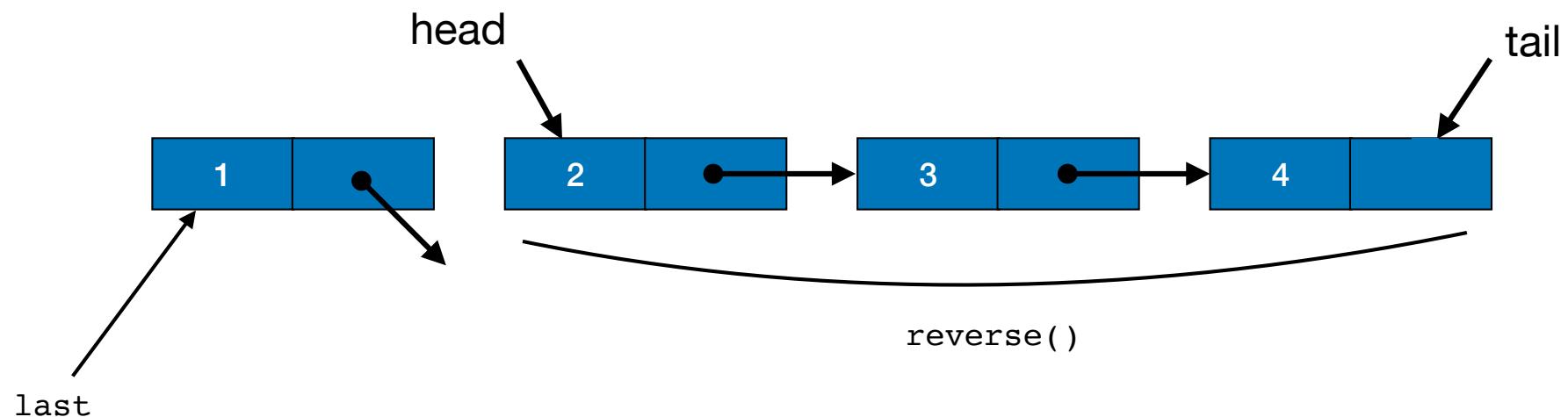
```
// future last Node is the head of this list
Node<E> last = this.getHead();
// remove the head of this list
this.head = last.getNext();
this.size--;
// next of the future last Node is null
last.next = null;
```

## SinglyLinkedList<Integer> recursiveReverse()



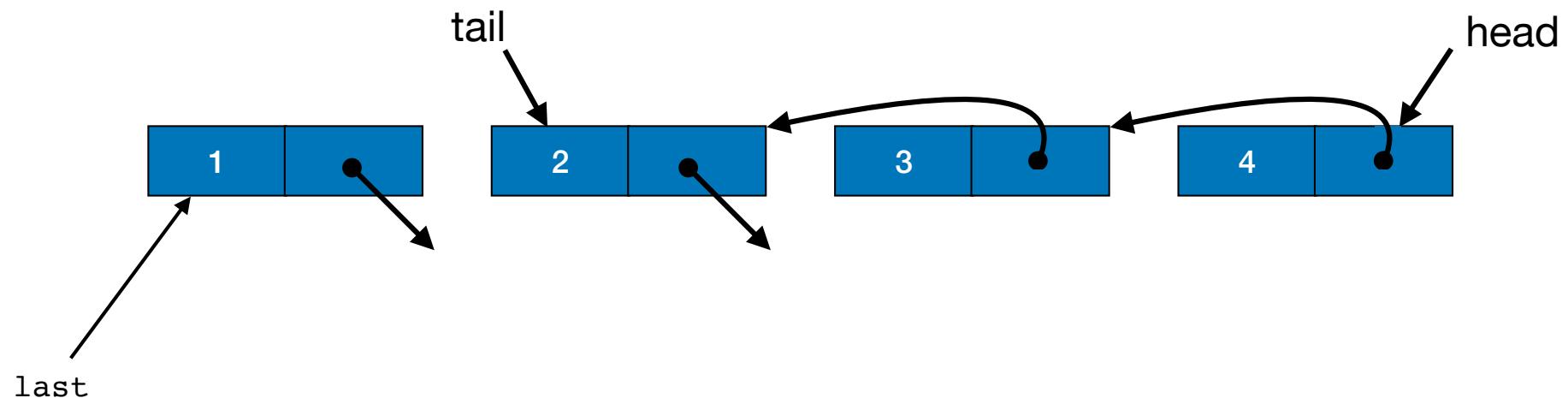
```
// future last Node is the head of this list
Node<E> last = this.getHead();
// remove the head of this list
this.head = last.getNext();
this.size--;
// next of the future last Node is null
last.next = null;
```

## SinglyLinkedList<Integer> recursiveReverse()



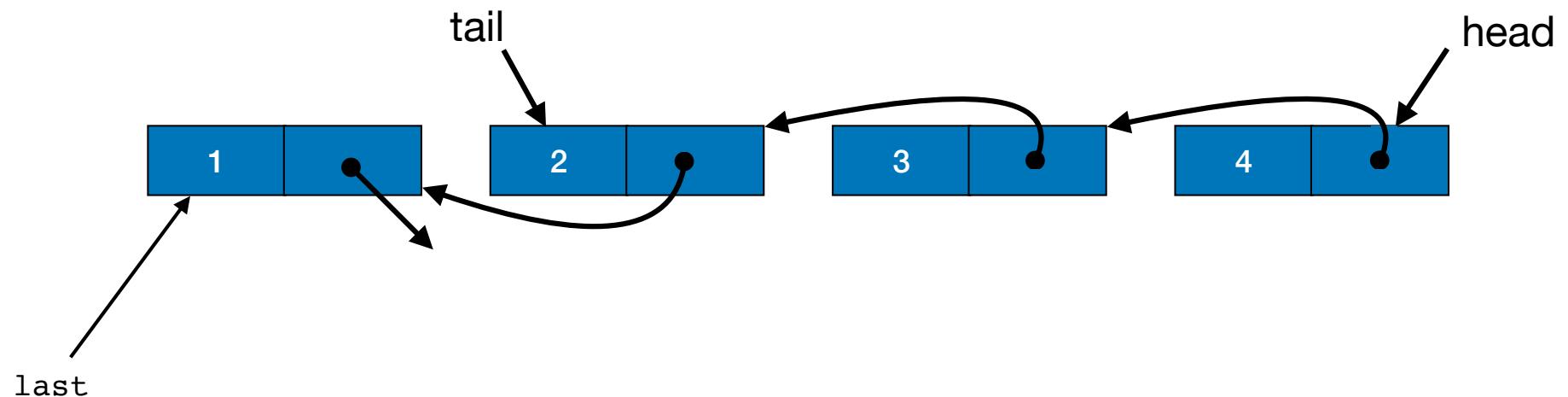
```
// reverse the rest of this list (this list minus old head)  
this.recursiveReverse();
```

## SinglyLinkedList<Integer> recursiveReverse()



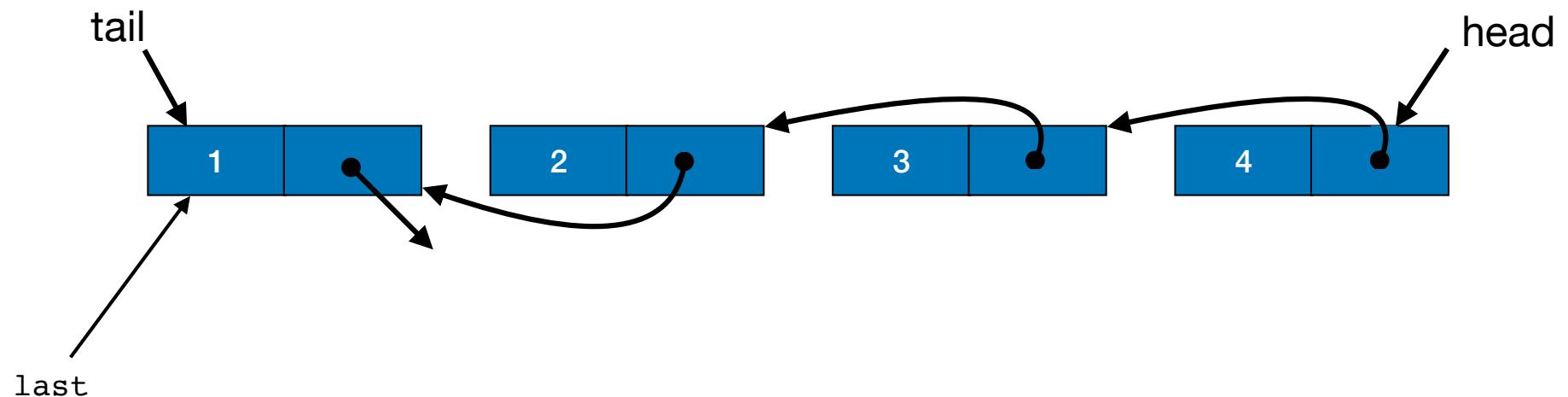
```
// reverse the rest of this list (this list minus old head)  
this.recursiveReverse();
```

## SinglyLinkedList<Integer> recursiveReverse()



```
// add the old head as the last
this.tail.next = last;
this.size++;
// set the new tail to the last Node
this.tail = last;
```

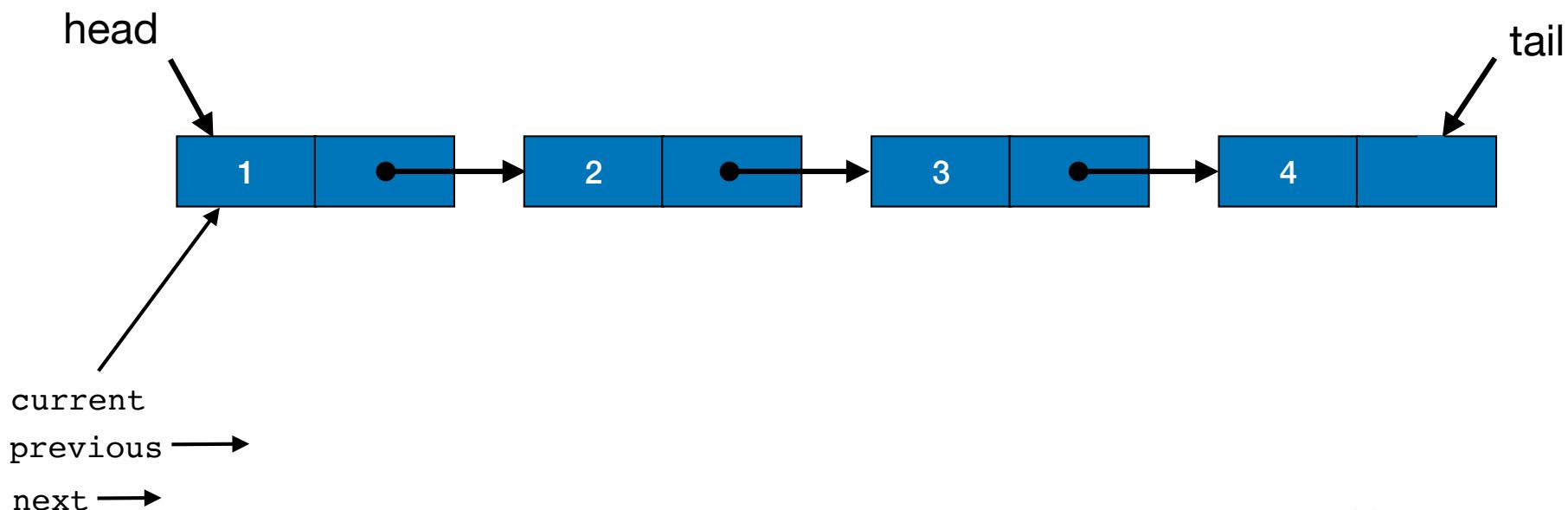
## SinglyLinkedList<Integer> recursiveReverse()



```
// add the old head as the last
this.tail.next = last;
this.size++;
// set the new tail to the last Node
this.tail = last;
```

## SinglyLinkedList<Integer>

### reverse()



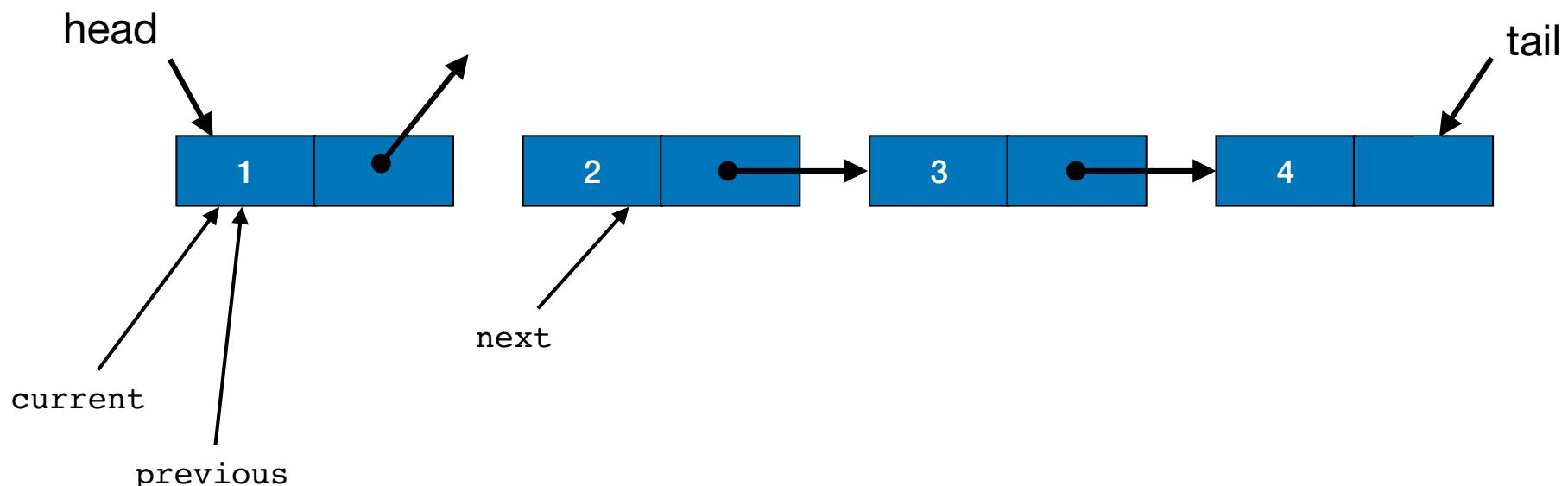
```

public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}

```

## SinglyLinkedList<Integer>

### reverse()

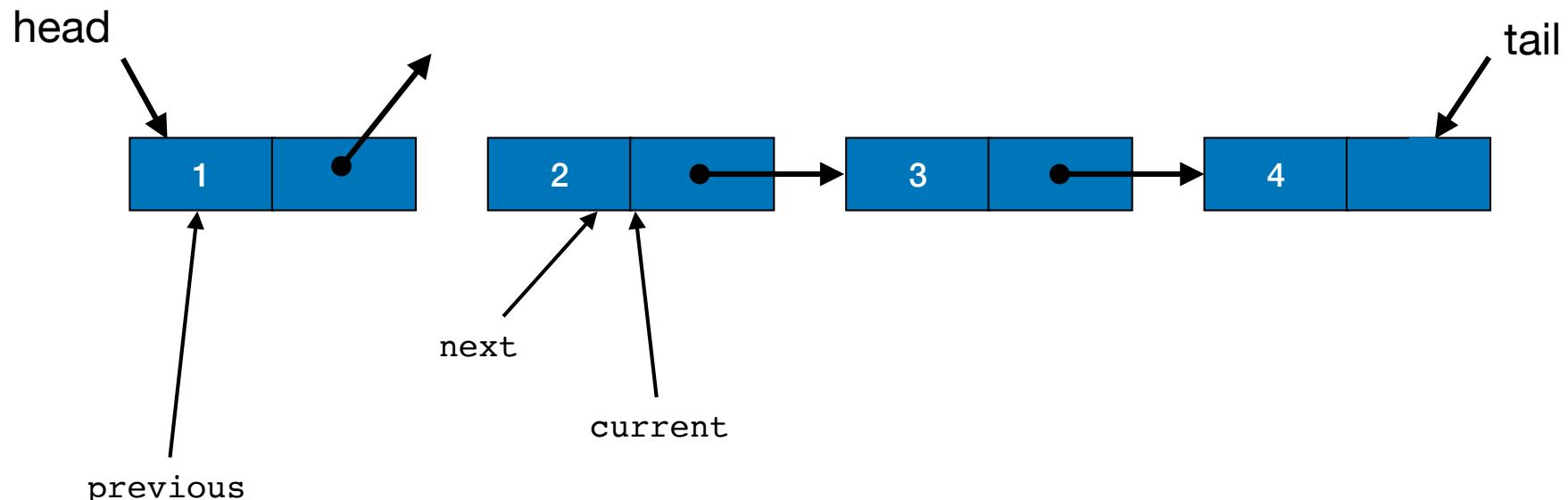


```

public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}

```

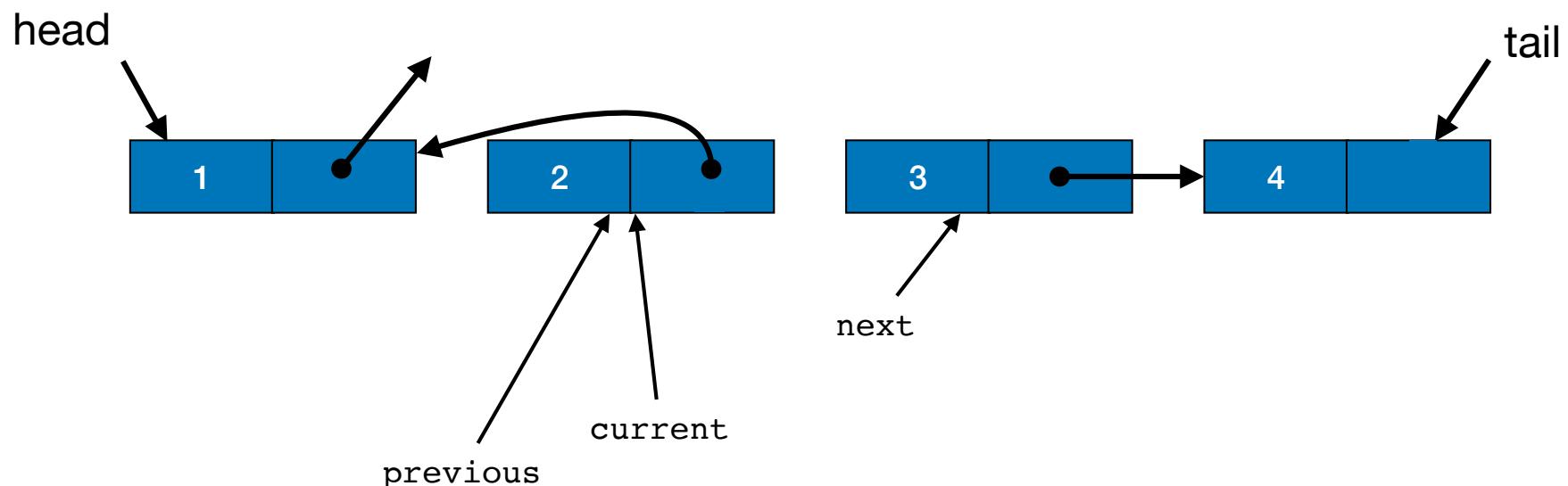
## SinglyLinkedList<Integer> reverse()



```
public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}
```

## SinglyLinkedList<Integer>

### reverse()



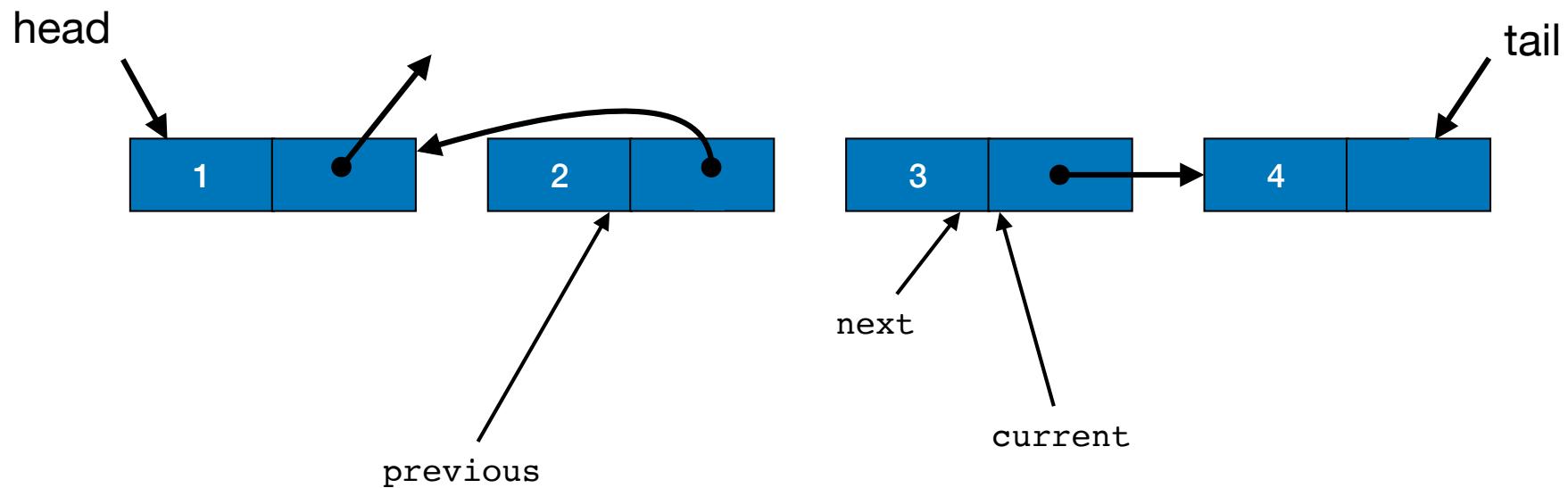
```

public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}

```

## SinglyLinkedList<Integer>

### reverse()



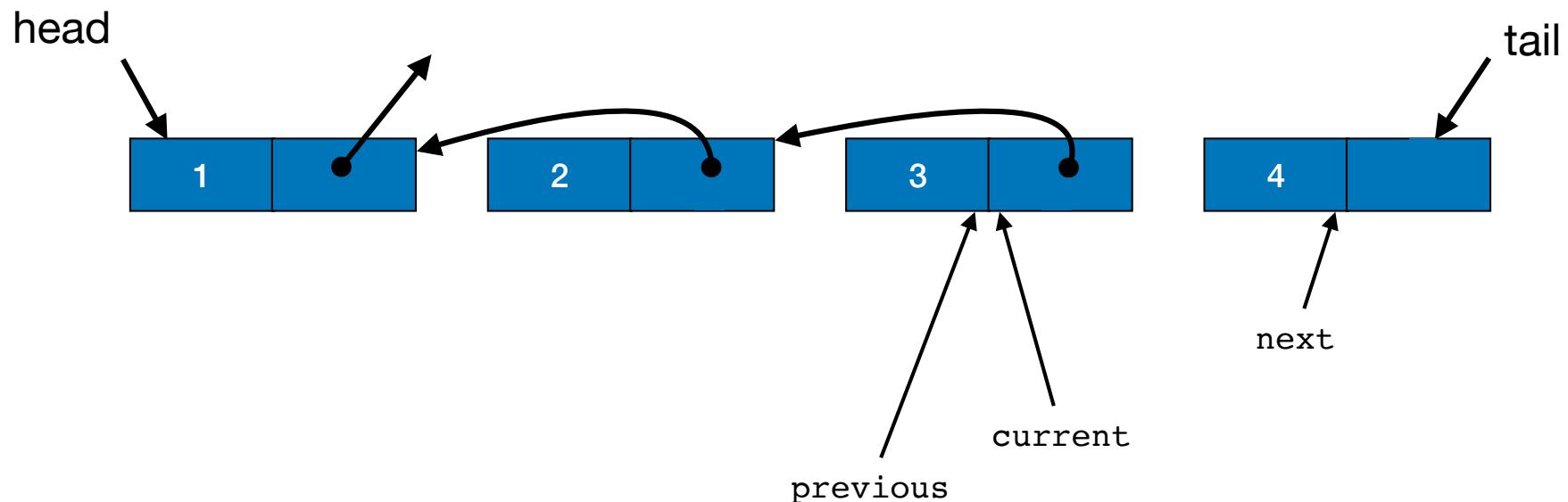
```

public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}

```

# SinglyLinkedList<Integer>

## reverse()



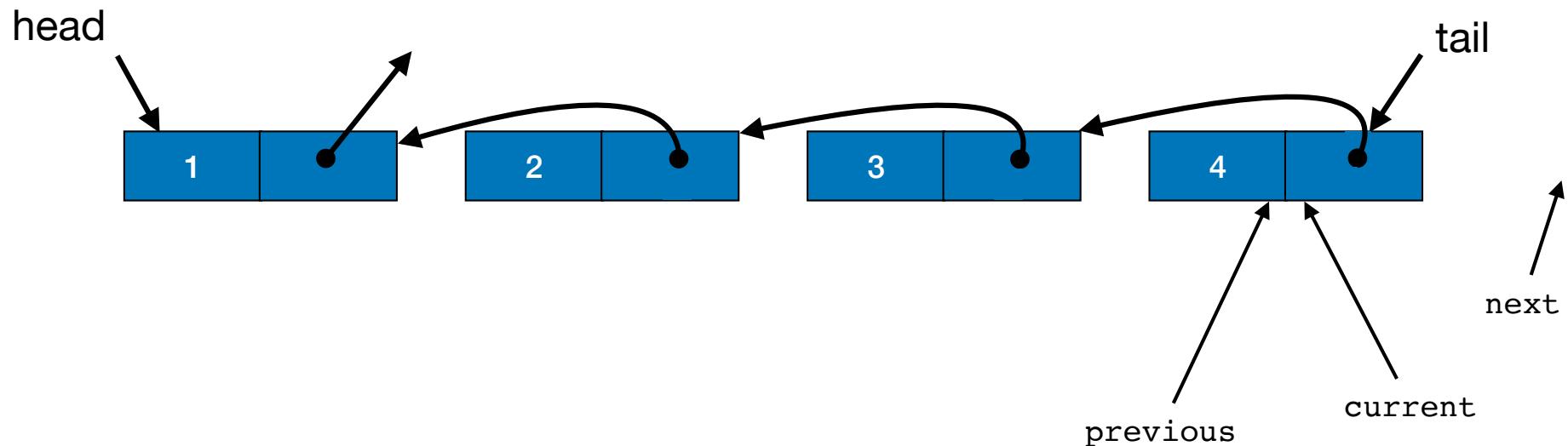
```

public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}

```

# SinglyLinkedList<Integer>

## reverse()



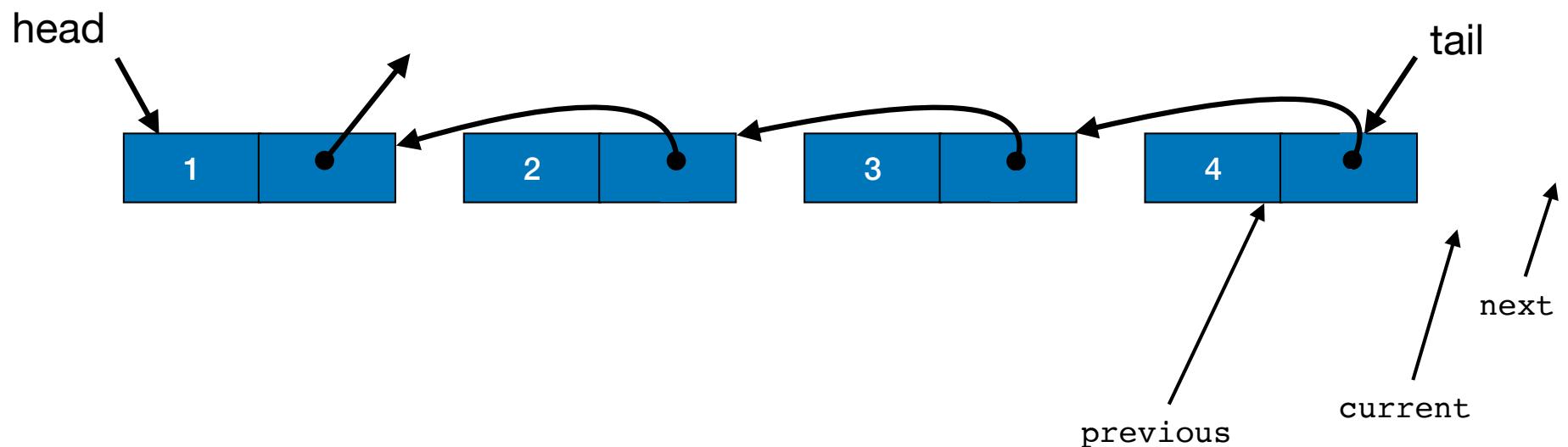
```

public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}

```

## SinglyLinkedList<Integer>

### reverse()



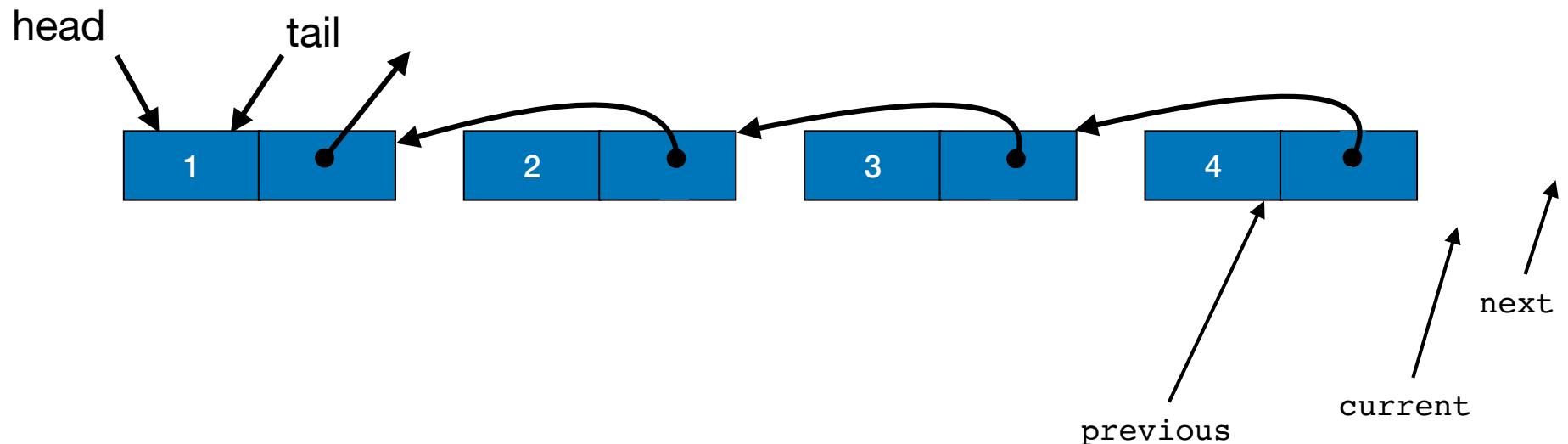
```

public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}

```

## SinglyLinkedList<Integer>

### reverse()



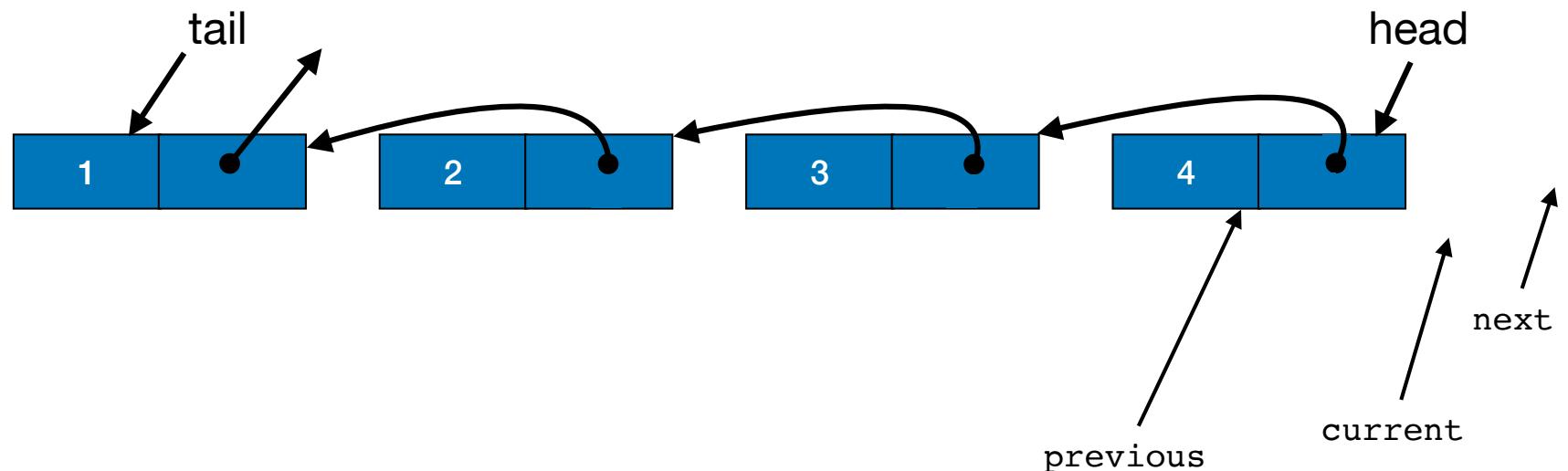
```

public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}

```

## SinglyLinkedList<Integer>

### reverse()



```

public void reverse() { // iterative version
    Node<E> current = this.getHead();
    Node<E> previous = null;
    Node<E> next = null;
    while( current != null ) {
        next = current.getNext();
        current.next = previous;
        previous = current;
        current = next;
    }
    this.tail = this.head;
    this.head = previous;
}

```