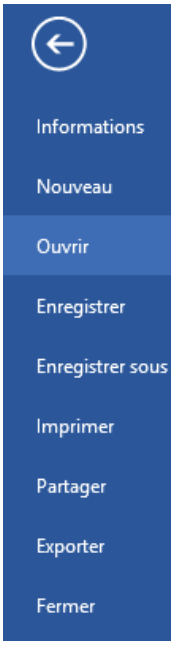
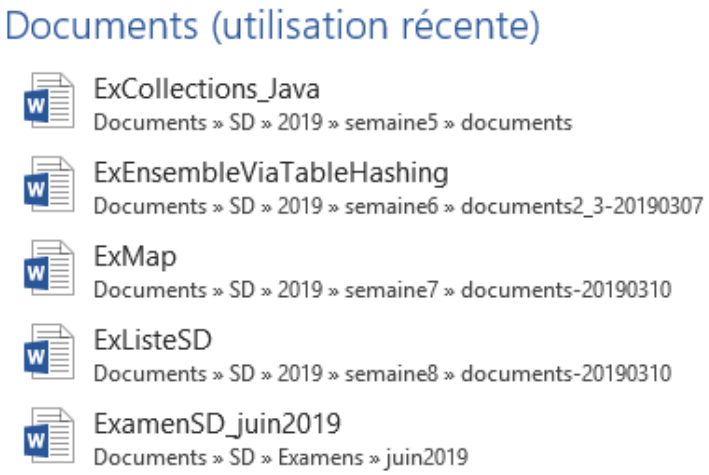
A Mécanisme de LRU (7 points)

Le menu *ouvrir* de Word présente les documents récemment utilisés.

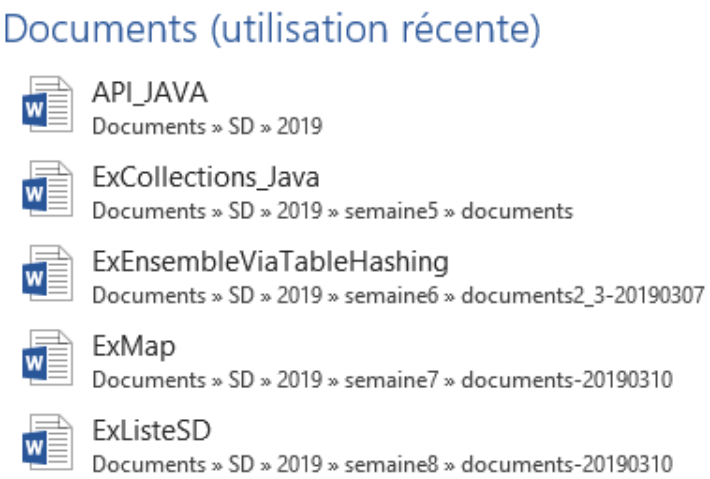


Exemple :



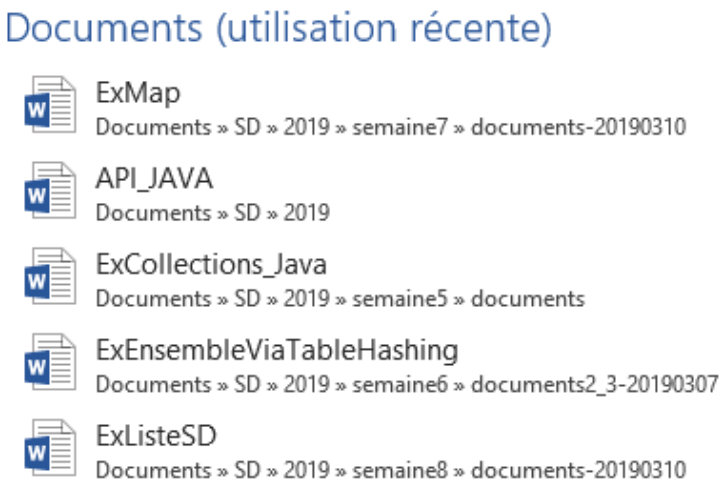
L'ordre d'apparition est important. C'est le document *ExCollections_Java* qui apparait en premier, car il est le document qui a été ouvert le plus récemment. Ensuite c'est *ExEnsembleViaTableHashing*, l'avant dernier à avoir été ouvert et ainsi de suite.

Le fait d'ouvrir un document qui n'était pas encore présent dans la liste a comme conséquence que celui-ci apparaît maintenant en premier lieu (MRU – Most Recently Used) et que le dernier document de la liste a disparu.
Ce document est celui qui a été le moins récemment utilisé (LRU – Least Recently Used).



Dans l'exemple, le document *API_JAVA* vient d'être ouvert. Il apparaît en premier lieu.
Le document *ExCollections_Java* apparait maintenant en deuxième, …
Le document *ExamenSD_juin2019* n'apparait plus.

Si on ouvre un document qui se trouve déjà dans la liste, il est déplacé en premier.
Il devient le document le plus récemment utilisé.

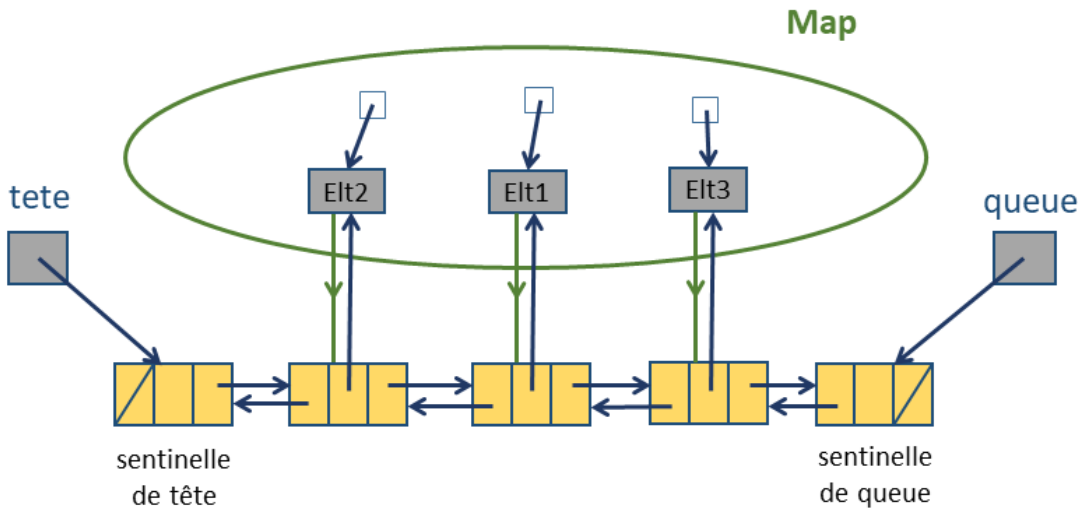


Dans l'exemple, le document *ExMap* vient d'être ouvert. Il apparaît en premier lieu.

Pour l'examen, nous allons nous intéresser à la structure de données qui permet d'optimiser le mécanisme de **LRU**.

A1 ListeLRU

Les documents vont être placés dans une liste sans doublon.
Voici l'implémentation choisie.

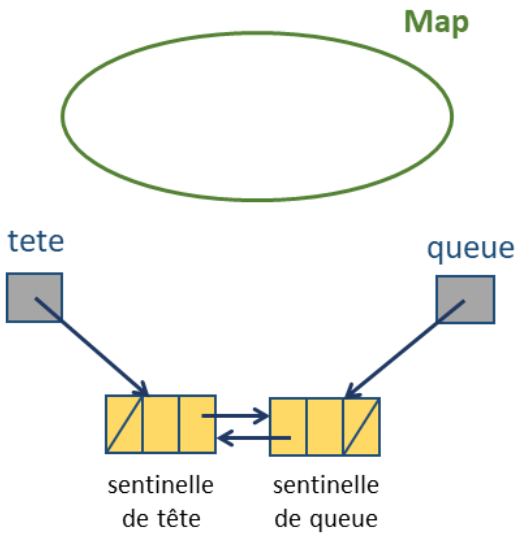


Cette implémentation utilise une liste doublement chaînée avec sentinelles et un *map*

Une amélioration bien connue de l'implémentation de liste est l'ajout de faux éléments.
L'ajout de ces faux éléments évite de devoir tester de nombreux cas particuliers.
Dans le cas d'une liste doublement chaînée, on ajoute 2 faux éléments. Ces faux éléments sont placés en tête et en queue de liste dans des nœuds appelés « **sentinelle** » ou encore « bidon ».
Les éléments contenus dans ces nœuds n'appartiennent pas à la liste.

Plusieurs méthodes doivent accéder au nœud contenant l'élément passé en paramètre.
Pour récupérer ce nœud, on pourrait parcourir la liste. Mais un parcours de liste se fait en O(N) !
L'utilisation d'un *map* permet la recherche d'un nœud en O(1).
La clé est l'élément recherché, la valeur associée est le nœud contenant cet élément.

Au départ, la liste est vide, mais contient 2 nœuds sentinelles :



Nous vous demandons de compléter la classe *ListeLRU*.
Elle contient les méthodes utiles pour implémenter le mécanisme de LRU.
Testez votre classe à l'aide de la classe *ListeLRU*.

A2 DocumentsLRU

Nous vous demandons de compléter la classe *DocumentsLRU*.

Cette classe possède une liste de documents (String).
Pour la liste, vous utiliserez un objet de la classe *ListeLRU*.

Cette liste est préremplie avec autant de documents que nécessaires pour atteindre la taille que l'on a décidée préalablement et qu'on ne peut pas dépasser.
Comme, au départ, il n'y a pas de document, la liste contiendra des documents bidon aisément reconnaissables.

Lorsqu'on ouvre un document qui ne se trouve pas dans la liste, il est placé en début de liste et il faut retirer celui qui se trouve en fin de liste.
Lorsqu'on ouvre un document se trouvant dans la liste, celui-ci devient le plus récent.
Il faut donc le déplacer en tête.
La tête de la liste contient donc toujours le document le plus récemment utilisé (Most Rencently Used).

Le jeu de tests de la classe *TestDocumentsLRU* suit le scénario suivant :

```
Au depart :
 doc1 doc2 doc3 doc4 doc5

ouvrir doc3
 doc3 doc1 doc2 doc4 doc5

ouvrir doc4
 doc4 doc3 doc1 doc2 doc5

ouvrir doc4
 doc4 doc3 doc1 doc2 doc5

ouvrir doc5
 doc5 doc4 doc3 doc1 doc2

ouvrir doc6
 doc6 doc5 doc4 doc3 doc1

ouvrir doc3
 doc3 doc6 doc5 doc4 doc1

ouvrir doc6
 doc6 doc3 doc5 doc4 doc1

ouvrir doc7
 doc7 doc6 doc3 doc5 doc4
```

B API JAVA : Application Concours (7 points)

De nombreux concours (The Voice, l'Eurovision, …) demande la participation du public pour élire un gagnant parmi les candidats inscrits.
Sur un temps de laps limité, les spectateurs sont invités à voter pour leurs candidats favoris via téléphone. Généralement, le nombre de votes par spectateur est limité.
Le candidat qui gagne est celui qui aura obtenu le plus de votes.

Voici l'implémentation qui a été choisie :

La classe *Candidat* vous est donnée.

Tout candidat possède un numéro, un nom et un nombre de votes.
La classe possède la méthode `ajouter1Vote()`

La classe *Concours* est à compléter.

Tous les candidats sont placés dans la table **`tableCandidats`**.
Le candidat qui se trouve à l'indice 0 porte le numéro 1, le candidat qui se trouve à l'indice 1 porte le numéro 2 et ainsi de suite …

Le *map* (*HashMap<String,Candidat>*) **`mapCandidats`** permet de retrouver facilement un candidat via son nom.

Le constructeur de la classe *Concours* reçoit comme paramètres le nombre de votes maximum qu'un votant peut faire et une table (*String*) avec les noms des candidats.

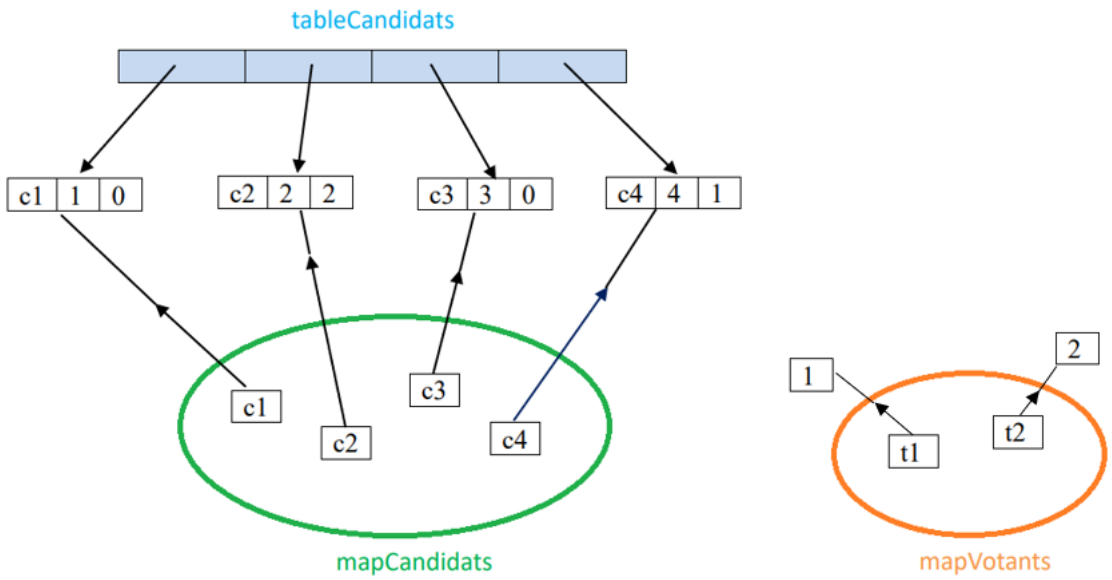
Ex :

« c1 »	« c2 »	« c3 »	« c4 »

Le 1er candidat portera le numéro 1, le 2ème portera le numéro 2 et ainsi de suite.
Au départ les candidats n'ont reçu aucun vote.
Le constructeur crée les différents candidats et les place dans la table et le *map*.
Ne perdez pas de vue que la table des candidats et le *map* **référencient** des mêmes candidats.

L'attribut **`nombreMaxVotes`** retient le nombre de votes maximum autorisé par spectateur.
Nous n'allons pas introduire de classe *Spectateur*.
Le spectateur exprime son choix via téléphone. Un spectateur sera reconnu via son numéro de téléphone (*String*).
Le *map* (*HashMap<String,Integer>*) **`mapVotants`** permet de connaître le nombre de votes obtenu par numéro de téléphone.
Seuls les numéros de téléphone qui ont au moins obtenu un vote y sont retenus.
(Attention : la méthode **`get()`** renvoie un *Integer* et non un *int*. Un *cast* sera peut-être nécessaire dans votre code)

Exemple :



Dans cet exemple, on remarque qu'il y a 4 candidats : c1, c2, c3 et c4.
Ils portent respectivement les numéros 1, 2, 3 et 4.
c2 a obtenu 2 votes et c4 en a obtenu 1.
2 spectateurs ont voté. t1 a voté 2x et t2 a voté 1x.

La méthode `toString()` de la classe *Arrays* a été utilisée dans de nombreuses classes pendant l'année.
Dans la classe *Concours*, elle est utilisée dans la méthode `toString()` !
Pour compléter la méthode `classement()`, on vous demande d'utiliser les méthodes *static* `copyOf()` et `sort()` de la classe *Arrays*.
La méthode `sort()` modifie la table passée en paramètre. Il est donc indispensable de travailler sur une copie de la table des candidats.
La méthode `sort()` demande un comparateur. Fournissez-lui un objet de la classe *ComparateurCandidats*. Cette classe vous est donnée. Les candidats y sont comparés selon leur nombre de votes.
Pour mieux connaitre les méthodes `copyOf()` et `sort()`, consultez la *JavaDoc*. Des extraits se trouvent dans le document *ArraysExtraitsJavaDoc*.

Complétez la classe *Concours* en respectant bien la *JavaDoc* et les choix d'implémentation imposés ci-dessus.
Vous devrez compléter le constructeur et les méthodes `voterViaNom()`, `voterViaNumero()` et `classement()`.
Si c'est plus facile pour vous, vous pouvez introduire d'autres attributs et des méthodes (*private*)

La classe *GestionConcours* va vous servir pour tester la classe *Concours*.
Vous pouvez la modifier.
Ne perdez pas de temps à l'améliorer. Cette classe ne sera pas évaluée.

Lors de vos tests, pour éviter de devoir chaque fois encoder toute une séquence de commande identique, on vous conseille de les encoder dans un fichier. La classe *MonScanner* permet de passer de l'encodage via fichier à l'encodage manuel. On vous a fourni le fichier *input.txt* avec déjà quelques commandes. En utilisant ce fichier, le « concours » correspondra à l'exemple ci-dessus.

C ABR (6 points)

Pour les arbres, vous avez eu l'habitude d'écrire les méthodes de façon récursive.
Mais toutes ces méthodes peuvent s'écrire de façon itérative à condition de disposer d'un itérateur (via un *foreach*)

C1 Parcours d'arbres.
Complétez « à la main » le document qui vous a été remis.

C2 Vous allez compléter la classe *ABRDEntiers*.

Chaque nœud de cet arbre binaire de recherche contient un entier.
Dans cet arbre, la descendance gauche d'un nœud ne contient que des entiers strictement plus petits que l'entier de ce nœud et sa descendance droite ne contient que des entiers égaux ou plus grands.

La classe interne *Iterateur* est presque complète.
Elle possède un attribut : une file d'entiers (*ArrayDeque<Integer>).*
Le constructeur de la classe s'occupe de remplir cette file avec tous les entiers contenus dans l'arbre.
Le but de cet itérateur est de parcourir l'arbre en suivant l'**ordre croissant** des entiers.
Il faut donc « enfiler » les objets dans la file de façon à respecter ce parcours. C'est la méthode `remplirFile()` qui se charge de remplir la file. Plusieurs suites d'instructions sont proposées. A vous de choisir la bonne !
Pour les méthodes `nombreNegatifs()` et `tousPositifs()` , vous écrirez une version itérative et une version récursive.
On vous demande chaque fois de fournir une version la plus optimale possible.
Prenez bien en compte que l'arbre est un arbre de recherche et aussi que l'itérateur parcourt les entiers selon l'ordre croissant.

Vous pouvez ajouter d'autres méthodes.
La classe *TestABRDEntiers* permet de tester les méthodes demandées.
Le document *ABRTestes* donne une visualisation des arbres testés.

C3 Comparaison `tousPositifsVI()` - `tousPositifsVR()`
Complétez « à la main » le document qui vous a été remis.