Une liste chaînée avec des blocs

Dans cet exercice, nous allons programmer une liste : pour l'implémentation, nous allons choisir un compromis entre le tableau et la liste chaînée. En effet, pour améliorer les performances (en particulier les accès mémoire en cache), nous allons stocker quelques valeurs dans un même nœud. Les nœuds seront programmés dans une classe interne à votre classe de liste. Vous devrez aussi programmer un itérateur sur votre liste, aussi sous forme de classe interne. Vous aurez donc trois classes programmées dans le même fichier Java :

- La classe principale implémentant l'interface `List<E>` que vous nommerez `BlockLinkedList`.
- La classe interne stockant un nœud, nommée `ArrayNode`.
- La classe interne utilisée pour l'itérateur nommée `BlockLinkedListIterator`.

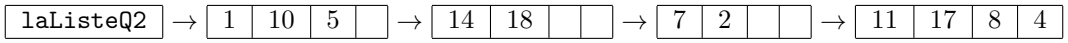
Vous devez **commencer** votre source par un commentaire javadoc précisant au minimum le contexte de la réalisation, l'auteur et la date, avec les bons tags. Vous devez aussi impérativement mettre **avant** chaque classe (interne ou pas) un commentaire javadoc précisant le rôle et les propriétés de la classe que vous implémentez.
Vous avez à votre disposition un fichier de tests, organisé par question, pour vous aider. Ces tests sont uniquement fonctionnels. Ils ne sont pas exhaustifs et ne présagent pas du tout de l'adéquation de votre implémentation, de vos algorithmes à ce qui est demandé. N'hésitez pas donc à ajouter vos propres tests.

Question 1 : Créez la classe `ArrayNode<E>` avec les attributs que vous jugerez nécessaires pour stocker au plus `n` éléments [1] et garantir le chaînage : c'est-à-dire que chaque nœud doit connaître un autre `ArrayNode<E>` qui sera son successeur.

Question 2 : Créez une classe `BlockLinkedList<E>` Vous y ajouterez deux constructeurs :

1. Un constructeur sans paramètre [2] : le nombre maximal d'éléments par nœud est alors 4 par défaut [3].
2. Un constructeur avec une taille spécifiée. Pour simplifier les algorithmes d'ajout et suppression, cette taille **doit** être paire, sinon une exception de type `IllegalArgumentException` sera levée. Tous les nœuds auront alors la même taille maximale dans tous les blocs de la liste.

Voici par exemple, une structure de liste par blocs contenant les valeurs 1, 10, 5, 14, 18, 7, 2, 11, 17, 8 et 4.



Question 3 : Programmez les méthodes de l'interface : `clear`, `isEmpty` et `size`, et redéfinissez aussi la méthode `toString` de telle sorte que l'affichage soit cohérent avec la convention usuelle des collections, c'est-à-dire que la représentation textuelle de la liste ci-dessus serait `[1, 10, 5, 14, 18, 7, 2, 11, 17, 8, 4]`.

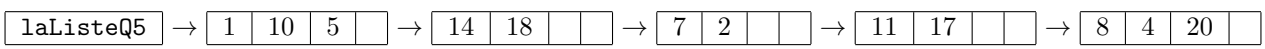
Question 4 :

1. Programmez une méthode privée qui localise un élément [4].
2. Utilisez cette méthode pour implémenter `contains`, `indexOf`, `get` et `set`.
3. Programmez la méthode `containsAll`.

Question 5 : Pour insérer un nouvel élément dans la liste (méthode `add(int index, E element)`), il suffit de trouver le nœud dans lequel il trouve sa place et insérer l'élément dans la tableau. Si le tableau dans le nœud est plein, on insère un nouveau nœud dans lequel, on déplace [5] la moitié des éléments du nœud courant. Programmez les méthodes `add` de l'interface :

- `void add(int index, E element);`
- `boolean add(E element);`
- `boolean addAll(int index, Collection<? extends E> c);`
- `boolean addAll(Collection<? extends E> c).`

Pour illustrer cet algorithme d'ajout, voici par exemple ce que nous obtenons après l'appel de `laListeQ2.add(20)` :



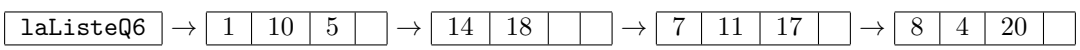
1. `n` sera **forcément** un paramètre du constructeur de l'`ArrayNode`.
2. Pensez à utiliser l'autre constructeur pour ce constructeur sans paramètre.
3. Pensez à déclarer une constante `ARRAYNODE_DEFAULT_SIZE` !!!
4. Si vous avez besoin de retourner un couple de valeur, vous pouvez utiliser `SimpleEntry`, mais ce n'est pas forcément nécessaire.
5. Utilisez la fonction `System.arraycopy`.

Indication : pour vous simplifier la programmation, vous pouvez commencez par faire la méthode d'ajout en queue (`add(e)`).

Question 6 : Pour les méthodes `remove`, on trouve l'élément à supprimer et on l'enlève du tableau. Si cela réduit l'utilisation du tableau à moins que sa moitié, alors on transfère des éléments du nœud suivant pour remplir jusqu'à moitié. S'il après cela, le nœud suivant n'est plus assez rempli(strictement moins de la moitié des éléments), alors on transfère tous les éléments restants dans le nœud courant. Programmez les méthodes de suppression de la liste :

- `E remove(int index);`
- `boolean remove(Object o);`
- `boolean removeAll(Collection<?> c);`
- `boolean retainAll(Collection<?> c).`

Pour illustrer cet algorithme de retrait, voici par exemple ce que nous obtenons après l'appel de `laListeQ5.remove(2)` :



Question 7 : Créez une classe interne qui implémente `Iterator<E>` et ajoutez une méthode `iterator()` qui retourne un itérateur sur la `List`.
Indication : pensez à l'information nécessaire pour repérer une position dans cette structure de liste.

Question 8 : Programmez la méthode `equals` adaptée à la liste : c'est-à-dire que deux listes sont égales si et seulement si elles ont la même taille et leurs éléments correspondants sont égaux deux à deux.

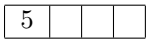
Question 9 : Programmez un algorithme de tri dans votre classe `sort(Comparator<? super E> c)`. Précisez l'algorithme qui vous a inspiré et justifiez ce choix dans le commentaire javadoc.

Question 10 : Programmez les dernières méthodes de l'interface : `lastIndexOf`, `sublist` et `toArray`.

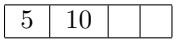
Question 11 : S'il vous reste du temps, ajoutez les fonctions statiques `of`.

Pour illustrer les algorithmes d'ajout et de retrait, voici ce qu'il se passe alors de l'appel de `add` pour les valeurs 5, 10, 1, 18, 14 à partir d'une file vide :

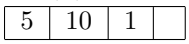
`add(5)`



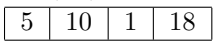
`add(10)`



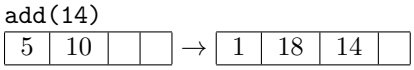
`add(1)`



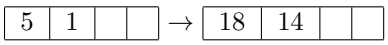
`add(18)`



`add(14)`



Puis le retrait de la valeur 10 : `remove(10)`



Interface `Set`

Question 1 : Dans la méthode `main` d'une classe `ExoSet`, créez deux ensembles *ens1* et *ens2* d'`Integer` en utilisant l'implémentation `HashSet` de l'interface `Set`. Initialiser ces ensembles avec quelques entiers choisis aléatoirement.

Question 2 : Écrivez une méthode `public static void afficher(Set ens)` permettant l'affichage d'un ensemble `ens` en commençant par le nombre d'éléments contenus dans `ens` entre crochets, puis chacun des éléments de cet ensemble entre accolades. Par exemple, l'ensemble $A = \{8, 1, 4, 2\}$ aura comme affichage :

```
[4]{8, 1, 4, 2}
```

Pour les méthodes suivantes, vous veillerez à ne pas modifier les ensembles passés en paramètres.

Question 3 : Créez une méthode `public static Set<Integer> inter(Set<Integer> set1, Set<Integer> set2)` qui renvoie un ensemble contenant l'intersection de deux ensembles en paramètres sans utiliser la méthode `retainAll` de `Set`.

Question 4 : Créez une méthode `public static Set<Integer> union(Set<Integer> set1, Set<Integer> set2)` qui renvoie un ensemble contenant l'union de deux ensembles sans utiliser la méthode `addAll` de `Set`.

Question 5 : Créez une méthode `public static boolean isIn(Set<Integer> set1, Set<Integer> set2)` qui renvoie vrai si l'ensemble `set1` est inclus dans l'ensemble `set2` sans utiliser la méthode `containsAll` de `Set`.

Question 6 : Implémentez les méthodes de test `testInter`, `testUnion` et `testIsIn` dans une classe de test `ExoSetTest` qui vérifient le bon comportement des 3 méthodes décrites plus haut.

Implémentation de l'interface `SortedSet`

Question 1 : Créez une classe `SortedArraySet<E>` qui implémente `SortedSet`. Nous choisissons pour celle-ci comme conteneur, un tableau trié, avec comme attributs additionnels, un entier désignant le nombre d'élément de notre collection, ainsi qu'une référence `comparator`[6] sur un `Comparator`[7]. Cette référence doit être définie comme `final`. Le tableau aura une taille pas défaut de 16.

Question 2 : Ajoutez les constructeurs suivants : `SortedArraySet<E>()`, `SortedArraySet<E>(Comparator<? super E> comparator)`, `SortedArraySet<E>(Collection<? extends E) collection)`

Question 3 : Programmez une méthode `compare(E e1, E e2)` qui compare les deux éléments passés en paramètre. Cette méthode qui respecte la convention usuelle du Java pour les comparaisons, utilise le `comparator` s'il n'est pas `null`, sinon elle utilise l'ordre naturel des éléments. Si aucune de ces deux possibilités n'est disponible, la méthode soulève une exception de type `ClassCastException`.

Question 4 : Programmez les méthodes `isEmpty`, `size` et `clone`. Après avoir programmé `clone`, vous pouvez déclarer votre classe comme `Clonable`.

Question 5 : Programmez une méthode privée[8] `indexOf` qui retourne l'index du premier élément supérieur ou égal à l'élément passé en paramètre (et pas -1 s'il n'est pas présent !). Cette méthode tirera bien évidemment parti du fait que le tableau est trié[9] !

Question 6 : Programmez deux méthodes privées `insert(int index, E element)` et `remove(int index)`. Ces deux méthodes qui modifient le conteneur à partir d'un index, seront très pratiques pour programmer l'interface.

Question 7 : Programmez le reste des méthodes de l'interface `SortedSet<E>`. Pour vos tests, vous pouvez utiliser la classe de tests fournie avec le sujet du TP. Pour l'implémentation de l'`Iterator`, n'oubliez pas la méthode `remove`.

6. `comparator` sera `null` si les éléments sont triés suivant leur ordre naturel
7. Le type exact est `Comparator<? super E>`.
8. Pourquoi faut-il absolument qu'elle soit privée ?
9. Vive le retour de la dichotomie...