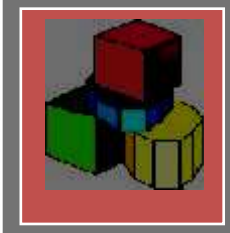
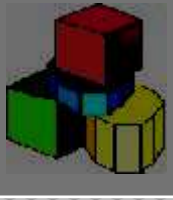


TRI

Les objets opératoires

L'algorithme vu comme un objet



Objectif

L'objectif de ce travail consiste à compléter l'application de tri dont les prémisses vous sont fournies. Il permet de mettre en œuvre le principe de l'héritage et du polymorphisme, l'élaboration d'objet fonctionnel, ainsi que la mise en œuvre de la généricité.

On souhaite mettre en place un système permettant de comparer plusieurs algorithmes de tri. Pour cela l'application doit permettre de construire la collection d'éléments devant être triée. Il est important que la nature de ces éléments ne soit pas préétablie de façon unique, notre système doit pouvoir traiter aussi bien des « String » que tout autre type d'informations. De même pour avoir une certaine souplesse dans le système on ne veut pas que soit associée de manière unique une relation d'ordre aux types d'éléments que nous devons trier. Enfin le nombre d'éléments constituant la collection doit pouvoir être choisi dans un souci d'adapter les conditions d'exécution en fonction des performances de l'environnement. La collection d'éléments à trier sera placée dans un tableau, plus tard nous pourrons envisager de relâcher cette contrainte. En ce qui concerne les principes (algorithmes) de tri, nous souhaitons aussi disposer d'une grande souplesse permettant de choisir dynamiquement l'algorithme à mettre en œuvre. On souhaite pouvoir tester les algorithmes suivants : MaximumSort et HeapSort.

L'interface proposée pour cette application est constituée de listes, celle à gauche est la collection initiale, les éléments sont numérotés selon leur rang de génération. La(es) liste(s) à droite est(sont) le(s) résultat(s) d'un tri, les coûts pour faire ce tri sont notés au-dessus.

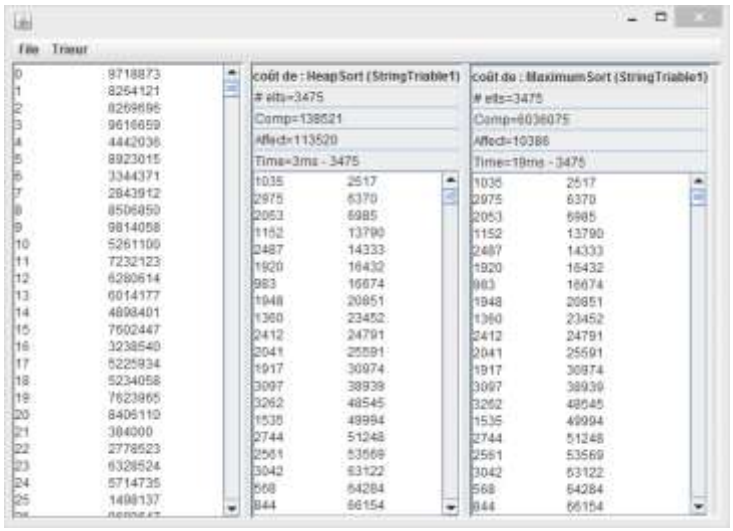
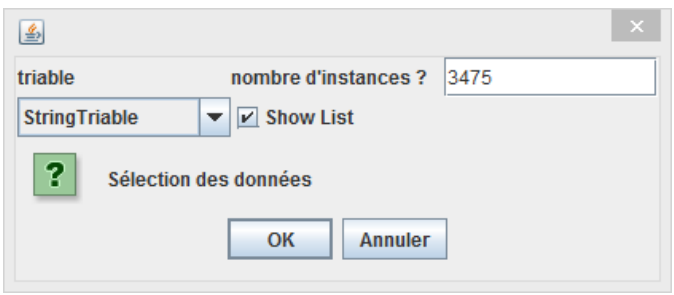
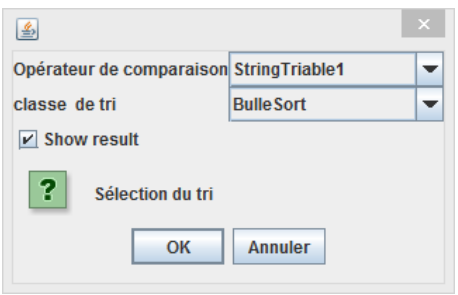


Schéma 1 : Interface utilisateur

Dans le menu Trieur on dispose de la possibilité de fixer la collection à trier et de celle de trier comme le montre les 2 panneaux de saisies suivant :

Un panneau de saisie des informations caractérisant les éléments à trier et permettant d'engendrer la collection et un panneau permettant de choisir la relation d'ordre et l'algorithme de tri à mettre en œuvre :





Informations disponibles

Structure générale de l'application

Afin de répondre aux exigences énoncées dans les objectifs concernant la nature des éléments « triables », nous décidons d'utiliser une classe comme descriptif des caractéristiques que l'on doit connaître des types des éléments à trier. Ceci se réduit aux points suivants :

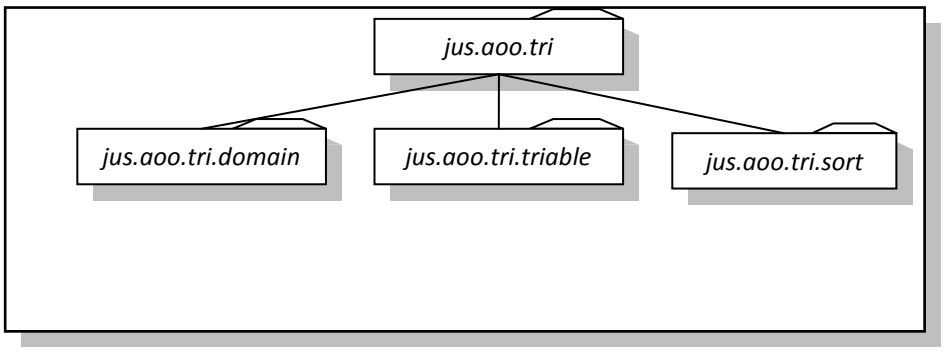
1. Comment construire un élément du type,
2. Comment comparer 2 éléments du type,
3. Comment représenter textuellement un élément du type de façon à permettre un contrôle,
4. Fournir un décompte des appels aux comparaisons.

Pour abstraire ce besoin, nous définissons l'interface Triable qui spécifie les services requis :

Pour abstraire le mécanisme de tri, nous procédons de manière similaire en réalisant de la classe abstraite Tri.

Enfin, on vous fournit un programme principal (classes Trieur) assurant l'interface avec l'utilisateur. Celui-ci peut donc fixer les caractéristiques du tri qu'il veut mettre en place en engendrant la collection d'éléments à trier et en pouvant ensuite effectuer différents tris sur cette collection offrant ainsi la possibilité d'une comparaison précise des algorithmes. Pour structurer l'application, vous mettrez en place les packages suivants : « jus.aoo.tri.triable » contiendra l'ensemble des classes décrivant les types triables, « jus.aoo.tri.sort » contiendra les algorithmes de tri, enfin les classes Tri, Trieur et Triable seront dans le package jus.aoo.tri. Les types additionnels seront dans le package jus.aoo.tri.domaine.

Vous trouverez dans le fichier Jus.Aoo.Tri.jar les classes citées précédemment, vous aurez aussi besoin de la librairie jus.util.jar.



Le travail à réaliser

Objectif n°0

Le code qui vous est fourni ci-dessus ne tire pas partie des possibilités génériques mises à disposition par la version 1.5 de Java. Elaborer une nouvelle version de ces classes où vous utiliserez pleinement la généricité Pour cela n'hésitez pas à consulter la javadoc des classes utilisées.

1. Donnez l'ensemble des modifications que vous avez apportées en justifiant le fait d'avoir introduit de la généricité dans les parties concernées par vos modifications.
2. Complétez, à votre convenance, les assertions de la méthode « sort » dans la classe Tri.

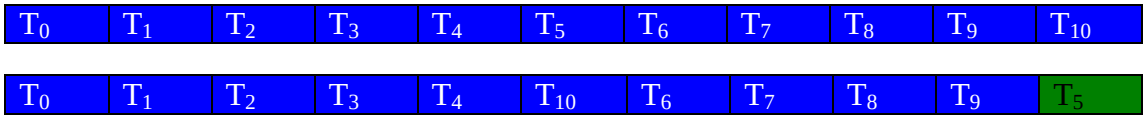
Objectif n°1

3. Réalisez la classe StringTriable de telle sorte que les chaînes engendrées correspondent à la représentation décimale d'**entiers** pris aléatoirement dans l'intervalle [0-10000000]. Pour faire cela vous utiliserez la méthode « random » de la classe « java.lang.Math ». Dans cette version vous fournirez le « comparator » **standard** proposé par la classe « java.lang.String ». Enfin l'afficheur se réduira à rendre la chaîne elle-même et le « count » ne sera pas significatif (la méthode renverra systématiquement 0).
4. Engendrez avec l'application « Trieur » une collection initiale de String. Attention si vous souhaitez engendrer de grandes collections, vous devrez augmenter la taille de la mémoire de la JVM. Pour cela utilisez l'option –Xmx256m pour indiquer que la taille maximum de ma mémoire de la JVM est de 256Mo voire plus si vous êtes très gourmand.
5. Utilisez à la place de la méthode random de la classe Math la classe Random du package java.util.

Objectif n°2

6. Réalisez la classe MaximumSort qui implante l'algorithme de tri d'un tableau par recherche du maximum local. Votre code devra respecter **scrupuleusement** la spécification qui est donnée dans le texte ci-dessous. N'hésitez pas à fonctionnaliser quand cela vous parait utile.

   Principe de ce tri : On partage **virtuellement** le tableau T en 2 zones G et D. Initialement D est vide et G est égal à T. Cet algorithme est basé sur la propriété invariante suivante : D est trié en ordre croissant. Un pas de l'algorithme consiste à faire passer un élément de G dans D. Pour cela on cherche dans G le rang du maximum et on l'échange avec le dernier élément de G. G est diminué de ce dernier élément et D est augmenté de ce même élément. L'algorithme converge puisque la taille de G diminue. L'algorithme est trivial lorsque la taille de G est de 1. Dans l'exemple ci-dessous on admet que le maximum de [T_0, T_{10}] est la valeur T_5.



   Vous utiliserez en priorité les primitives proposées dans la classe Tri pour réaliser la méthode « sort » car on y assure des décomptes.

A partir de la collection engendrée dans l'objectif précédent, trier celle-ci avec cet algorithme (attention : Commencez avec des collections de petite de taille). Vérifier que le tri est correct. Veuillez soigner la qualité du code produit.

Objectif n°3

7. Réalisez la classe StringTriable1 sous-classe de StringTriable qui fixe un autre « Comparator » afin de produire un ordre équivalent à l'ordre des entiers naturels.

   Vous réaliserez cet ordre à partir de la longueur des chaînes et de l'ordre lexicographique fourni par la classe String. Pour cela vous devez compléter la définition proposée ci-dessous. Dans cette version vous devez mettre en place le comptage d'appels au Comparator.

```
package jus.aoo.tri.triable;
import java.util.Comparator;

class StringTriable1$Comparator implements Comparator {
        ...
        public int compare(String s1, String s2) {
                ...
        }
        ...
}
public class StringTriable1 extends StringTriable {
    ...
    /** restitue l'opérateur de comparaison du Triable
     * @return un comparator
     */
    public Comparator comparator(){
          ...
    }
    ...
}
```

   A partir de la collection engendrée dans l'objectif précédent, triez celle-ci avec ce nouvel algorithme. Vérifier que le tri est correct.

Objectif n°4

8. Donnez le coût théorique de cet algorithme (MaximumSort).
9. Réalisez une série de mesures qui montre la validité de votre coût théorique.

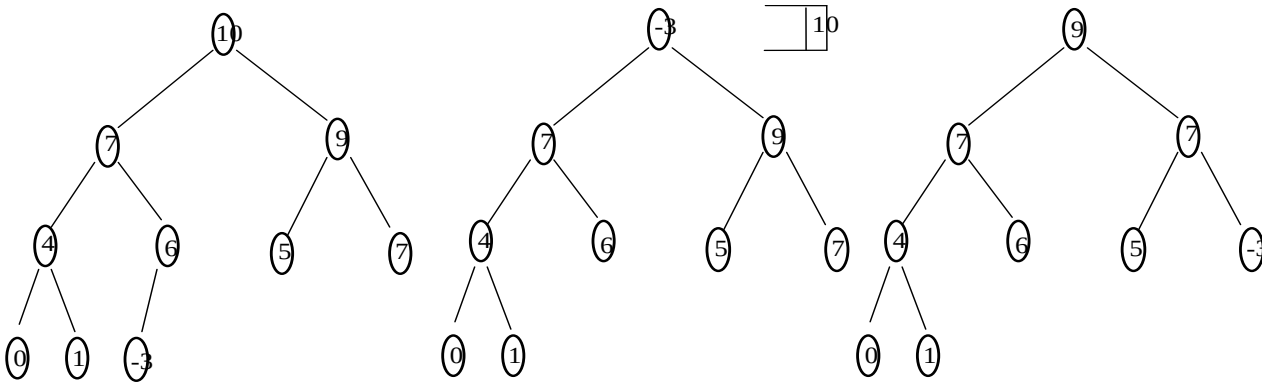
Objectif n°5

On souhaite réaliser un autre algorithme de tri. Le heapsort est un tri qui est plus performant que le précédent. Il est basé sur une structuration **virtuelle** arborescente (binaire) du tableau que l'on nomme heap.

Principe du tri : Si un nœud de l'arbre est à l'index n dans le tableau, alors le fils gauche est à l'index 2n et le fils droit à l'index 2n+1 (**attention l'indexation du tableau doit commencer à 1**). De ce fait le nœud occupant le dernier index du tableau est sur le niveau le plus profond et le plus à droite (-3 dans l'exemple ci-dessous).

Le heap est un ordre partiel tel que les fils d'un nœud sont plus petits que lui. Un pas de l'algorithme (« fixheap ») consiste à échanger la valeur la plus grande (la racine de l'arbre) avec le dernier du tableau (dans l'exemple ci-dessous 10 et -3) puis à replacer -3 à sa position dans le heap privé du dernier (10 qui désormais est correctement placé dans l'ordre).

Pour que cet algorithme fonctionne, il faut construire un heap initial, c'est le rôle de « constheap ».



Voici **une spécification de la solution que vous devez scrupuleusement implanter** :

On note [X,G,D] l'arbre de racine X, de sous-arbre gauche G et de sous-arbre droit D. On note _ une variable anonyme qui n'a pas d'utilité, enfin on note par [] l'arbre vide. L'action « echanger » permet de faire l'échange de 2 éléments dans le heap, Ø représente l'action vide. L'action fixheap permet de faire glisser la valeur à la racine de l'arbre vers la position, qui conserve à l'arbre sa propriété de heap, par une suite d'échanges. Dans l'exemple ci-dessus, il faut 2 échanges soit 6 affectations, on peut, sous certaines conditions, remplacer cette suite d'échanges par un simple décalage dont le coût est moindre.

10. Donnez l'arbre des index du tableau contenant les valeurs des nœuds, quelles propriétés a cet arbre ?

On vous donne ci-dessous une spécification dans une notation pseudo déclarative. Combien de paramètres ont chacune des définitions ci-dessous ? Comment modéliser ceux-ci avec un langage sans mécanisme de filtrage (semi-unification) pour une telle construction ?

fixheap([X,[],[]])	*- Ø*
fixheap([X, [g,_,_],[]]) $\mid X<g$	*- echanger(X,g)*
fixheap([X, [g,_,_],[]]) $\mid X \geq g$	*- Ø*
fixheap([X, [g,_,_], [d,_,_]]) $\mid X \geq g \wedge X \geq d$	*- Ø*
fixheap([X, G, [d,_,_]]) $\mid G=[g,\_,\_] \wedge X<g \wedge g \geq d$	*- echanger(X,g); fixheap(G)*
fixheap([X, [g,_,_], D]) $\mid D=[d,\_,\_] \wedge X<d \wedge d \geq g$	*- echanger(X,d); fixheap(D)*
constheap([X,[],[]])	*- Ø*
constheap([X,G,[]])	*- constheap(G); fixheap([X,G,[]])*
constheap([X,G,D])	*- constheap(G); constheap(D); fixheap([X,G,D])*

11. Réalisez ce programme de tri avec toute l'élégance que requiert la programmation.
12. Pour rendre le programme plus performant, on peut mettre en place plusieurs améliorations :

- *une, très significative, consiste à constater que la méthode fixheap à une récursivité à droite qui peut donc être réduite par une itération (à combiner avec la troisième).*
- *Une autre, pas des moindres, consiste à constater que le replacement de la valeur dans l'arbre suit une branche de celui-ci. Dans la version récursive cette opération requiert approximativement 3logn affectations que l'on devrait pouvoir réduire à logn.*
- *Enfin, on peut limiter le passage de paramètres lors de l'utilisation des méthodes fixheap et constheap.*

13. Construisez une classe abstraire qui généralise à un certain degré les 2 tris MaximumSort et HeapSort que vous venez de réaliser, Il faut construire l'enveloppe[1] maximale des 2 classes. Sur cette base retrouvez le coût théorique de l'algorithme MaximumSort et déduisez celui de HeapSort. Réalisez les 2 sous-classes "SortByMaxSort" et "SortByHeapSort" qui reproduisent "InsertionSort "et "HeapSort".

Objectif n°6

14. Reprenez la classe PointCartesien dans une forme simplifiée (pas besoin de translation & rotation, …). Réalisez, en faisant en sorte d'assurer le plus possible de mise en commun, les classes descriptives caractérisant les ordres suivants :

    1. Selon la projection sur l'axe des abscisses (classe PointTriable1), vous le ferez en reprenant ce qui a été fait dans l'Objectif n°3
    2. Selon la projection sur l'axe des ordonnées (classe PointTriable2), *faites-le en utilisant la notion d'inner-class (sauf si le cours correspondant n'a pas eu lieu).*
    3. Selon la distance à l'origine (classe PointTriable3), *faites-le en utilisant de surcroit la notion de classe anonyme (sauf si le cours correspondant n'a pas eu lieu).*

15. Testez les différents tris sur une même collection

Objectif n°7

Programmez le tri RapidSort. Ce tri est basé sur la décomposition en 2 parties G et D du tableau tel que tous les éléments de G sont plus petits que les éléments de D. Le tri de chacune des 2 parties conduit à un ordre total. Cette première étape de répartition construit un ordre partiel. Pour constituer ces 2 parties, on choisit une valeur X déterminant celles devant se situer dans G et celles devant se situer dans D. On nommera cette opération « divide ».

16. A quelle condition ce tri est-il performant ? Quelle propriété doit avoir la valeur X ?
17. Proposez une manière de choisir cette valeur.
18. Réalisez la méthode « divide » assurant : $\forall e_1 \in G\ e_1 < T_5\ \&\ \forall e_2 \in D\ e_2 \geq T_5$

T_0	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}

où T_5 a la valeur X.

- Pour ce faire on place la valeur X en position 0 et on note p l'index du dernier élément de G tel que G = T[1 .. p],
- on le fixe au départ à 0 ce qui revient à considérer qu'initialement G est vide.
- On note i l'index tel que T]p .. i[correspond à D, on le fixe au départ à 1 ce qui revient à considérer qu'initialement D est vide.
- Un pas consiste à traiter T[i], si celui-ci doit appartenir à G, il suffit de l'échanger avec le premier de D.
- Quand on a terminé la répartition, il faut échanger T[0] avec le dernier de G.

19. Quel est le coût théorique de cette opération ? En déduire le coût théorique du tri.
20. Terminez en réalisant le programme assurant le tri du tableau.

Objectif n°8

[1] *union des méthodes et attributs nécessaires aux 2 méthodes*

21. Etendre la possibilité de tri à d'autres structures de collection que les tableaux. Proposez une solution simple à mettre en œuvre. Pour cela observer les classes décrivant des collections.

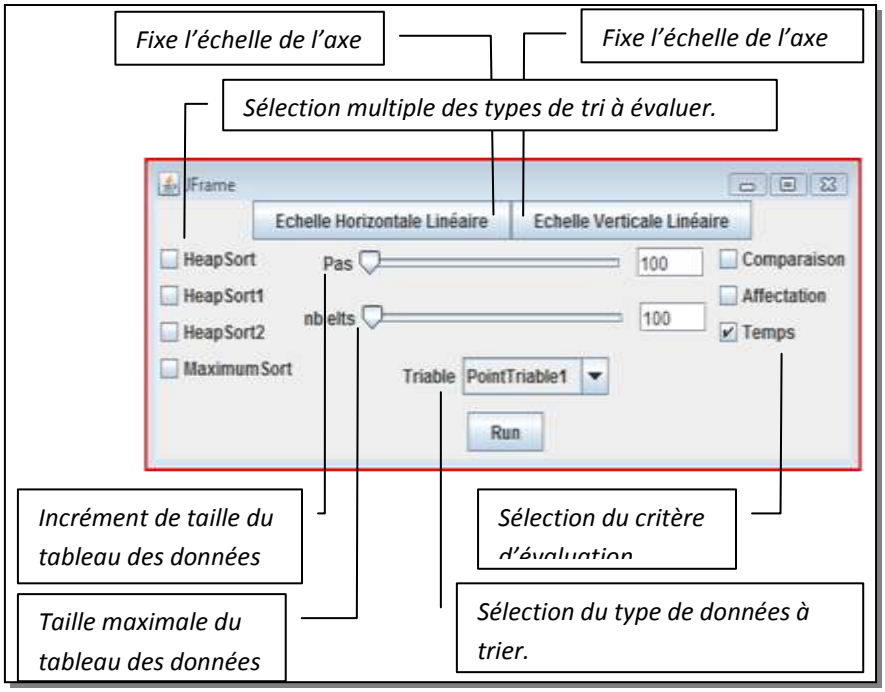
Objectif n°9

Afin d'avoir une analyse complète, on se propose de considérer les cas particuliers de collections pour lesquelles les coûts réels peuvent s'éloigner des coûts moyens.

22. Programmez le tri InsertionSort. On partage **virtuellement** le tableau T en 2 zones G et D. Initialement G est composé du premier élément de T et D des autres. Cet algorithme est basé sur la propriété invariante suivante : G est trié en ordre croissant. Un pas de l'algorithme consiste à faire passer le premier élément de D dans G. Pour cela tant que cet élément est plus petit que son prédécesseur, on l'échange avec celui-ci. A la fin de cette opération, G est augmenté d'une unité et D est diminué en conséquence. L'algorithme converge puisque la taille de D diminue. Pour optimiser le coût de modification utilisez la méthode "circularSwap".
23. Proposez une liste des configurations de collections qui vous semblent être intéressantes à observer.
24. Elaborez des « Triable » qui produisent ces configurations particulières.
25. Testez ces différentes configurations avec les tris que vous avez réalisés.

Objectif n°10

A l'aide de la librairie JFreeChart, établissez des courbes de coûts en fonction du nombre d'éléments à trier. On pourra établir le coût sur la base du temps passé à exécuter l'opération, du nombre de comparaisons effectuées ou bien encore sur la base du nombre d'affections réalisées.



On peut envisager de comparer plusieurs méthodes de tris avec une même donnée initiale, les courbes affichées montrant explicitement les différences de performance. Faites le test avec les différentes techniques de tri élaborées précédemment. Attention de tester avec des quantités adaptées aux performances de l'environnement hôte.

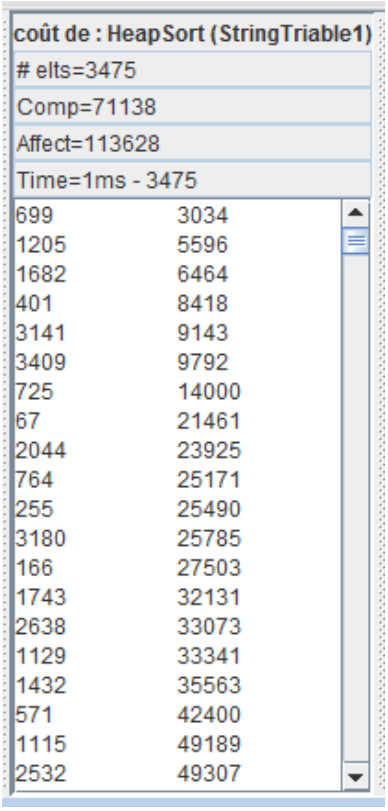
L'interface utilisateur proposée est présentée ci-dessus.

26. Complétez la méthode « run » de la classe Evaluation pour réaliser une campagne de test qui fournisse des graphiques comme ceux proposés ci-après. Le titre indique le Triable utilisé par cette campagne de tests. L'abscisse indique le nombre d'éléments de la collection. En ordonnée on

trouve la quantité observée c.-à-d. soit le temps effectif passé, soit le nombre de comparaisons effectuées ou enfin le nombre d'affectations. Pour avoir une indication plus fiable du temps passé durant l'exécution d'un tri, vous pouvez utiliser en lieu et place de `System.currentTimeMillis()` la méthode `System.nanoTime()` ou un Counter. La première n'a d'utilité que lorsque la durée est relativement longue, la seconde est une bonne unité de base. La seconde méthode est plus appropriée pour des durées courtes.

Objectif n°11

27. Dans la présentation du résultat d'un tri, comment interprétez-vous la première colonne ?
28. Quel est le coût du calcul de cette information à partir des états initial et final du tableau ?
29. A partir de l'état initial du tableau et de la première colonne, quel est le coût du calcul de la seconde colonne ?
30. Proposez une nouvelle spécification "sorting" dans la classe Tri qui restitue le tableau représentant la première colonne. On peut, à ce niveau d'abstraction, écrire une réalisation générale dont le coût est en relation avec la question 28.
31. Proposez une redéfinition de cette méthode dans la classe InsertionSort en restant au plus proche de la méthode sort que vous y avez réalisée.
32. Réaliser les méthodes "triage", "sorting" et "verifierTriage" à l'instar des méthodes "trier", "sort", "verifier" dans la classe Tri.



La spécification de la classe $Evaluation

Constructor Summary

protected	**$Evaluation** ()

Method Summary

protected void	**createChart** (java.lang.String titre, java.lang.String titreV, java.lang.String titreH, org.jfree.data.xy.XYSeriesCollection data) Affiche le chart
protected abstract void	**run** () Exécute les tris suivant la configuration donnée.
protected $Evaluation.Critere	**selectedCriterion** () retourne le critère sélectionné
protected int	**selectedMaxSize** () retorune la taille maximum du tableau
protected int	**selectedPace** () retourne la valeur du pas
protected <T> Triable<? super T>	**selectedSortable** () retourne une instance du triable sélectionné
protected java.util.List<Tri>	**selectedSorts** () retourne la liste des instances de tris sélectionnés.

Constructor Detail

protected **$Evaluation**() throws java.awt.HeadlessException

Throws: java.awt.HeadlessException

Method Detail

protected void **createChart**(java.lang.String titre,
java.lang.String titreV,
java.lang.String titreH,
org.jfree.data.xy.XYSeriesCollection data)

Affiche le chart

---

protected <T> Triable<? super T> **selectedSortable**() throws java.lang.Exception

retourne une instance du triable sélectionné

Returns: un triable

Throws: java.lang.Exception

---

protected int **selectedPace**()

retourne la valeur du pas

Returns: la valeur du pas

---

protected int **selectedMaxSize**()

retorune la taille maximum du tableau

Returns: la taille maximum du tableau

---

protected java.util.List<Tri> **selectedSorts**() throws java.lang.Exception

retourne la liste des instances de tris sélectionnés.

Returns: la liste des instances de tris sélectionnés.

Throws: java.lang.Exception

---

protected $Evaluation.Critere **selectedCriterion**()

retourne le critère sélectionné

Returns: le critère sélectionné

La spécification de la classe $Evaluation.Critere

Enum Constant Summary

AFFECTATION
COMPARAISON
TEMPS

La spécification de la classe $Evaluation.Echelle

Enum Constant Summary

LINEAIRE
LOGARITHMIQUE