TdA7

Approche de la Programmation et de la Conception Orientée Objet

1. Objectif

L'objectif de ce Td est de 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.

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 triés. 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'information. 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. 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, nombre d'appels au comparateur pour faire ce tri est noté au-dessus. 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 :

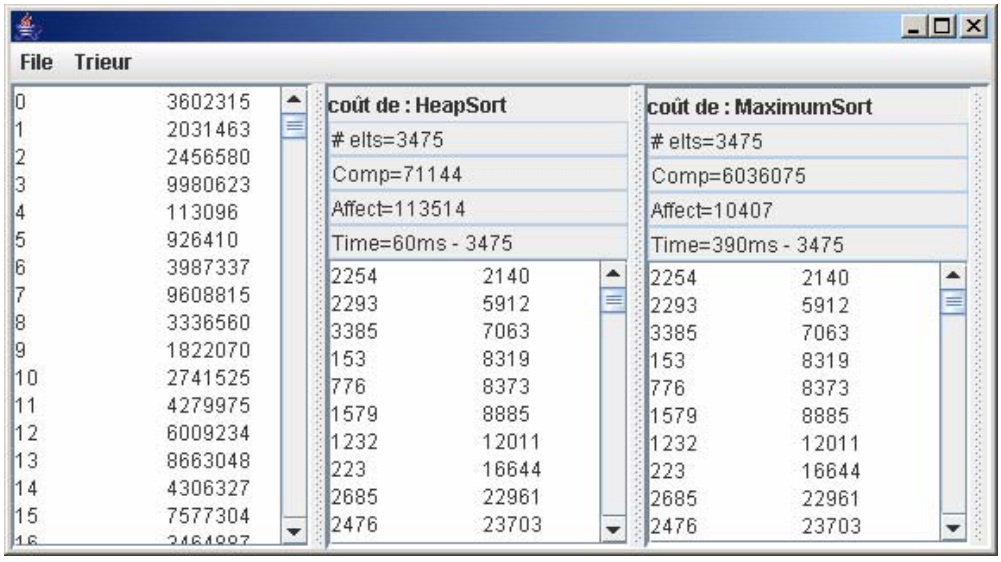
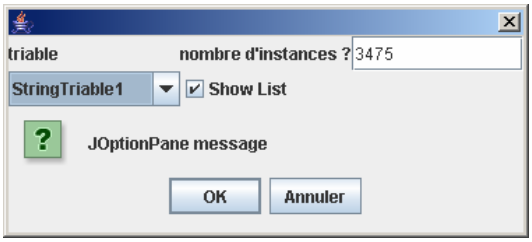


Schéma 1 : Interface utilisateur

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



2. Informations disponibles

2.1. 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 au comparator.

Pour abstraire ce besoin, nous définissions l'interface suivante qui spécifie les services requis :

```
package jus.apcoo.tda7;
import java.util.Comparator ;

public interface Triable {
    /** générateur aléatoire d'un élément du type
     * @return une valeur d'un type
     */
    public Object newInstance();
    /** restitue l'opérateur de comparaison du Triable
     * @return un comparator
     */
    public Comparator comparator();
    /** restitue la représentation textuelle de la
     * valeur servant dans la relation d'ordre.
     * @return une chaine
     */
    public String toString(Object o);
      /** restitue le nombre d'appels au comparator
       */
      public long count();
      /** réinitialise le nombre d'appels au comparator
       */
      public void resetCount();
}
```

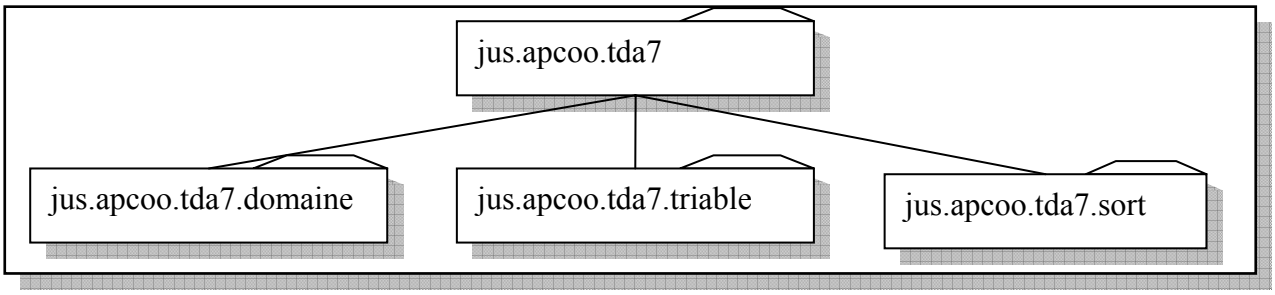
Pour abstraire le mécanisme de tri, nous procédons de manière similaire en réalisant une classe abstraite suivante :

```
package jus.apcoo.tda7;
import java.util.Comparator;

public abstract class Tri {
    /**le nom du tri*/
    private String name;
    /** Le nombre d'affectations (significatives) engendrées par le tri.*/
    private long count;
    /** le temps CPU mis par le tri.
     * Attention celui-ci peut être profondément altéré par des conditions externes.
     */
    private long time;
    /**  restitue le nom du tri.
     * @return le nom du tri
     */
    public String name(){return name;}
    /** restitue le comptage du nombre d'affectation
     * @return le nombre d'affectation
     */
    public long count(){return count;}
    /** restitue le temps passé pour le tri.
     * @return la durée du tri.
     */
    public long time(){return time;}
    /** renvoie un tableau trié.
     * @param t le tableau à trier
     * @return le tableau triée
     */
    public Object[] trier(Object[] t,Comparator c){
      count=0;
      Object[] tab = t.clone();
      time=System.currentTimeMillis();
      Object[] res = sort(tab,c);
      time=System.currentTimeMillis()-time;
      return res;
    }
    /** version assurant le tri selon un algorithme particulier **/
    protected abstract Object[] sort(Object[] t,Comparator c) ;
    /*-------------------------------------------------------------------------
        la suite de méthodes sont utiles pour faire la mise en œuvre de l'algorithme
        et assurer le décompte des affectations
       -------------------------------------------------------------------------*/
    /** Effectue un échange de 2 valeurs dans le tableau
     * @param t le tableau de valeurs
     * @param a l'index de la première valeur
     * @param b l'index de la seconde valeur
     */
    protected void swap(Object[] t, int a, int b) {
      Object[] x = new Object[1] ;
      copy(t,a,x,0,1); copy(t,b,t,a,1); copy(x,0,t,b,1);
    }
    /** Réalise un décalage circulaire d'une suite contiguë de valeurs d'un tableau.
     * @param t le tableau de valeurs
     * @param start l'index minimum du sous-tableau
     * @param end l'index maximum du soius-tableau
     */
    protected void circularSwap(Object[] t, int start, int end){
      Object[] x = new Object[1] ;
      copy(t,end,x,0,1);
      if(start<end)copy(t,start,t,start+1, end-start);
      else{copy(t,0,t,1,end-1); copy(t,t.length-1,t,0,1); copy(x,0,t,start,1);}
    }
    /** Effectue une copie de sous-tableau.
     * @param src le tableau source
     * @param srcPos la position dans la source
     * @param dest le tableau destination
     * @param destPos la position dans la destination
     * @param length le nombre d'éléments à copier
     */
    protected void copy(Object[] src,int srcPos,Object[] dest,int destPos,int length){
      System.arraycopy(src,srcPos,dest,destPos,length);
      count+=length;
    }
    /** Vérifie que le tableau t est trié en ordre croissant.
     * @param t le tableau à vérifier
     * @param c le comparator de la relation d'ordre
     * @return t.length si le tableau est ordonné ou l'indice de violation de la relation
d'ordre.
     */
    protected static int verifier(Object[] t,Comparator c){
      for(int i=1 ;i<t.length ;i++) if(c.compare(t[i],t[i-1])<0) return i ;
      return t.length;
    }
}
```

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 des tris différents 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.apcoo.td7a.triable contiendra l'ensemble des classes décrivant les types triables, jus.apcoo.tda7.sort contiendra les algorithmes de tri, enfin les classes Tri, Trieur et Triable seront dans le package jus.apcoo.tda7. Les types additionnels seront dans le package jus.apcoo.tda7.domaine.

Vous trouverez dans le fichier Jus.Apcoo.tda7.jar les classes citées précédemment.



3. Le travail à réaliser

3.1. Objectif n°0

Le code qui vous est fourni ci-dessus ne tire pas partie des possibilités génériques fournies par la version 1.5 de Java. Elaborer une nouvelle version de ces classes où vous utiliserez pleinement la généricité.

3.2. Objectif n°1

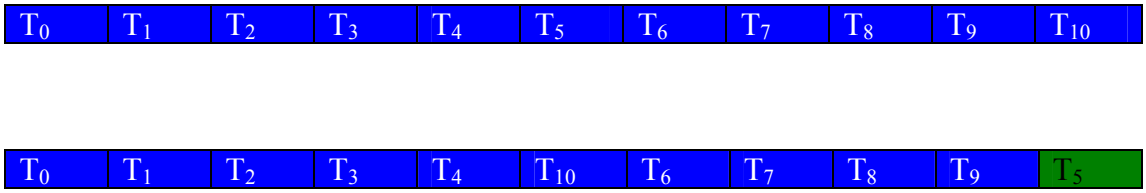
Réaliser la classe StringTriable de telle sorte que les chaînes engendrées correspondent à la représentation décimale d'entiers pris aléatoirement dans l'intervalle [0-10000000]. Pour faire cela vous utiliserez la méthode « random » de la classe « java.lang.Math ». Dans cette version vous fournirez le « 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).

Engendrer 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 –xm256m pour indiquer que la taille maximum de ma mémoire de la JVM est de 256Mo.

3.3. Objectif n°2

Réaliser la classe MaximumSort qui implante l'algorithme de tri d'un tableau par recherche du maximum local. On partage 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 n'utiliserez que les primitives proposées dans la classe Tri pour réaliser la méthode « sort ». A partir de la collection engendrée dans l'objectif précédent, trier celle-ci avec cet algorithme (attention commencez avec des collections de petites de taille). Vérifier que le tri est correct.

3.4. Objectif n°3

Réaliser 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.apcoo.tda7.triable;
import java.util.Comparator;

class StringTriable1Comparator 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, trier celle-ci avec ce nouvel algorithme. Vérifier que le tri est correct.

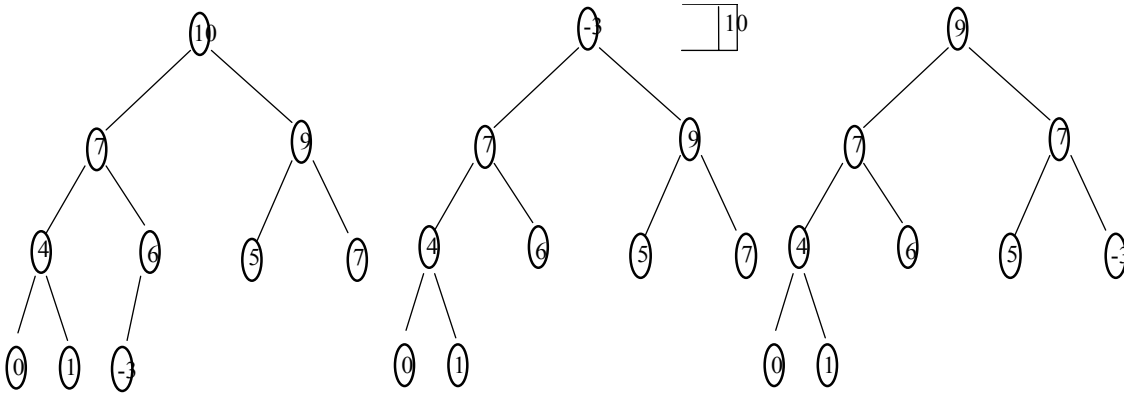
3.5. Objectif n°4

Donner le coût théorique de cet algorithme. Réaliser une série de mesures qui montre la validité de votre coût théorique.

3.6. Objectif n°5

On souhaite réaliser un autre algorithme de tri. Le heapsort est tri qui est plus performant que le précédent. Il est basé sur une structuration arborescente (binaire) du tableau que l'on nomme heap. 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 d'un tableau commençant à 0, il faut adapter cette règle). De ce fait le nœud occupant le dernier index du tableau est sur le niveau le plus profond et le nœud 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. 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 la spécification de la solution que vous avez à 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.

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

3.7. Objectif n°6

Définissez la classe Point dans le package jus.apcoo.tda7.domaine sur la base de PointCartesien ou PointPolaire à votre convenance. 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),
2. Selon la projection sur l'axe des ordonnées (classe PointTriable2),
3. Selon la distance à l'origine (classe PointTriable3).

Testez les différents tris sur une même collection.