	<b>Algorithmique et programmation séquentielle</b>	
	<b>ITI1</b>	<b><i>Listes dynamiques simplement chaînées</i></b>
	hepia, HES-SO//Genève	Laboratoire Ada

## Préambule

Ce travail pratique est organisé en trois parties traitant de la notion de listes dynamiques simplement chaînées. Les parties sont les suivantes :

1. Implémentation d'un paquetage générique de pile dynamique et application à l'algorithme du tri à deux piles.
2. Implémentation d'une liste dynamique circulaire simplement chaînée et application au problème de Joseph (permutation d'une liste de nombre).
3. Implémentation d'un paquetage non-générique de liste dynamique triée simplement chaînée et application au comptage de mots dans un texte.

## Recommandation

**Vous êtes fortement encouragé à aller au libre service pour obtenir des conseils et de l'aide pour avancer votre TP.**

## Rendu

Le listing du code doit être rendu sur papier. **Veuillez imprimer deux pages par feuille.** Le code doit être bien **indenté, commenté et modularisé** avec des fonctions et procédures.

Les codes sources doivent être rassemblés dans une **archive .zip à votre nom** qui sera déposée sur le site du cours dans <https://cyberlearn.hes-so.ch>

Plus précisément, l'étudiant **Laurent Sampaio** mettra **uniquement** 7 fichiers nommés :


```
gestion_pile.ads
gestion_pile.adb
tri2piles.adb
joseph.adb
gestion_liste.ads
gestion_liste.adb
dico.adb
```

dans un **répertoire** nommé **laurent\_sampaio** qui sera lui même zippé en un **fichier** nommé **laurent\_sampaio.zip**. C'est ce fichier zippé qui devra être déposé dans <https://cyberlearn.hes-so.ch>

**Attention ni espaces, ni accents, ni majuscules dans les noms !**

**Sous peine de sanction, vous devez respecter toutes ces spécifications.**

**Ce travail pratique est noté.** L'évaluation sera faite sur la base du listing et d'une interrogation orale lors de laquelle vous expliquerez le travail réalisé.

	<b>Algorithmique et programmation séquentielle</b>	
	<b>ITI1</b>	<b><i>Piles dynamiques et application</i></b>
	hepia, HES-SO//Genève	Laboratoire Ada

## Objectifs

1. Création d'un paquetage générique en Ada.
2. Implémentation d'une pile dynamique.
3. Instanciation et utilisation de cette pile.
4. Tests systématiques
5. Indentation, commentaires et modularisation du code.

## Enoncé

Ecrire un paquetage Ada générique permettant la gestion de piles dynamiques. Ce paquetage sera utilisé pour effectuer le tri d'un tableau à l'aide de deux piles. Le tri fonctionne de la manière suivante.

- Le tableau est parcouru élément par élément.
- Une première pile (celle de gauche dans l'exemple de la figure 1 contient toujours des éléments triés en ordre croissant en partant du sommet.
- Une seconde pile (celle de droite dans l'exemple de la figure 1 contient toujours des éléments en ordre décroissant en partant du sommet.
- A chaque étape, on dépile les éléments d'une des piles pour les empiler dans l'autre, jusqu'à ce que l'élément traité soit empilable dans la pile de gauche en respectant l'ordre dans chacune des deux piles.
- En fin de tri, tous les éléments se trouvent dans la première pile; on termine donc le tri en vidant cette pile dans le tableau de départ.

## Cahier des charges

- Dans un fichier `gestion_pile.ads`, écrire les spécifications d'un paquetage générique permettant la gestion d'une pile dynamique.
- Dans un fichier `gestion_pile.adb`, écrire le corps de ce paquetage.
- Ecrire un programme `tri2piles.adb` qui utilise le paquetage de gestion de pile pour implémenter le tri décrit ci-dessus. Créer une procédure pour le tri.
- Le lancement du programme avec une liste quelconque de nombres entiers sur la ligne de commande doit afficher la liste triée avec un nombre par ligne.  
**Rien d'autre ne devra être affiché.**

Par exemple :

```
~> ./tri2piles 18 2 34 21 7 4
2
4
7
18
21
34
```

## Règles du jeu

L'exemple décrit ci-dessous est illustré à la figure 1. Soit le tableau à trier suivant :

17	34	20	40	25
----	----	----	----	----

Le tableau est parcouru de gauche à droite.

- 17 est placé dans la pile de gauche.
- 34 est plus grand que 17: 17 est dépilé de la pile de gauche pour être empilé dans la pile de droite; 34 est placé dans la pile de gauche.
- 20 est plus petit que 34 et plus grand que 17: il est empilé dans la pile de gauche.
- 40 est comparé avec 20 et 34: ceux-ci sont dépilés de la pile de gauche pour être empilés dans la pile de droite; 40 est empilé à gauche.
- 25 est plus petit que 34: celui-ci est dépilé de droite pour être empilé à gauche; 25 est empilé à gauche.
- Il n'y a plus d'élément à traiter: la pile de droite est vidée, chacun de ces éléments étant empilé dans la pile de gauche; cette pile contient au final tous les éléments en ordre croissant en partant du sommet.
- Il ne reste plus qu'à vider la première pile dans le tableau.

17	

Traitement de 17

34	17

Traitement de 34

20	
34	17

Traitement de 20

	34
	20
40	17

Traitement de 40


25	
34	20
40	17

Traitement de 25

17	
20	
25	
34	
40	

Etat final

Figure 1 : Utilisation des deux piles pour le tri

	Algorithmique et programmation séquentielle	
	ITI1	<i>Listes circulaires et application</i>
	hepia, HES-SO//Genève	Laboratoire Ada

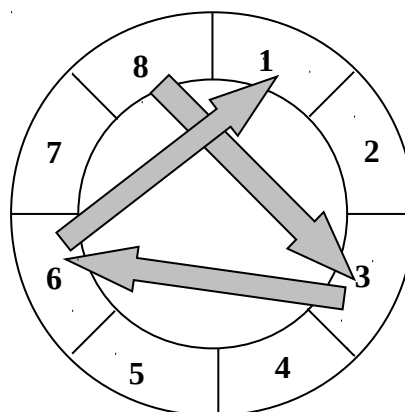
## Objectifs

1. Manipulation de pointeurs
2. Mise en œuvre d'une liste dynamique circulaire simplement chaînée
3. Application au problème de Joseph
4. Indentation, commentaires et modularisation du code.

## Enoncé

Le problème de Joseph sert à trouver une permutation de nombres en utilisant une liste circulaire (un anneau). Dans cette liste circulaire, tous les nombres entiers compris entre 1 et  $n$  sont initialement introduits dans le sens horaire. Puis, dans une deuxième phase, les  $n$  nombres seront éliminés de cette liste comme suit : en commençant à partir de l'élément contenant  $n$ , on supprime successivement tous les  $k^{\text{ème}}$  éléments de la liste, en effectuant un parcours circulaire dans la liste ; on répète ceci jusqu'à ce que tous les éléments ont été supprimés. Dès qu'un élément est supprimé, son prédécesseur devient le nouveau point d'accès à la liste.

**Exemple :** si  $n=8$  et  $k=3$ , la suppression des nœuds contenant les huit entiers se fait dans cet ordre : 3, 6, 1, 5, 2, 8, 4, 7.



## Cahier des charges

Pour la définition d'une liste dynamique simplement chaînée, on peut utiliser l'article :

```
type T_Element;  
type T_Liste is access T_Element;  
type T_Element is record  
    Info : Integer;  
    Suivant : T_Liste;  
end record;
```

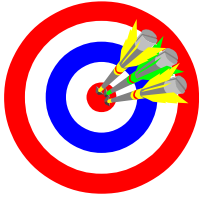
Votre programme `joseph.adb` doit être bien modularisé. Il comprendra notamment :

- la structure de données correspondant à la liste circulaire ; celle-ci ne doit être accessible que par un seul pointeur ;
- un sous-programme pour insérer un nombre dans une liste circulaire ; celui-ci sera appelée  $n$  fois pour initialiser le problème de Joseph ;
- un sous-programme pour supprimer le  $k^{\text{ème}}$  élément dans la liste circulaire à partir du pointeur d'accès à la liste ; celui-ci est déplacé avant l'élément supprimé.

La liste des éléments extraits doivent être affichés au fur et à mesure de leur suppression dans la liste circulaire. Le lancement du programme avec la valeur de  $n$  suivie de  $k$  sur la ligne de commande doit afficher la liste des éléments extraits avec un nombre par ligne. **Rien d'autre ne devra être affiché.**

Par exemple :

```
~> ./joseph 8 3  
3  
6  
1  
5  
2  
8  
4  
7
```

	<b>Algorithmique et programmation séquentielle</b>	
	<b>ITI1</b>	<b><i>Listes triées et application</i></b>
	hepia, HES-SO//Genève	Laboratoire Ada

## Objectifs

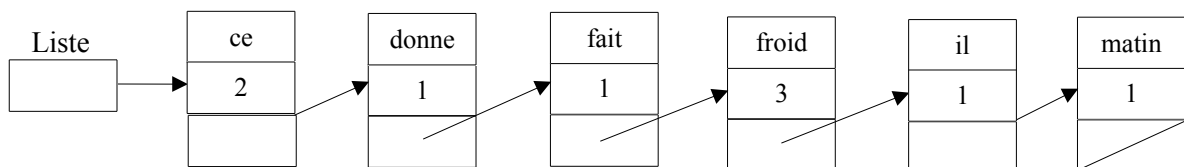
1. Manipulation de pointeurs
2. Mise en œuvre d'une liste dynamique triée simplement chaînée à l'aide d'un paquetage non-générique
3. Application au comptage de mots dans un texte
4. Tests systématiques
5. Indentation, commentaires et modularisation du code

## Enoncé

Il s'agit de créer un paquetage non générique permettant de traiter les mots d'un texte afin de pouvoir les comparer. Les mots du texte seront placés dans une liste triée en ordre lexicographique (ordre du dictionnaire). Chaque fois qu'un mot apparaît plusieurs fois, un compteur sera incrémenté. Le paquetage contiendra des sous-programmes permettant de créer une liste, d'ajouter un mot ou de retrancher un mot. De plus, on peut imaginer les opérations : donner les mots communs à deux listes, donner les mots d'une liste non contenus dans une autre liste, concaténer deux listes, ...

**Exemple :** « il fait froid ce matin, ce froid donne froid »

Ceci donne la liste :

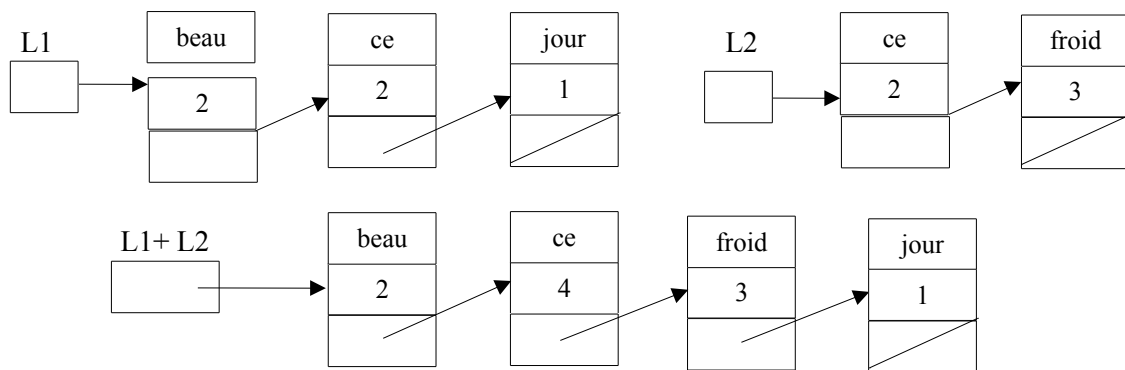


## Cahier des charges

- Ecrire les spécifications du paquetage non-générique `gestion_liste.ads` : toute la structure ainsi que les sous-programmes y seront définis.
- Ecrire le corps du paquetage correspondant `gestion_liste.adb`, lequel contiendra :
  - Une procédure `insert` permettant d'insérer un mot dans une liste triée lexicographiquement, et qui incrémente le compteur si le mot est déjà présent.
  - Une fonction `search` permettant de rechercher un mot dans une liste triée et qui retourne le nombre d'occurrences de ce mot.
  - Une procédure `delete` permettant de supprimer un mot dans une liste triée. Une exception sera levée si le mot n'est pas dans la liste.

- Une fonction `intersect` qui retourne une liste triée avec tous les mots communs à deux listes triées (mettre le nombre d'occurrences de ces mots communs à 1).
- Une fonction `difference` retournant une liste triée avec tous les mots d'une première liste triée n'apparaissant pas dans une deuxième liste triée.
- Un opérateur "+" effectuant la concaténation de deux listes triées. Le nombre d'occurrences des mots communs sont sommés.

Par exemple :



- Ecrire un programme `dico.adb` permettant de tester ce paquetage. Utiliser des fichiers texte sans accents ni ponctuation, tout en minuscule. Le lancement du programme sur la ligne de commande :

```
~> ./dico -p book.txt
```

affichera la liste triée des mots du fichier texte `book.txt` avec par ligne un mot et son nombre d'occurrences :

```
beau 2
ce 4
froid 3
jour 1
```

La ligne de commande :

```
~> ./dico -s book.txt beau
```

recherchera le mot `beau` dans la liste triée créée à partir du fichier `book.txt` et affichera le nombre d'occurrences de `beau`, soit 2 dans l'exemple.

La ligne de commande :

```
~> ./dico -x book.txt beau
```

supprimera le mot `beau` dans la liste triée créée à partir du fichier `book.txt` et affichera la liste triée résultante avec par ligne un mot et son nombre d'occurrences. Le fichier original n'a pas besoin d'être modifié.

Les lignes de commande :

```
~> ./dico -i book1.txt book2.txt
```

```
~> ./dico -d book1.txt book2.txt
```

```
~> ./dico -c book1.txt book2.txt
```

afficheront chacune la liste triée résultant de l'intersection (`-i`), de la différence (`-d`) et la concaténation (`-c`) des listes triées créées à partir des fichiers `book1.txt` et `book2.txt` toujours selon le format avec un mot et son nombre d'occurrences par ligne. Les fichiers originaux n'ont pas besoin d'être modifiés.

**Rien d'autre ne devra être affiché.**