TP 8 – Structures, tableaux dynamiques

1. Nombres complexes

On désire écrire une bibliothèque de fonctions permettant de manipuler les nombres complexes. Les nombres complexes seront stockés dans une structure de données composée de 2 nombres flottants représentant la partie réelle et la partie imaginaire. Ce nouveau type de donnée s'appellera Complexe. S'aider du fichier exStruct.c disponible dans l'archive.
Écrire les fonctions :

- `void affiCplx(Complexe z)` qui affiche un nombre complexe
- `Complexe addCplx(Complexe z1, Complexe z2)` qui retourne la somme de 2 nombres complexes
- `Complexe multCplx(Complexe z1, Complexe z2)` qui retourne le produit 2 nombres complexes

Écrire le main permettant de valider ces fonctions.

2. Taille du tas, taille de la pile

Reprendre le programme du crible d'Ératosthène et déclarer un tableau statique de 8 millions de cases de type entier de cette manière :

```
char tab [8000000] ;
```

Votre programme compile-t-il ? Votre programme fonctionne-t-il ?
Essayer maintenant de déclarer un tableau de 9 millions de case. Votre programme fonctionne-t-il ?
Réécrire ce programme en faisant une allocation dynamique du tableau. Essayez d'allouer 10 millions de cases, puis essayer d'allouer 200 millions de cases. Votre programme compile-t-il ? Votre programme fonctionne-t-il ?
Déterminez alors, le nombre premier le plus grand que l'on puisse trouver avec ce programme.

Explication : les tableaux statiques sont déclarés dans la pile qui, par défaut, a une taille de 8 Mo sous Linux. Par contre, les tableaux dynamiques sont déclarés dans le tas qui a une taille dépendant de la mémoire qui est disponible sur la machine (RAM + mémoire virtuelle de la partition swap).

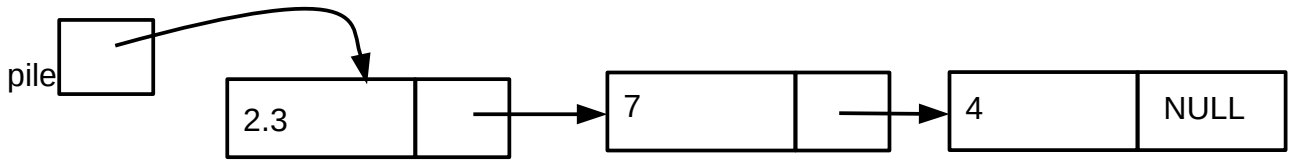
3. Piles et listes

Une file est un container de type FIFO : First IN First Out. On peut penser à une File d'attente dans un magasin ou le premier client entré est le premier servi.
Une pile est un container de type LIFO : Last IN First Out. On peut penser à une Pile d'assiettes : c'est la dernière assiette rangée qui sera la première à être réutilisée.

Une liste chaînée est un ensemble de cellules. Chaque cellule contient un pointeur vers une autre cellule.
La liste est repérée par un "pointeur de tête" qui pointe sur la première cellule et éventuellement par un pointeur dit "pointeur de queue" qui pointe sur la dernière cellule.

Télécharger le fichier GestionPile.zip sur moodle.

Cet ensemble de fichiers insère des nombres dans une liste. Le pointeur pointe sur la première cellule. L'ajout se fait en tête de liste. S'il faut enlever un élément, ou en ajouter un, c'est en tête de liste que cela se passe. On est donc en présence de l'implémentation d'une Pile par une Liste avec une seul pointeur sur la tête.
Voici la forme de la liste obtenue par le fichier DemoPile.c



Compiler le programme à l'aide du Makefile en tapant dans la console où se situe le Makefile :

```
Terminal Fichier Édition Affichage Rechercher Terminal Aide
demo> ls -l
total 16
-rw-rw-r-- 1 lefebvre lefebvre 1458 déc.   8 23:00 DemoPile.c
-rw-rw-r-- 1 lefebvre lefebvre  278 déc.   2 18:01 Makefile
-rw-rw-r-- 1 lefebvre lefebvre 1626 déc.   2 18:27 Pile.c
-rw-rw-r-- 1 lefebvre lefebvre 1114 déc.   2 15:50 Pile.h
demo> make
gcc -o DemoPile.o -c DemoPile.c -Wall
gcc -o Pile.o -c Pile.c -Wall
gcc -o DemoPile DemoPile.o Pile.o -lm
demo> ./DemoPile
Pile : Pile vide
Pile : 2.300000 7.000000 4.000000
demo>
```

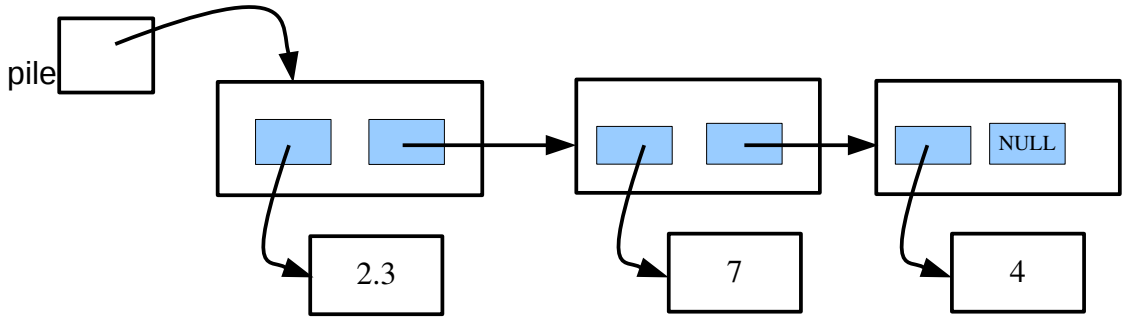
Compléter le main du fichier DemoPile.c afin de coder :

- la fonction *depiler :* supprime le premier élément de la pile et renvoie la nouvelle pile. La valeur du nombre stocké dans la cellule est retournée dans un paramètre passé par adresse (*cf. exemple d'utilisation dans les commentaires du main).*
- la fonction *insererFinPile :* insére une nouvelle cellule à la fin de la liste. (*cf. exemple d'utilisation dans les commentaires du main).*

4. Pile générique

Dans l'exercice précédent la pile ne peut contenir que des nombres. Il serait intéressant de dissocier la gestion de la pile (création, insertion, recherche…) du contenu.

Une solution à ce problème est la suivante. La cellule composant la liste contient un pointeur générique vers un élément :



Vous trouverez dans l'archive GestionPileVoid.zip une implémentation de cette solution. Les éléments constituant la liste sont des Villes dont on stocke quelques caractéristiques : nom, température moyenne, longitude, latitude.

Les fichiers VilleEmpilable.c et VilleEmpilable.h contiennent la déclaration du type Ville ainsi que des fonctions relatives à ce type.
Les fichiers PileVoid.c et PileVoid.h contiennent la déclaration du type Tcellule ainsi que des fonctions relatives à ce type et à la gestion de la pile.
Le fichiers DemoPileVoid.c contient le main de test des différentes fonctions.

Le code de la fonction afficherElement dépend du type des éléments stockés. Cette fonction, utilisée dans la fonction afficherPile, doit être implémentée (codée) par le développeur qui utilise la gestion des piles. On ne peut donc pas avoir dans un même programme des piles d'éléments différents puisqu'il faudrait plusieurs fonctions afficherElement. La programmation objet, abordée plus tard dans votre cursus, vous donnera une solution élégante à ce problème.

Télécharger l'archive.

1. **Compléter** le Makefile en ajoutant la règle manquante afin de **compiler** le projet. **Testez le.**

2. **Compléter** le programme afin d'ajouter le champ « pays » dans la structure de donnée.

3. **Coder** la fonction :
   `int sontEgaux (const Element *e, const void* cle)`

   qui compare un élément à une cle de recherche. Ici, la caractéristique de comparaison sera le nom de la ville. On utilisera la fonction de cette façon :
   `if (sontEgaux(e, "Paris"))`…
   Cette fonction sera utilisée plus tard pour rechercher un élément dans la pile.
   Comme pour la fonction afficherElement, le prototype sera déclaré dans PileVoid.h mais l'implémentation se retrouvera dans VilleEmpilable.c

4. **Coder** la fonction :
   `Element *rechercherElement (const TCellule * p, const void *cle)`

   qui recherche, dans la pile, la ville dont le nom est passé par le pointeur « cle ». La fonction retourne un pointeur sur cet élément ou NULL si non trouvé.

5. **Coder** la fonction :
   `Tcellule *chargerBaseVille (const char* fichier)`

   qui lit un fichier de villes et charge les éléments dans la pile. La fonction retourne la nouvelle pile. Vous trouverez dans l'archive une base de données au format csv : cities.csv

6. **(facultatif) Coder** la fonction :
   `Tcellule *supprimerElement (const TCellule * p, const void *cle)`

   qui supprime une ville de la pile dont le nom est passé par le pointeur « cle ».

5. Table de hachage et liste chainée.

La gestion par pile impose des recherches avec une complexité en O(n). Cela veut dire que le nombre d'opérations pour rechercher est proportionnel au nombre d'éléments stockés. Nous avons déjà vu des recherches plus performantes, par exemple en O(log(n)) lors de la recherche du zéro d'une fonction par dichotomie. On se propose ici d'implémenter la gestion d'une table de hachage qui permet la recherche en O(1). C'est à dire que la recherche ne dépend pas (ou très peu) du nombre d'éléments stockés.

On désire stocker des informations sur des villes comme au paragraphe précédent.

Dans sa version simple, une table de hachage est représentée par un tableau d'éléments. Chaque élément est inséré à une position donnée par une fonction de hachage que nous appellerons « *fh* ». Prenons par exemple une fonction de hachage qui retournerait la somme des codes ASCII composant le nom de la ville, modulo la taille de la table.
Par exemple pour une table à 5 éléments,
fh(«Nice») = (78+105+99+101)%5 = 3
fh(« Caen ») = (67+97+101+110)%5 = 0

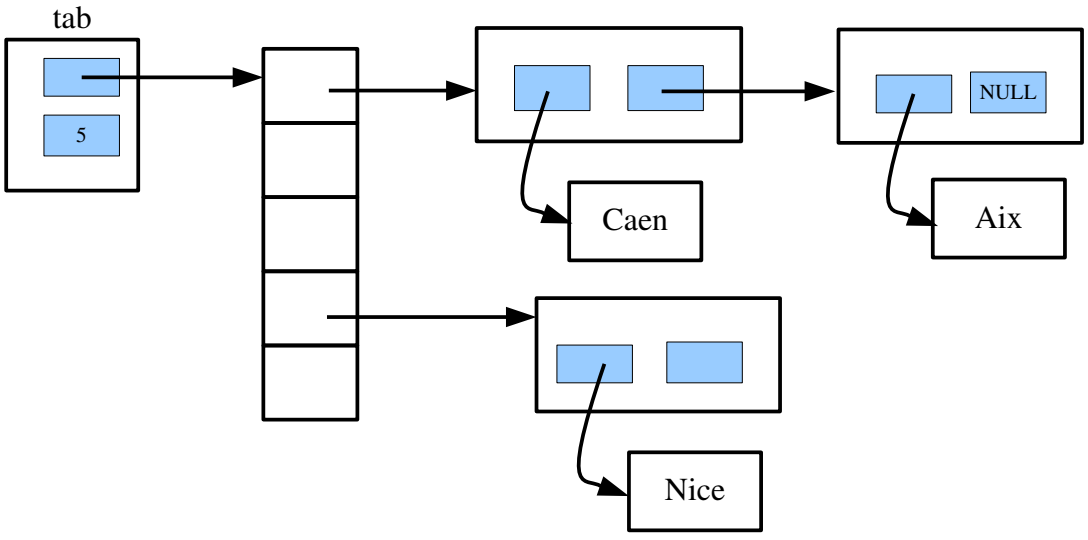
Ce qui permettrait de stocker les éléments ainsi :

0	Caen
1	
2	
3	Nice
4	

Bien sûr, il peut arriver qu'il y ait des collisions lorsque le calcule de hachage donne un même résultat. Par exemple fh(« Aix ») = 0, ce qui est le même index que Caen.

Une façon élégante de gérer le problème est d'insérer dans une liste les éléments qui ont le même index.

Ce qui donnerait :



La table de hachage « tab » est représentée par une structure de données à 2 champs de type « THachage » :
- un entier taille permettant de stocker le nombre de cases de la table.
- un pointeur de type Tcellule** vers un tableau de liste d'éléments du paragraphe précédent.

Reprendre l'intégralité du projet GestionPileVoid_eleve.zip, et le copier dans un nouveau répertoire. Nous y ajouterons TableHachage.c et TableHachage.h qui permettront de gérer la table de hachage.
Modifier le Makefile en conséquence.

Écrire les fonctions suivantes et les valider (à chaque étape) dans une fonction main ( ):

- `int hashCode (const char* cle, int n)` qui retourne la somme des codes ASCII de cle modulo

n ;

- *THachage creerTab(* qui retourne une table de hachage de n cases ;
- *void afficherTab(THachage t)* qui affiche la table de hachage ;
- *void ajouterElement(THachage t, const Element *e)* qui ajoute un élément à la table de hachage.
- *Element *rechercherElement (THachage t, const void *cle)* qui recherche, dans la table, la ville dont le nom est passé par le pointeur « cle ». La fonction retourne un pointeur sur cet élément ou NULL si non trouvé.
- *THachage chargerBaseVille (const char* fichier)* qui lit un fichier de villes et charge les éléments dans la table de hachage. La fonction retourne la nouvelle table.

Facultatif :

- *void supprimerElement (THachage t, const void *cle)* qui supprime une ville de la table dont le nom est passé par le pointeur « cle ».
- *void sauvegarderTable (THachage t, const char* fichier)* qui sauvegarde la table dans un fichier csv dont le nom est passé en paramètre.