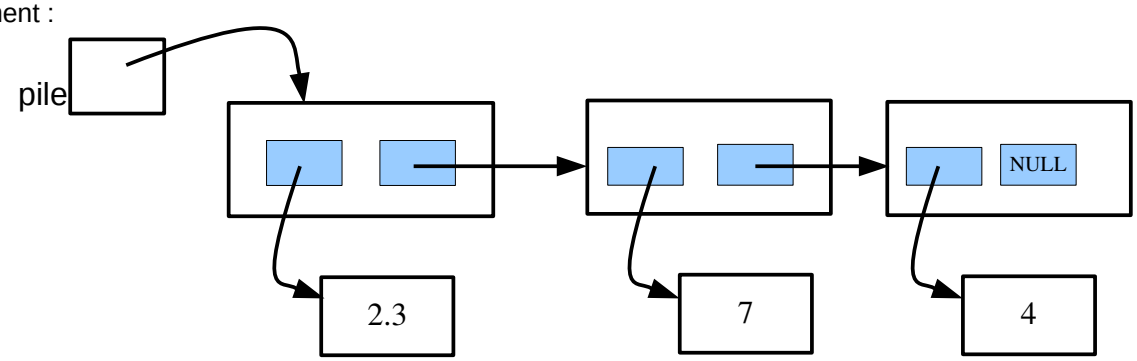
TP 10 – Structures, tableaux dynamiques et images

1. Pile générique

Dans le TP7, 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 des caractéristiques la déclaration du type Tcellule ainsi que des fonctions relatives à ce type zt 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 une nouvelle 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 ».

2. Table de hachage et liste chaînée.

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 les éléments de ville du 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

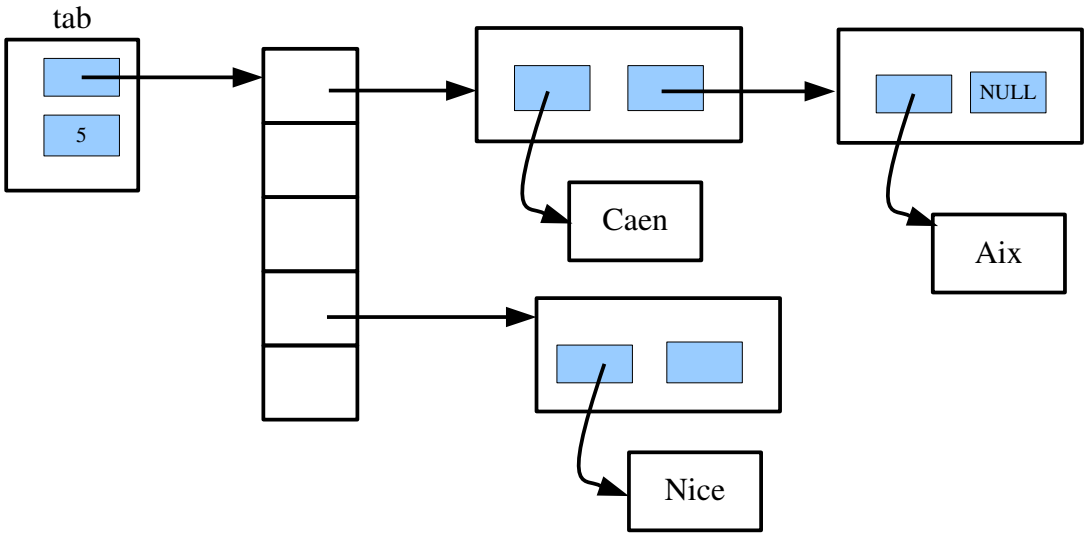
Ce qui permettrait de stocker les éléments ainsi :

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(int n)* 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 *rechercherElementTab (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 chargerBaseVilleTab (const char* fichier, int n)* qui lit un fichier de villes et charge les éléments dans une nouvelle table de hachage. La fonction retourne la nouvelle table. Il est d'usage de construire une table de hachage de taille double du nombre d'éléments à stocker pour éviter les collisions.

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.