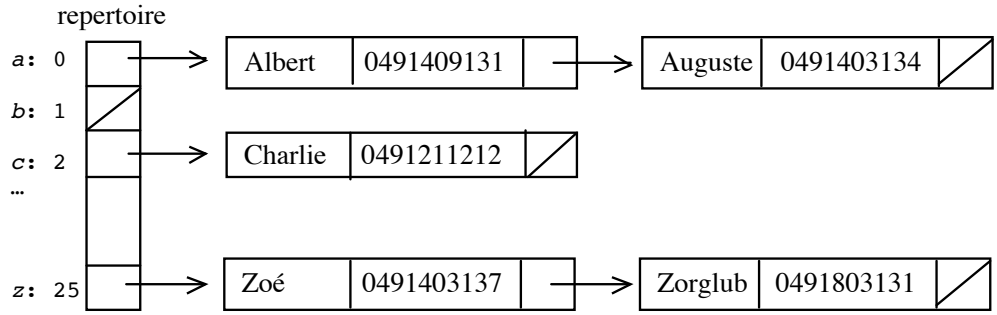
TD 10 : Listes chainées (*et fonction de Hashcode*)

On Souhaite gérer un annuaire téléphonique en minimisant l'occupation mémoire. Aussi, on se propose d'utiliser les structures de données suivantes :



Chaque lettre de l'alphabet donne accès à une "page" du répertoire téléphonique (on définit ainsi une fonction de hashcoding[1]) ; une "page" du répertoire est représentée par une liste chaînée éventuellement vide (cas de la page 'b') dont l'adresse du 1er élément est contenue dans la table de répertoire.

Soit les déclarations suivantes :

```
typedef struct triplet
{ char nom[20] ;
  char tel[11] ;
  struct triplet * suiv ;
} Triplet, *Ptriplet ;

Ptriplet repertoire[26] ;   /* tableau des têtes de listes */

Ptriplet allouer3(char n[20], char phone[10], Ptriplet suivant) ;
```

Ecrivez la fonction `allouer3` définie par le prototype ci-dessus. Cette fonction a pour rôle d'allouer un Triplet, de remplir ses champs et de retourner l'adresse de ce Triplet.

Ecrivez la fonction main qui initialise au départ la table *répertoire* à NULL et qui propose ensuite le menu suivant :

- *ajouter* quelqu'un dans le répertoire,
- *afficher* tout le répertoire dans l'ordre des listes,
- *chercher* quelqu'un dans le répertoire et afficher son n° ,

Pour la suite du programme, on procèdera en deux étapes.

Etape 1 :

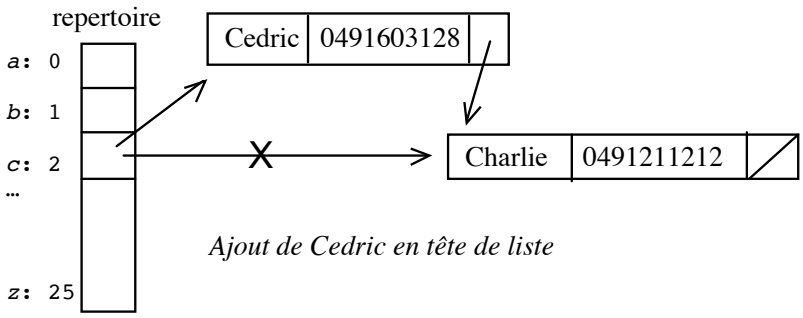
Dans un premier temps, les listes chaînées seront gérées comme des piles :

- les ajouts se feront toujours en tête de liste (on ne respecte pas l'ordre alphabétique des noms propres dans la liste).
- La recherche d'une personne se fera séquentiellement dans une liste.
- L'affichage se fera avec un parcours simple des 26 listes (l'ordre alphabétique ne sera donc pas respecté).

Ecrivez les trois fonctions qui permettent de faire les opérations proposées par le menu.

---

[1] Une fonction de **hashcode** permet de restreindre l'espace de recherche et de réduire ainsi les temps de calcul. L'idéal est de faire une étude statistique des données pour définir des sous-ensembles de tailles comparables (équirépartition) et aussi petits que possible. Dans le cas des noms propres, un aiguillage sur la première lettre est très simple mais cela crée d'importants déséquilibres (il y a peu de nom commençant par y…). Il vaut mieux proposer comme accès des fonctions du type « la somme des lettres d'un nom modulo N », N étant le nombre de sous-ensembles que l'on veut créer (plus N est grand, plus l'accès sera rapide).



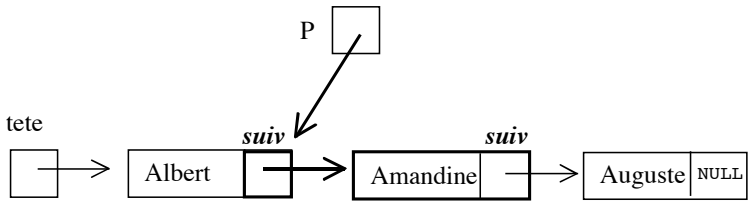
Etape 2 :
On modifiera le programme pour que les listes chaînées soient construites en respectant l'ordre alphabétique des noms (l'ajout d'un élément peut donc se faire n'importe où dans la liste chaînée. Par exemple, Amandine doit s'insérer entre Albert et Auguste).

- Modifiez la fonction *ajouter* écrite dans la partie 1 pour que les listes chaînées respectent cet ordre alphabétique.
- Ajoutez la fonction *supprimer* quelqu'un dans le répertoire.

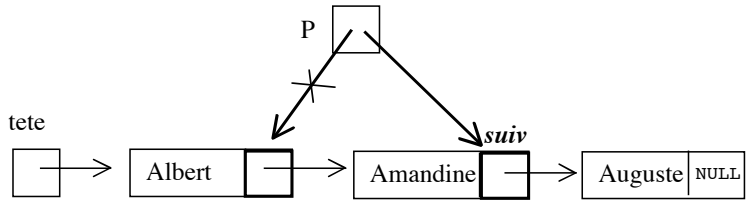
On constatera dans les situations ci-dessous qu'une variable P de type Ptriplet* peut contenir aussi bien l'adresse de la tête de liste que l'adresse d'un champ `suiv`. Cette variable P permettra de modifier indifféremment dans la liste chaînée :

- soit le lien sur le premier triplet (modification de la tête de liste)
- soit le lien sur un triplet quelconque (modification d'un champ suiv).

Accéder à un champ du triplet indirectement désigné par P se fait avec `(*P)->nom (*P)->tel` ou `(*P)->suiv`



Avancer dans la liste chaînée se traduit par : `P = &((*P)->suiv)`



Modifier le lien courant dans la liste se traduit par `*P =` *adr du nouvel élément*