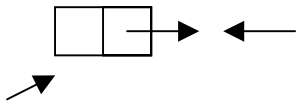
Les listes

1. Déclaration :

Il faut tout d'abord déclarer la 'brique' de base dans laquelle l'information va être stockée :

```
struct couple
{
     int info;
     struct couple *suiv;
}
```

Si l'on déclare un élément x de type struct couple (par `struct couple x;`), on a :

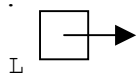


pointeur vers la 'brique' suivante (champs `x->suiv`)

champs 'info' qui va contenir l'information, on y accède par `x->info`

```
typedef struct couple *liste, brique;
```

Ici on a défini 2 nouveaux types : un type brique qui créera une brique (voir dessin ci-dessus), et un type liste qui sera un pointeur sur une brique. La déclaration de variable `liste L;` créera :



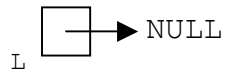
2. Initialisation :

Il faut toujours penser à initialiser une liste qui vient d'être créée. Pour cela on va utiliser une constante prédéfinie (NULL) se trouvant dans la bibliothèque stdio.h (ajouter `#include <stdio.h>`).

Récapitulons la phase de déclaration et l'initialisation :

```
#include <stdio.h>

struct couple
{
     int info;
     struct couple *suiv;
}

typedef struct couple *liste, brique;

liste L;

L=NULL;
```

La représentation graphique de la liste l est alors la suivante :



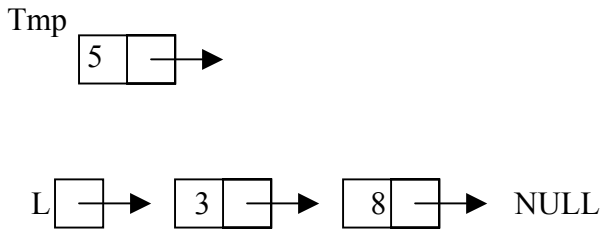
3. Test de vacuité :

Nous venons de voir comment initialiser une liste (la rendre 'vide'). Donc le test de vacuité (savoir si la liste est vide) portera sur la constante NULL :

```
if (L==NULL)
     printf("La liste est vide …\n");
else
     printf("La liste contient un (ou des) élément(s) …\n");
```

4. Ajout d'un élément en tête de liste :

On déclare une brique que l'on remplit (le nouvel élément). Ex :

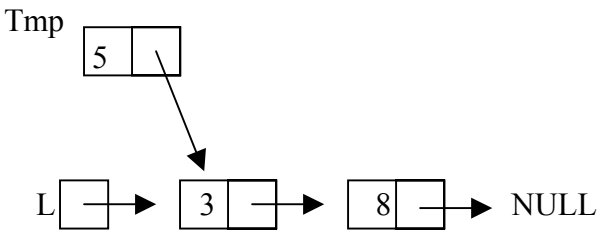


Pour cela il faut allouer l'espace mémoire de cet élément avec un malloc (memory allocation) :

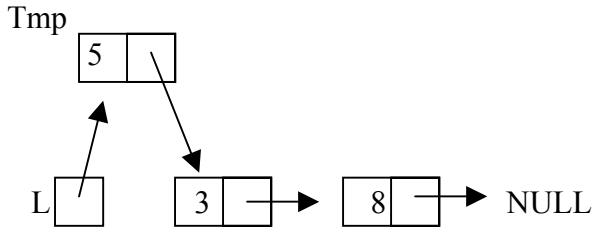
```
Tmp=(liste) malloc (sizeof(brique));
```

(en toute rigueur il faudrait tester s'il y a assez d'espace en mémoire …)

Il faut maintenant lier la brique à la liste : son suivant sera l'objet (la brique) pointé(e) par L (`Tmp->suiv=L`)



Maintenant le début de la liste (L) doit pointer sur le nouvel élément Tmp (donc `L=Tmp`) :

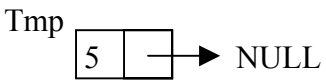


Pour une liste L créée et initialisée, et une brique Tmp déclarée, les instructions sont donc :

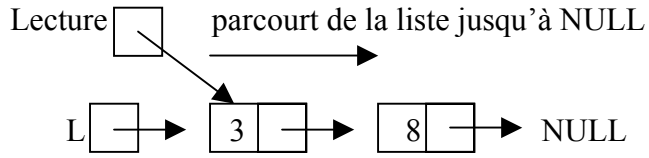
```
Tmp->info='nouvelle valeur';
Tmp->suiv=L;
L=Tmp;
```

5. Ajout d'un élément en queue de liste :

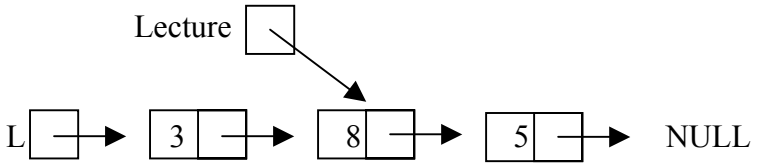
On doit tout d'abord créer le nouvel élément Tmp qui sera en queue de liste (avec `Tmp->suiv=NULL` puisqu'il s'agit du dernier élément) :



Il faut ensuite parcourir toute la liste jusqu'à la fin. Pour cela nous allons utiliser un nouveau pointeur sur la liste qui nous servira de tête de lecture (si on utilise le pointeur L, on perd la tête de la liste, on ne pourra plus la parcourir à nouveau …). Le nouveau pointeur (appelé Lecture) sera donc naturellement initialisé à l (`Lecture=L`), ce qui donne :



Donc, lorsque `Lecture->suiv=NULL` on sait que l'on a atteint la fin de la liste et que l'on peut insérer le nouvel élément (`Lecture->suiv=Temp`).



Le code est donc le suivant :

```
liste Lecture, L, Tmp;
…
Tmp=(liste) malloc (sizeof(brique));
Tmp->info='nouvelle valeur';
Tmp->suiv=NULL;
Lecture=L;
While (Lecture->suiv != NULL)
  Lecture=Lecture->suiv;
Lecture->suiv=Tmp;
```

6. Affichage de la liste :

Pour l'affichage de la liste, tout comme pour l'insertion en queue de liste, nous allons utiliser un pointeur qui fera office de tête de lecture. Là également nous allons parcourir la liste jusqu'à la fin tout en affichant le champs `Lecture->info`.

```
Lecture=L;
While (Lecture != NULL)
{
  printf(«%d\n», Lecture->info);
  Lecture=Lecture->suiv;
}
```

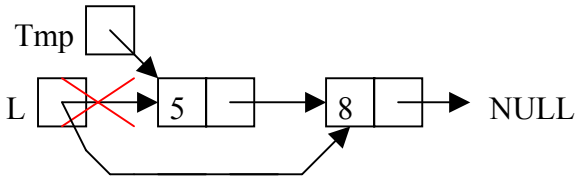
7. Recherche d'un élément dans la liste :

La recherche d'un élément dans la liste se fait comme dans les points 5 et 6 précédents en utilisant une tête de lecture. Seule différence : on s'arrête si l'on trouve l'élément recherché.

```
int trouve;
…
trouve=0;
Lecture=L;
While ((!trouve) && (Lecture != NULL))
{
  if (Lecture->info=='valeur recherchée')
    trouve=1;
  Lecture=Lecture->suiv;
}
if (trouve)
  printf(« L'élément recherché est dans la liste\n »);
else
  printf(« L'élément recherché n'est pas dans la liste\n »);
```

8. Suppression du premier élément de la liste :

Pour supprimer le premier élément, il suffit de conserver son adresse en mémoire par un pointeur Tmp qui nous servira à libérer la mémoire puis à 'sauter' cet élément dans la liste L.

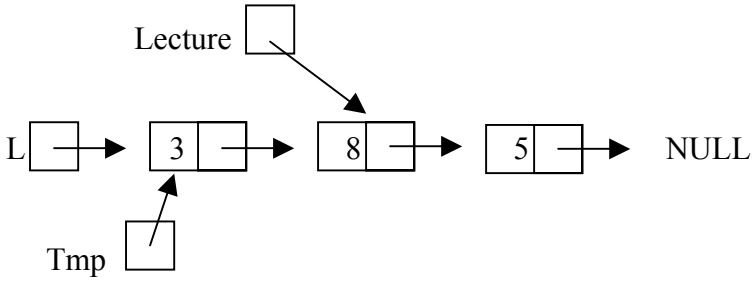


On obtient le programme suivant :

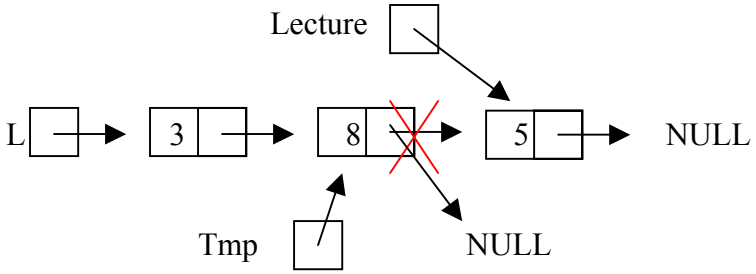
```
Tmp=L;
L=L->suiv;
free(Tmp);
```

9. Suppression du dernier élément de la liste :

On va parcourir la liste à l'aide de 2 pointeurs Lecture et Tmp (l'un sera en avance d'une brique par rapport à l'autre). Ceci nous permettra de conserver l'information permettant d'insérer l'élément au bon endroit.



Lorsque le suivant de Lecture pointe vers NULL on sait donc que Lecture pointe sur l 'élément à supprimer (de même que le suivant de Tmp). Il suffit donc de mettre `Tmp->suiv` à `NULL` et de libérer la mémoire occupée par Lecture :

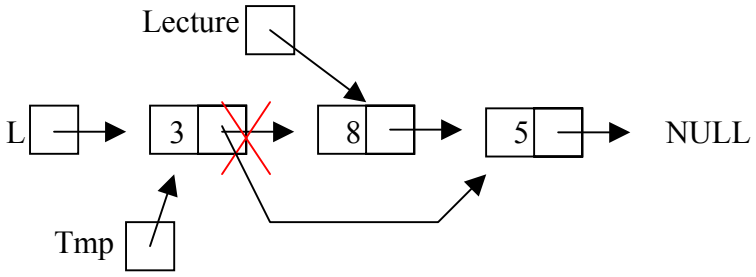


Le code permettant d'exécuter ceci est alors :

```
Lecture=L->suiv;
Tmp=L;
While (Lecture->suiv != NULL)
{
  Lecture=Lecture->suiv;
  Tmp=Tmp->suiv;
}
Tmp->suiv=NULL;
free(Lecture);
```

10. Suppression d'un élément choisi par l'utilisateur :

On se retrouve exactement dans le même cas que précédemment avec la suppression du dernier élément de la liste. Seule la condition d'arrêt du parcours de la liste va changer : on s'arrête lorsque l'on a trouvé l'élément à supprimer ou lorsque la liste est vide :



```
Lecture=L->suiv;
Tete=L;
if (Tete->info==’valeur recherchée’)
{
  free(Tete);
  Tete=Lecture;
  Printf(« Elément supprimé …\n »);
}
else
{
  trouve=0;
  while ((Lecture != NULL) && (!trouve))
  {
    if (Lecture->info==’valeur recherchée’)
    {
      Tete->suiv=Lecture->suiv;
      free(Lecture);
      trouve=1;
    }
    else
    {
      Lecture=Lecture->suiv;
      Tete=Tete->suiv;
    }
  }
  if (trouve)
    printf(« Elément supprimé …\n »);
}
```

11. Les fonctions :

Pour ajouter un élément dans une liste par une fonction, on fait exactement comme avec une variable normale : passage de paramètres par adresse. Ex :

```
void ajouter(liste *l, type_valeur valeur)
{
…
}

liste p;
ajouter(&p, ‘valeur’);
```

Conclusion :

Bon courage pour les examens … (vous n’avez pas besoin de chance, la chance se construit par le travail …).
Si vous avez la moindre question, le moindre problème, n’hésitez pas à m’écrire : tristan.colombo@laposte.net