# BE 2 : Listes dynamiques

**Principe.** Une liste dynamique est un ensemble d'éléments ordonnés auquel on peut ajouter ou supprimer des éléments à volonté. Ceci nécessite de pouvoir allouer et libérer la mémoire des éléments en question.

On se propose de créer une liste suivant le schéma suivant. Deux structures de données sont définies : la liste en elle-même qui pointe sur le premier et le dernier élément de la liste, et les différents nœuds de cette liste qui pointent chacun sur l'élément suivant dans la liste. C'est une liste chaînée de manière unidirectionnelle. Dans un premier temps, on supposera que les éléments sont des entiers (type `int`).

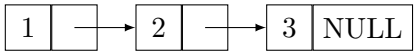



FIGURE 1 – Structure de liste chaînée

La structure de données associée est la suivante :

```
typedef struct list
{
  int item;           /* Élément inclus dans la liste  */
  struct list *next;  /* Pointeur sur l'élément suivant dans la liste */
} list;
```

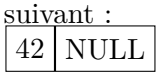
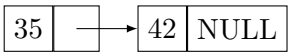
Une liste sera alors de type `list *`, c'est-à-dire un pointeur sur une structure `list`. La liste vide sera représentée par le pointeur `NULL`.

Nous vous fournissons un squelette : `be2-squelette-partie1.c`. Dans ce squelette, toutes les fonctions sont définies, mais avec un corps vide (des `return NULL;` ou `return 0;` pour que le compilateur ne râle pas). La fonction `main` contient une suite de tests qui devraient tous afficher `OK` à la fin du BE. Nous vous encourageons évidemment à compléter ces tests, si vous en ressentiez le besoin.

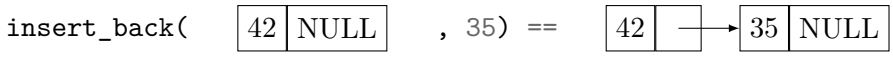
**Question 1** (Création d'une liste)**.**
Écrire une fonction `list* new_list()` qui renvoie une liste vide.

**Question 2** (Création d'un nouveau nœud)**.**
Écrire une fonction `list * new_node(int x)` qui crée un nœud d'une liste, avec une valeur donnée en paramètre.
Un appel à `new_node(42)` doit renvoyer un pointeur vers une structure ressemblant à :

**Question 3** (Ajout d'un élément en tête de liste)**.**
Écrire une fonction `list * insert_front(int x, list* l)` qui ajoute un élément en début de liste.
On doit obtenir le résultat suivant :

`insert_front(35,`  `) ==` 

**Question 4** (Ajout d'un élément en queue de liste)**.**
Écrire une fonction `list* insert_back(list* l, int x)` qui ajoute un élément en fin de liste.
On doit obtenir le résultat suivant :

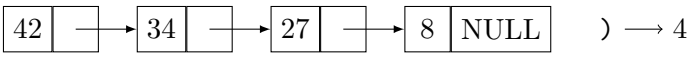



**Question 5** (Égalité de listes).
Écrire une fonction `int equal(list * l1, list * l2)` qui renvoie 1 si les deux listes sont égales (*i.e.* elles ont les mêmes éléments dans le même ordre), et 0 sinon.

**Question 6** (Appartenance).
Écrire une fonction `int contains(list * l, int x)` qui renvoie 1 si l'entier `x` appartient à la liste `l`, 0 sinon.

**Question 7** (Inclusion).
Écrire une fonction `int subset(list * l1, list * l2)` qui renvoie 1 si la liste `l1` est un sous-ensemble de `l2`, c'est-à-dire que tous les élements de la première liste apparaissent dans la deuxième (pas nécessairement dans le même ordre), 0 sinon.

**Question 8** (Taille d'une liste).
Écrire une fonction `int length(list * l)` qui renvoie la taille, c'est-à-dire le nombre d'éléments d'une liste.



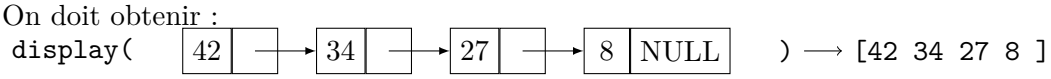

`length(NULL) = 0`

**Question 9** (Appliquer une fonction à tous les éléments de la liste).
Écrire une fonction `void iter(list * l, void (*fct)(int))` qui applique une fonction passée en paramètre à tous les éléments de la liste.
Le paramètre `fct` est un pointeur de fonction, qui prend en paramètre l'élément à traiter, et qui ne retourne rien.

**Question 10.**
Écrire une fonction `void display_int(int i)` qui prend un entier et l'affiche, suivie d'une espace.

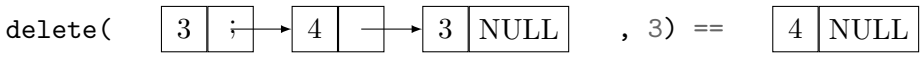
**Question 11.**
Écrire une fonction `void display_list(list * l)` qui affiche une liste d'entiers, en utilisant notamment les fonctions `iter` et `display_int`.

On doit obtenir :




**Question 12** (Libération d'une liste en mémoire).
Écrire une fonction `void free_list(list * l)` qui libère l'espace mémoire utilisé par une liste.

**Question 13** (Effacer un nœud de la liste).
Écrire une fonction `list* delete(list * l, int x);` qui efface un nœud de la liste. Cette fonction devra prendre en compte les modifications de chaînage des nœuds qui résultent de la modification de la structure de la liste. Un pointeur sur ce nœud est reçu en paramètre de la fonction.
On doit obtenir le résultat suivant :




**Question 14** (Tri à bulle de la liste).
Le tri à bulle consiste à scruter toutes les valeurs d'un tableau ou d'une liste en les comparant deux à deux avec leurs voisines et à les échanger si elles ne sont pas bien classées, puis à recommencer jusqu'à ce que toute la liste soit triée. Écrire une fonction `void bubble(list * l);` qui trie suivant un algorithme de tri à bulle la liste passée en paramètre.

**Indice** : il suffit d'échanger les valeurs contenues dans les nœuds, pas les mœuds eux-mêmes. Le chaînage ne devrait donc pas être impacté.

Voici une séquence d'opérations sur une liste :

```
[3, 5, 12, 2, 8, 1, 7]
[3, 5, 2, 12, 8, 1, 7]
[3, 2, 5, 12, 8, 1, 7]
[2, 3, 5, 8, 12, 1, 7]
[2, 3, 5, 8, 1, 12, 7]
[2, 3, 5, 1, 8, 12, 7]
[2, 3, 1, 5, 8, 12, 7]
[2, 1, 3, 5, 8, 12, 7]
[1, 2, 3, 5, 8, 12, 7]
[1, 2, 3, 5, 8, 7, 12]
[1, 2, 3, 5, 7, 8, 12]
```

**Question 15** (Listes polymorphes)**.**
On souhaite maintenant être capable de traiter des listes de n'importe quel type d'élément (pas seulement des entiers, comme c'est le cas jusqu'à présent). Pour ce faire, on modifie la structure `list` en remplaçant le type de `item` par `void*`. Ce type correspond à des pointeurs sur n'importe quel type de données.

Copiez le fichier qui contient vos définitions sur les listes d'entiers, et modifier dans cette copie le type de la structure de listes, ainsi que vos fonctions pour vous adapter à ce nouveau type.

Pour les fonctions qui testent l'égalité entre éléments (par exemple `contains` ou `delete`), il faut passer un nouveau paramètre de type `int (*eq)(void*, void*)` qui permet de tester l'égalité entre deux éléments.

Pour la fonction de tri, comme les éléments contenus dans la liste peuvent être de différents types, il est nécessaire de passer en paramètre de la fonction une fonction de comparaison de deux éléments. Cette fonction devra renvoyer :

- -1 si le premier élément est inférieur au second ;
- 0 si les deux éléments sont égaux
- 1 si le premier élément est supérieur au second.

Le prototype de la fonction devient :

```
void bubble(list * l, int (*compar)(void*, void*));
```

**Question 16.**
On propose d'utiliser ce type de listes polymorphes pour contenir des structures de type `etudiant`, dont voici la définition :

```
typedef struct etudiant {
  char * nom;
  int note;
} etudiant;
```

Dans la fonction `main`, créez une liste de 5 étudiants avec chacun un nom et une note différentes.

Écrivez les fonctions nécessaires pour trier cette liste par note croissante, puis affichez cette liste triée.

N'oubliez pas de libérer la mémoire utilisée par la liste avant de quitter le programme.