

1 Listes chaînées

1.1 Structure d'une liste chaînée

Une **liste chaînée** est une structure de données « formée de manière **réursive** ».

Une **liste** est :

- **soit** la **liste** vide,
- **soit** un élément suivi d'une **liste**.

Une liste permet de stocker une **suite d'éléments** de **même type** de manière **non contigue**.

Une liste d'**entiers** permettra de stocker des **entiers**, une liste de « **struct blah_t** » permettra de stocker des « **struct blah_t** ».

Ainsi, la **liste** [17;42;5] est la **liste** composée d'un élément 17 suivie de (la **liste** composée d'un élément 42 suivie de (la **liste** composée d'un élément 5 suivie de (la **liste** vide))).

En C une liste est encodée comme une chaîne de « **cellules** » dans laquelle on passe à l'élément suivant en suivant un **pointeur** vers la prochaine « **cellule** ».

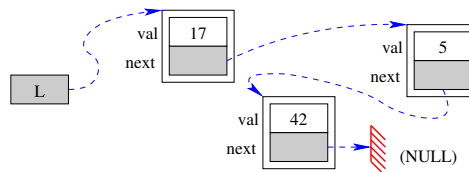
Une liste est matérialisée par un **pointeur** vers son (premier) élément : l'adresse de sa (première) « **cellule** ».

La **liste vide** est représentée par le pointeur **NULL**.

En C, une liste contenant des éléments de type « **un_type** » est représentée par une structure :

```
struct list {
    un_type val ;           // Le contenu.
    struct list *next ;    // Pointeur vers l'element suivant (ou NULL).
};
```

La liste [17;42;5] en C :



1.2 Intérêts des listes chaînées

Ajout d'un élément **en tête** d'une **liste** en temps constant. Permet de créer une **liste** contenant un élément de plus.

Extraction du **premier** élément d'une **liste** en temps constant. Permet de récupérer ce premier élément et la **liste** « qui reste ».

Avancer à l'élément suivant en temps constant.

Structure facilement **redimensionnable** (contrairement aux tableaux) : on crée / supprime des éléments **dynamiquement selon les besoins** sans affecter toute la structure de données (pas besoin de recopier/déplacer comme dans les tableaux).

Besoin d'allouer des zones de mémoire nombreuses (pour les éléments) mais de **petite taille** (contrairement à un tableau où il faut une zone **contigue** de la taille globale de tous les éléments).

1.3 Inconvénients des listes chaînées

Structure données un peu plus compliquée à manipuler que des tableaux.

Accès «direct» à un élément quelconque **coûteux** (nécessite le parcours de la liste).

(Moins immédiat : fragmentation de la mémoire).

1.4 Opérations sur les listes

Le schéma général du traitement récursif d'une liste est induit par la structure d'une liste. Deux cas possibles :

- **soit** la **liste** est **vide** : traitement de fin,
- **soit** la liste est composée d'un **élément** suivi d'une **liste «reste»** : on traite l'**élément**, puis on recommence le traitement sur la **liste «reste»** (ou inversement).

```
Traitement (struct list_t *l)
{
    if (l == NULL) {
        Rien à faire ou traitement de fin de liste
    }
    else {
        e = l->val
        travailler avec e (et autre chose si besoin)
        Traitement (l->next)
    }
}
```

```
Traitement (struct list_t *l)
{
    if (l == NULL) {
        Rien à faire ou traitement de fin de liste
    }
    else {
        Traitement (l->next)
        e = l->val
        travailler avec e (et autre chose si besoin)
    }
}
```

Certains traitements peuvent être écrits en **itératif** (utilisation d'une **boucle** au lieu de la récursion). Schéma général à deux cas possibles :

- si le pointeur représentant l'élément courant est **NULL** : fin de traitement
- sinon traiter l'élément courant, affecter le **pointeur** représentant l'élément courant en suivant le champ «**next**», recommencer le traitement.

```
Traitement (struct list_t *l)
{
    while (l != NULL) {
        e = l->val
        travailler avec e (et autre chose si besoin)
        l = l->next ;
    }
}
```