Cours de Programmation Impérative
Les listes chaînées

Julien David

A101 - david@lipn.univ-paris13.fr

Allocation dynamique

Dans le cours précédent nous avons vu

- Comment réserver de l'espace mémoire.
- Comment modifier la taille d'un espace mémoire.
- Comment libérer cet espace.

Grâce à cela, on a vu

- Comment créer des tableaux "à la volée", en ajustant leurs tailles selon le besoin.

Allocation dynamique

Dans le cours précédent nous avons vu

- Comment réserver de l'espace mémoire.
- Comment modifier la taille d'un espace mémoire.
- Comment libérer cet espace.

Grâce à cela, on a vu

- Comment créer des tableaux "à la volée", en ajustant leurs tailles selon le besoin.

Les ensembles

Types d'ensembles

En TD, nous avons défini des ensembles d'entiers. Mais on aurait pu par exemple :

- Définir un ensemble de voitures $\rightarrow$ un garage.
- Définir un ensemble d'images et de sons $\rightarrow$ une vidéo.
- N'importe quel type de données...

Il n'y a pas que les ensembles

La notion d'ensemble implique que l'on ne stocke pas deux fois le même élément, c'est à dire qu'on ne peut avoir :

$$\{3, 3, 5, 6\}.$$

Mais on peut également considérer des structures où deux occurences d'un même élément peuvent apparaître.

Les ensembles

Types d'ensembles

En TD, nous avons défini des ensembles d'entiers. Mais on aurait pu par exemple :

- Définir un ensemble de voitures $\rightarrow$ un garage.
- Définir un ensemble d'images et de sons $\rightarrow$ une vidéo.
- N'importe quel type de données...

Il n'y a pas que les ensembles

La notion d'ensemble implique que l'on ne stocke pas deux fois le même élément, c'est à dire qu'on ne peut avoir :

$$\{3, 3, 5, 6\}.$$

Mais on peut également considérer des structures où deux occurences d'un même élément peuvent apparaître.

Les tableaux dynamiques

Manipulation

On a vu les fonctions suivantes pour les tableaux non-triés :

- Initialiser : on a fixé une taille initiale au tableau qui stocke les données.
  On a peut-être perdu de la place pour rien.
- Détruire : $\Theta(1)$
- Tester si un tableau est vide : $\Theta(1)$
- Connaître le nombre d'éléments dans le tableaux : $\Theta(1)$
- Ajouter un élément :
  - temps constant si le tableau n'est pas plein
  - temps linéaire si le tableau est plein.
- Supprimer un élément : $\mathcal{O}(n)$
  Il faut décaler toutes les valeurs qui suivent celle que l'on a supprimé.
- Supprimer le dernier élément ajouté : $\Theta(1)$

Le problème de la taille supplémentaire

Demander de l'espace supplémentaire

Pour l'instant, lorsqu'un tableau est plein et que l'on souhaite ajouter un élément :

- on utilise la fonction `realloc` (Complexité en temps : $\Theta(n)$)

Exemple

On veut ajouter la valeur 3 au tableau

6	8	3	8	4	12	5	1

On crée un tableau plus grand.

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

On recopie le petit tableau dans le grand.

6	8	3	8	4	12	5	1	?	?	?	?	?	?	?	?

On ajoute la valeur 3.

6	8	3	8	4	12	5	1	3	?	?	?	?	?	?	?

Le problème de la taille supplémentaire

Demander de l'espace supplémentaire

Pour l'instant, lorsqu'un tableau est plein et que l'on souhaite ajouter un élément :

- on utilise la fonction `realloc` (Complexité en temps : $\Theta(n)$)

Exemple

On veut ajouter la valeur 3 au tableau

6	8	3	8	4	12	5	1

On crée un tableau plus grand.

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

On recopie le petit tableau dans le grand.

6	8	3	8	4	12	5	1	?	?	?	?	?	?	?	?

On ajoute la valeur 3.

6	8	3	8	4	12	5	1	3	?	?	?	?	?	?	?

Le problème de la taille supplémentaire

Demander de l'espace supplémentaire

Pour l'instant, lorsqu'un tableau est plein et que l'on souhaite ajouter un élément :

- on utilise la fonction `realloc` (Complexité en temps : $\Theta(n)$)

Exemple

On veut ajouter la valeur 3 au tableau

6	8	3	8	4	12	5	1

On crée un tableau plus grand.

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

On recopie le petit tableau dans le grand.

6	8	3	8	4	12	5	1	?	?	?	?	?	?	?	?

On ajoute la valeur 3.

6	8	3	8	4	12	5	1	3	?	?	?	?	?	?	?

Le problème de la taille supplémentaire

Demander de l'espace supplémentaire

Pour l'instant, lorsqu'un tableau est plein et que l'on souhaite ajouter un élément :

- on utilise la fonction `realloc` (Complexité en temps : $\Theta(n)$)

Exemple

On veut ajouter la valeur 3 au tableau

6	8	3	8	4	12	5	1

On crée un tableau plus grand.

?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?

On recopie le petit tableau dans le grand.

6	8	3	8	4	12	5	1	?	?	?	?	?	?	?	?

On ajoute la valeur 3.

6	8	3	8	4	12	5	1	3	?	?	?	?	?	?	?

Le problème de la taille supplémentaire

Demander de l'espace supplémentaire

Pour l'instant, lorsqu'un tableau est plein et que l'on souhaite ajouter un élément :

- soit on décide de réserver juste une seule case.
  - La quantité de mémoire sera toujours adaptée au besoin.
  - Mais à chaque fois, on devra recopier tout le tableau.
- soit on réserve plusieurs cases.
  - On risque de réserver plus d'espace que nécessaire.
  - Les ajouts suivants seront rapides.

Le problème de la taille supplémentaire

Est-on obligé de recopier tout le tableau ?

Si on pouvait juste ”brancher” différents espaces mémoires les uns aux autres, il n’y aurait pas besoin de recopier un tableau lorsqu’on veut l’agrandir.

Le problème de la taille supplémentaire

Si seulement on pouvait dire. . .

- voici le début du tableau,
- voici la **suite** du tableau.

Il suffit d'utiliser des pointeurs ! ! !

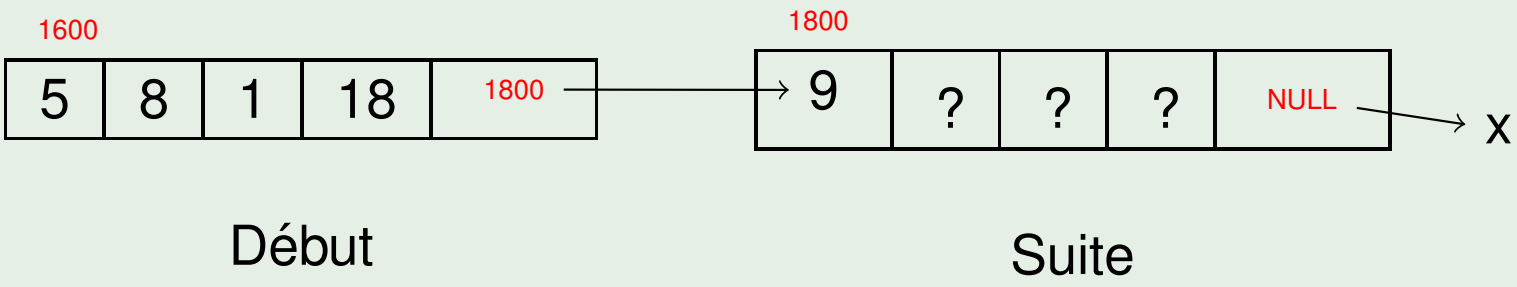
(Note : sur le schéma, quand une flèche va vers "x", on signifie que le pointeur ne pointe nul part)

Le problème de la taille supplémentaire

Si seulement on pouvait dire. . .

- voici le début du tableau,
- voici la **suite** du tableau.

Il suffit d'utiliser des pointeurs ! ! !



(Note : sur le schéma, quand une flèche va vers "x", on signifie que le pointeur ne pointe nul part)

Les listes chaînées

Les listes chaînées

L'idée

- On va crée une structure de donnée que l'on peut agrandir en temps constant.
- On ajoute juste une case mémoire à chaque étape.

En revanche

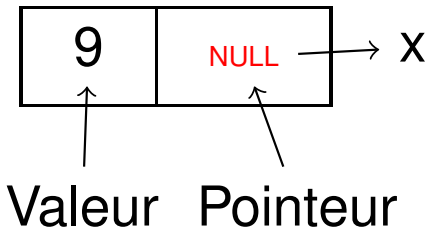
- On va voir que si certaines fonctions deviennent plus rapides, d'autres deviennent plus lentes.

Les listes chaînées : les maillons

L'idée : les maillons

Un `maillon` d'une liste est constituée de :

- une valeur
- un pointeur vers le maillon suivant.



Les maillons

```
1  #ifndef _LISTE_H
2  #define _LISTE_H
3
4  struct maillon_s{
5    int valeur;
6    struct maillon_s * suivant;
7  };
8  typedef struct maillon_s maillon_t;
9
0  #endif
```

Remarque : on utilise la notion de maillon dans sa propre définition.

Les maillons : création

```
1 maillon_t * maillon_creer(int val){
2   maillon_t * res=(maillon_t *)malloc(sizeof(maillon_t));
3   res->valeur=val;
4   res->suivant=NULL;
5   return res;
6 }
```

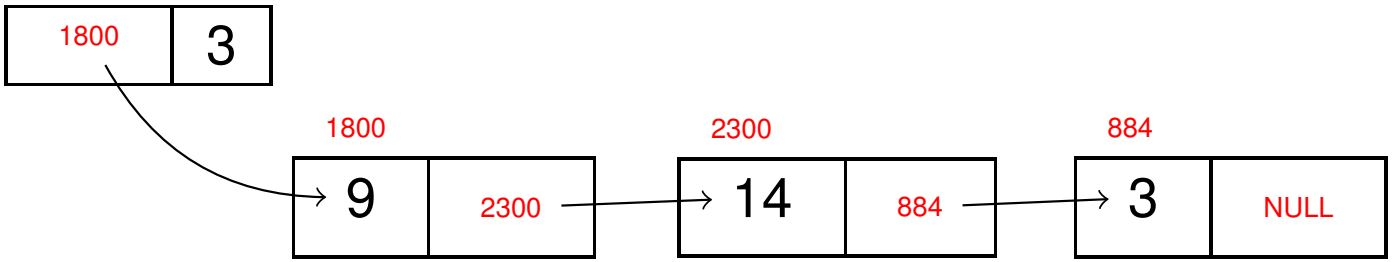
Les maillons : destruction

```
1 void maillon_detruire(maillon_t *m){
2   free(m);
3 }
```

Liste chaînée

Une liste est constituée de :

- un pointeur sur un maillon, qui pointe sur un maillon, qui pointe sur un maillon, . . .
- un entier pour stocker la taille de la liste*



∗ l'entier qui permet de connaître la taille de la liste n'est pas toujours utilisé dans la définition

Les listes chaînées

```
1 struct liste_s{
2   maillon_t * debut;
3   int taille;
4 };
5 typedef struct liste_s liste_t;
```

Les listes chaînées

Fonctions qui manipulent les listes

- Initialiser
- Détruire
- Tester si la liste est vide
- Connaître la taille de la liste
- Tester si un élément appartient à la liste
- Afficher la liste
- Accéder au i^{eme} élément.
- Ajouter un élément
- Extraire un élément

Les listes chaînées : initialisation

Initialisation

Une liste vide contient un pointeur de valeur NULL.

```
1 liste_t * liste_initialiser(){
2   liste_t * res=(liste_t *)malloc(sizeof(liste_t));
3   res->debut=NULL;
4   res->taille=0;
5   return res;
6 }
```

Complexité

- Temps : $\Theta(1)$
- Espace : $\Theta(1)$

Les listes chaînées : liste vide

Tester si une liste est vide

```
1 int liste_vide(liste_t l){
2   if(l.taille==0)
3     return 1;
4   return 0;
5 }
```

Complexité

- Temps : $\Theta(1)$
- Espace : $\Theta(1)$

Les listes chaînées : cardinalité

Taille de la liste

```
1 int liste_card(liste_t l){
2   return l.taille;
3 }
```

Complexité

- Temps : $\Theta(n)$
- Espace : $\Theta(1)$

Les listes chaînées : affichage

Affichage

- Comme avec un tableau on utilise une boucle `for` pour parcourir la liste,
- en revanche, la variable de boucle est un pointeur.

```
1 void liste_afficher(liste_t l){
2   maillon_t * m;
3   for(m=l.debut;m!=NULL;m=m->suivant)
4     printf("%d ",m->valeur);
5   printf("\n");
6 }
```

Complexité

- Temps : $\Theta(n)$
- Espace : $\Theta(1)$

Les listes chaînées : affichage

Affichage

- Comme avec un tableau on utilise une boucle `for` pour parcourir la liste,
- en revanche, la variable de boucle est un pointeur.

```
1 void liste_afficher(liste_t l){
2   maillon_t * m;
3   for(m=l.debut;m!=NULL;m=m->suivant)
4     printf("%d ",m->valeur);
5   printf("\n");
6 }
```

Complexité

- Temps : $\Theta(n)$
- Espace : $\Theta(1)$

Les listes chaînées : appartient

Tester si un élément x appartient à la liste

1. Comme pour les tableaux, on utilise une boucle `while`.

```
1 int liste_appartient(liste l,int x){
2   maillon_t * m=l.debut;
3   while(m!=NULL){
4     if(m->valeur==x)
5       return 1;
6     m=m->suivant;
7   }
8   return 0;
9 }
```

Complexité

- Temps : $\Omega(1)$ et $\mathcal{O}(n)$
- Espace : $\Theta(1)$

Les listes chaînées : appartient

Tester si un élément x appartient à la liste

1. Comme pour les tableaux, on utilise une boucle `while`.

```
1 int liste_appartient(liste l,int x){
2   maillon_t * m=l.debut;
3   while(m!=NULL){
4     if(m->valeur==x)
5       return 1;
6     m=m->suivant;
7   }
8   return 0;
9 }
```

Complexité

- Temps : $\Omega(1)$ et $\mathcal{O}(n)$
- Espace : $\Theta(1)$

Les listes chaînées : accès

Accès au i^{eme} élément

- Contrairement aux tableaux, on ne peut pas accéder directement à la i^{eme} valeur.
- exemple : pour accéder au 3^{eme} élément, il faut accéder au maillon **suivant** du maillon **suivant** du **début** de la liste.

```
1 maillon_t * liste_acces(liste_t l, int pos){
2   maillon_t * m;
3   for(m=l.debut;m!=NULL&& pos!=0;pos--)
4     m=m->suivant;
5   return m;
6 }
```

Complexité

- Temps : $\Theta(pos)$
- Espace : $\Theta(1)$

Les listes chaînées : accès

Accès au i^{eme} élément

- Contrairement aux tableaux, on ne peut pas accéder directement à la i^{eme} valeur.
- exemple : pour accéder au 3^{eme} élément, il faut accéder au maillon **suivant** du maillon **suivant** du **début** de la liste.

```
1 maillon_t * liste_acces(liste_t l, int pos){
2   maillon_t * m;
3   for(m=l.debut;m!=NULL&& pos!=0;pos--)
4     m=m->suivant;
5   return m;
6 }
```

Complexité

- Temps : $\Theta(pos)$
- Espace : $\Theta(1)$

Les listes chaînées : accès

Problème posé par l'accès

si l'accès au i^{eme} maillon a un coût linéaire,

- l'ajout d'un maillon à la position i aussi.

Heureusement

- l'ajout d'une valeur dans un tableau à la position i est aussi linéaire (il faut décaler les valeurs suivantes).

Mais comme on cherche juste à stocker un ensemble d'éléments non trié :

- on peut ajouter les éléments au début de la liste.

Les listes chaînées : ajouter un maillon

Ajout d'un maillon au début de la liste

Si un maillon devient le début d'une liste :

1. la "suite" du nouveau maillon devient le début de la liste précédente
2. le pointeur de départ doit contenir l'adresse du nouveau maillon.

Codage

On veut :

1. modifier un maillon (celui qu'on ajoute)
2. modifier une liste.

On prend donc en entrée :

1. un pointeur sur un maillon.
2. un pointeur sur une liste.

Les listes chaînées : ajouter un maillon

Ajout d'un maillon au début de la liste

Si un maillon devient le début d'une liste :

1. la "suite" du nouveau maillon devient le début de la liste précédente
2. le pointeur de départ doit contenir l'adresse du nouveau maillon.

Codage

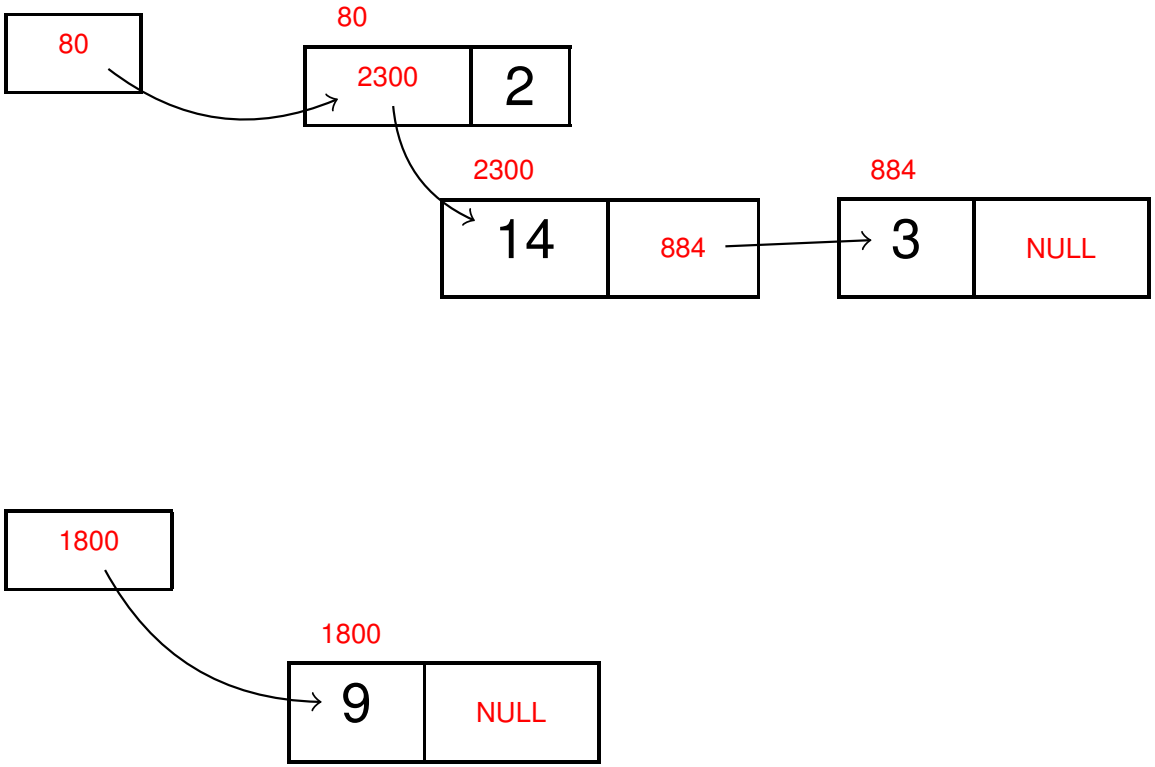
On veut :

1. modifier un maillon (celui qu'on ajoute)
2. modifier une liste.

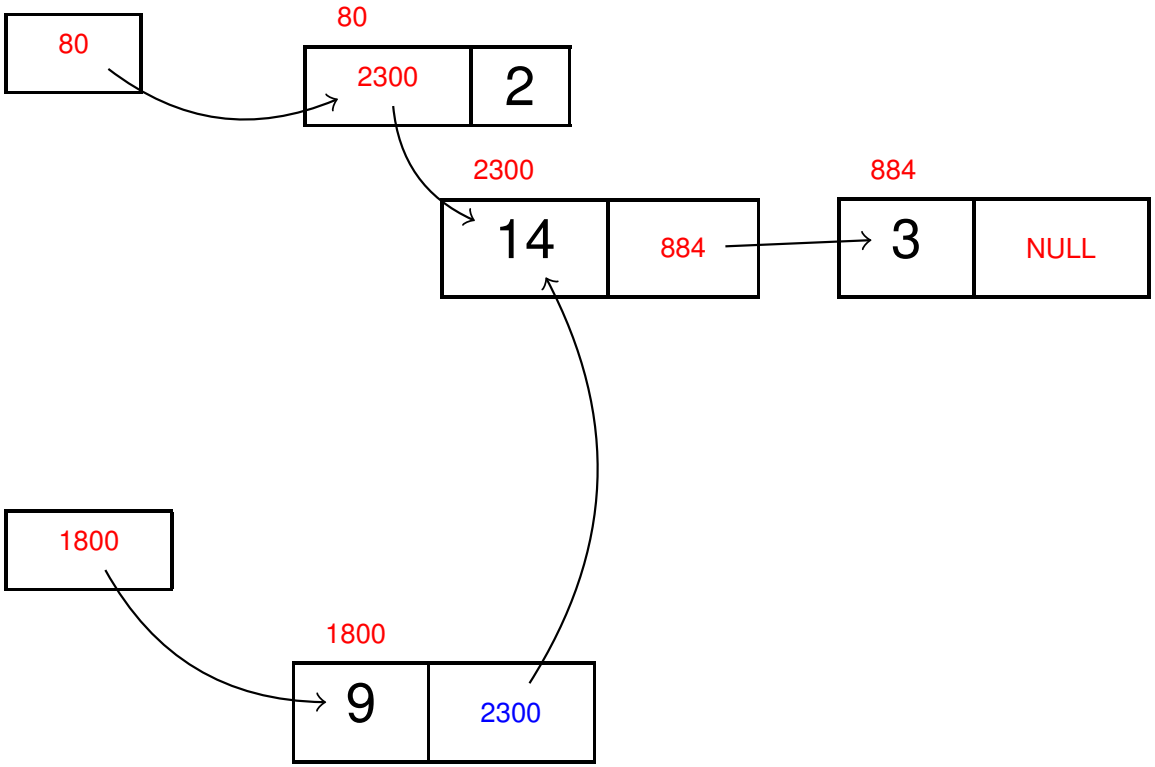
On prend donc en entrée :

1. un pointeur sur un maillon.
2. un pointeur sur une liste.

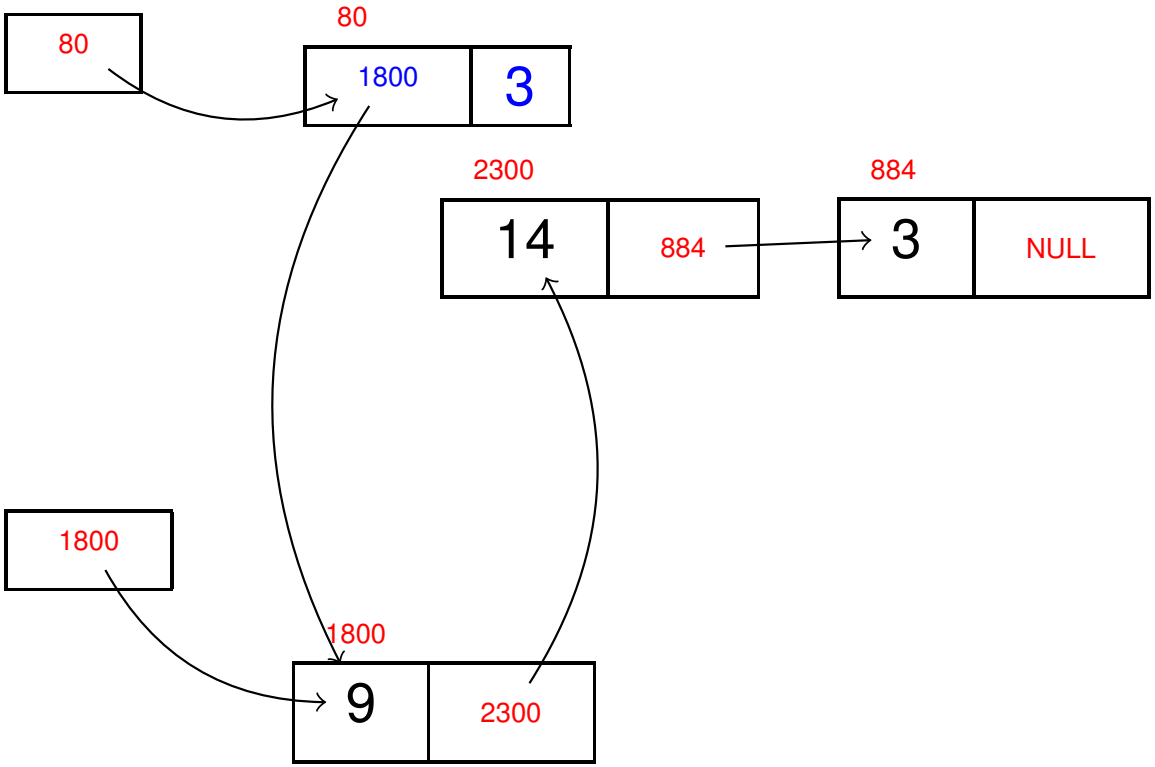
Les listes chaînées : ajouter un maillon



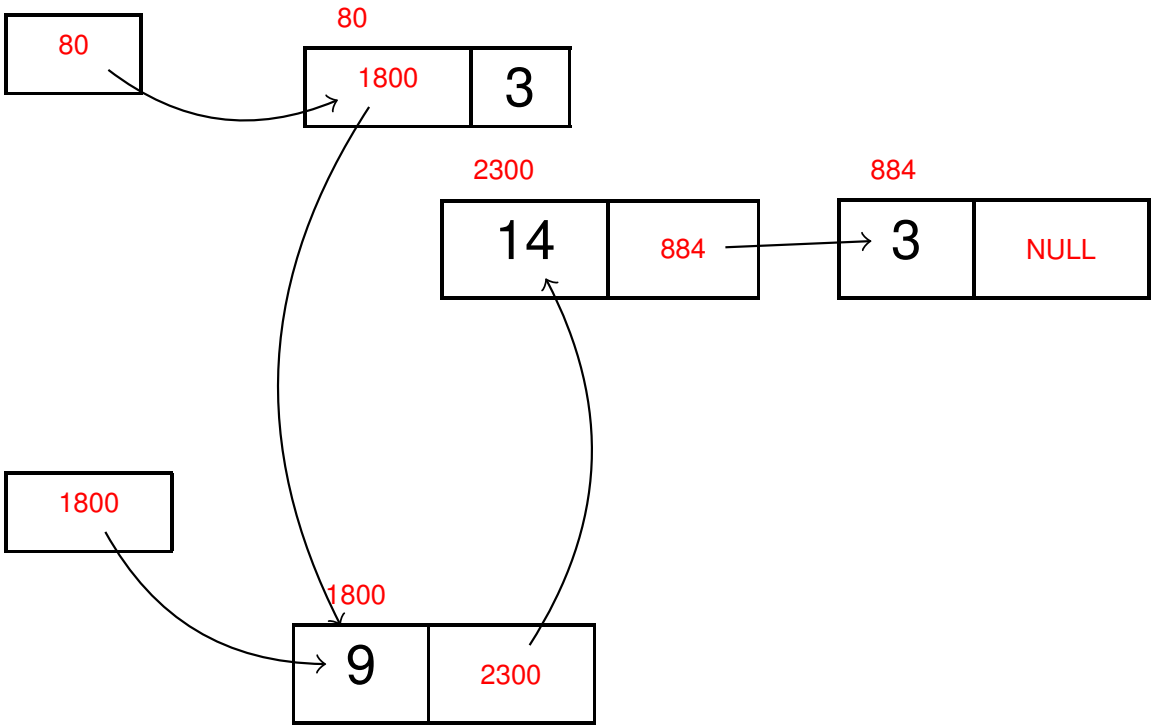
Les listes chaînées : ajouter un maillon



Les listes chaînées : ajouter un maillon



Les listes chaînées : ajouter un maillon



Compliqué ?

Les listes chaînées : ajouter un élément

```
1 void liste_ajout(liste_t * l, maillon_t * m){
2   m->suivant=l->debut;
3   l->debut=m;
4   l->taille++;
5 }
```

Complexité

- Temps : $\Theta(1)$
- Espace : $\Theta(1)$

Les listes chaînées : extraire un élément

```
1 maillon_t * liste_extraire_debut(liste_t * l){
2   maillon_t * res=l->debut:
3   l->debut=res->suiv;
4   l->taille --;
5   res->suiv=NULL;
6   return res;
7 }
```

Complexité

- Temps : $\Theta(1)$
- Espace : $\Theta(1)$

Les listes chaînées : Détruire

Pour détruire une liste, on la vide maillon par maillon.

```
1 void liste_detruire(liste_t * l){
2   while( !liste_vide(*l) )
3     maillon_detruire( liste_extraire_debut(l) );
4   free(l);
5 }
```

Complexité

- Temps : $\Theta(n)$
- Espace : $\Theta(1)$

Récapitulatif

Fonctions	Tableau non-trié	Liste chaînée
Initialisation	$\Theta(1)$	$\Theta(1)$
Test vide	$\Theta(1)$	$\Theta(1)$
Cardinalité	$\Theta(1)$	$\Theta(1)$
Affichage	$\Theta(n)$	$\Theta(n)$
Test appartient	$\Omega(1)$ et $\mathcal{O}(n)$	$\Omega(1)$ et $\mathcal{O}(n)$
Ajout	$\Omega(1)$ et $\mathcal{O}(n)$	$\Theta(1)$
Extraire dernier élément	$\Theta(1)$	$\Theta(1)$
Accès i^{eme} élément	$\Theta(1)$	$\Theta(i)$
Destruction	$\Theta(1)$	$\Theta(n)$

TABLE: Complexité en temps

Structure de données linéaire. . .

ISDL

Dans quelle structure de données linéaire est on limité à :

- ajouter un élément au début
- extraire un élément au début

? ? ? ? ? ?