Tableaux redimensionnables; Listes chaînées

Tableaux redimensionnables; Listes chaînées

Credits

- Openclassroom ici

Credits

- Openclassroom ici
- Wikipedia là

Objectif

- La taille d'un tableau C ou OCaml doit être connue à la déclaration.

Objectif

- La taille d'un tableau C ou OCaml doit être connue à la déclaration.
- Il n'est pourtant pas toujours aisé de déterminer la taille dont on a besoin à l'avance.

Objectif

- La taille d'un tableau C ou OCaml doit être connue à la déclaration.
- Il n'est pourtant pas toujours aisé de déterminer la taille dont on a besoin à l'avance.
- On se donne donc la possibilité de d'augmenter la capacité du tableau si besoin.

Interface

Dans un fichier **vector.h** :

```
/*Structure de tableau redimensionnable d'entiers*/
typedef struct Vector vector;
vector * vector_create();// créer un vecteur vide
int vector_size(vector *v);// O(1) espéré
int vector_get(vector *v, int i);// O(1)
void vector_set(vector *v, int i, int x);
void vector_resize(vector *v, int s);
void vector_delete(vector* v);
```

- La fonction `vector_create` est appelée un *constructeur* ; `vector_get` un *accesseur* et `vector_set` un *mutateur* (ou encore *transformateur*).

Interface

Dans un fichier **vector.h** :

```
/*Structure de tableau redimensionnable d'entiers*/
typedef struct Vector vector;
vector * vector_create();// créer un vecteur vide
int vector_size(vector *v);// O(1) espéré
int vector_get(vector *v, int i);// O(1)
void vector_set(vector *v, int i, int x);
void vector_resize(vector *v, int s);
void vector_delete(vector* v);
```

- La fonction `vector_create` est appelée un *constructeur* ; `vector_get` un *accesseur* et `vector_set` un *mutateur* (ou encore *transformateur*).
- On ne précise pas l'implémentation concrète de la structure `vector` mais les opérations qu'on peut effectuer dessus. Cette structure de données est dite *abstraite*.

Implémentation concrète

L'implémentation concrète de la structure précédente repose (par exemple) sur l'utilisation d'une structure dont un champ interne est un tableau d'entier. Mais ceci est complètement transparent pour l'utilisateur !

Implémentation

Dans un fichier **vector.c** écrivons :

```
1  typedef struct Vector{
2    int capacity;
3    int * data;// tableau de taille capacity
4    int size; // invariant 0 <=size <= capacity
5  } vector;
6
```

- En interne : tableau d'entiers de taille `capacity`
- `size` : nombre d'éléments du tableau redimensionnable.

Création

Comme d'habitude on préfère renvoyer des pointeurs sur structure :

```
vector * vector_create(){
  vector *v = malloc (sizeof(vector));
  v->capacity = 0;
  v->data = NULL;
  v->size=0;// il faudra redimensionner avant toute action
  return v;
}
```

`v->size` désigne la première case libre du tableau.

Taille, consultation, modification

```
int vector_size(vector *v){
  return v->size;
}

int vector_get(vector *v, int i){
  // renvoyer le contenu de la case i
  assert(0<=i && i <v->size);
  return v->data[i];
}

// Fonction outil à ne pas utiliser directement
// A utiliser avec précaution car aucun
// contrôle de la capacité du tableau
void vector_set(vector *v, int i, int x){
  // ajouter x position i
  v->data[i]=x;
}
```

Redimensionnement

```
void vector_resize(vector *v, int c){
  assert(0<=c);
  if (c>v->capacity){
    v->capacity = 2 * v->capacity;
    if (v->capacity < c)
       v->capacity = c ;
    // la capacité est donc au moins multipliée par 2
    int* old = v->data;
    v->data=malloc(v->capacity * sizeof(int));// O(1) supposé
    for (int i=0; i < v->size; i++)
       v->data[i] = old[i];
    if (old != NULL) free(old);
  }
}
```

Redimensionnement

- Si `c > v->capacity`, on redimensionne en `max(2*v->capacity,c)`.

Redimensionnement

- Si `c > v->capacity`, on redimensionne en `max(2*v->capacity,c)`.
- On réalloue un nouveau tableau interne et on copie les valeurs de l'ancien dedans.

Redimensionnement

- Si `c > v->capacity`, on redimensionne en `max(2*v->capacity,c)`.
- On réalloue un nouveau tableau interne et on copie les valeurs de l'ancien dedans.
- On désalloue l'ancien tableau.

Accumulation

La fonction `vector_push` ajoute un élément et incrémente la taille du tableau.

```
1  void vector_push(vector *v, int x){
2    int n = v->size;
3    vector_resize(v,n+1);// O(1) ou O(2 * v->capacity)
4    vector_set(v,n,x);// O(1)
5    v->size = n+1;
6  }
7
```

- Si on n'utilise que cette fonction pour ajouter des éléments, la taille du tableau est bien en adéquation avec le nombre d'éléments qui ont été effectivement ajoutés.

Accumulation

La fonction `vector_push` ajoute un élément et incrémente la taille du tableau.

```
1  void vector_push(vector *v, int x){
2    int n = v->size;
3    vector_resize(v,n+1);// O(1) ou O(2 * v->capacity)
4    vector_set(v,n,x);// O(1)
5    v->size = n+1;
6  }
7
```

- Si on n'utilise que cette fonction pour ajouter des éléments, la taille du tableau est bien en adéquation avec le nombre d'éléments qui ont été effectivement ajoutés.
- Avec `push` et le test `(v->size == 0)` on a presque tout ce qu'il faut pour une structure de pile (cf chap. suivant). MANQUE `pop` !!

Complexité amortie d'une séquence d'opérations

- On parle de complexité *amortie* d'une opération lorsque le coup d'une séquence de n operations est linéaire en n. En moyenne, on considère que CHAQUE opération a un coût $O(1)$ alors même que certaines opérations ne sont pas de coût constant.

Complexité amortie d'une séquence d'opérations

- On parle de complexité *amortie* d'une opération lorsque le coup d'une séquence de n operations est linéaire en n. En moyenne, on considère que CHAQUE opération a un coût $O(1)$ alors même que certaines opérations ne sont pas de coût constant.
- On considère une séquence de n opérations `vector_push` à partir d'une création avec `vector_create()` et on montre que la complexité de la séquence est linéaire en n.

Complexité amortie d'une séquence d'opérations

- On parle de complexité *amortie* d'une opération lorsque le coup d'une séquence de n operations est linéaire en n. En moyenne, on considère que CHAQUE opération a un coût $O(1)$ alors même que certaines opérations ne sont pas de coût constant.
- On considère une séquence de n opérations `vector_push` à partir d'une création avec `vector_create()` et on montre que la complexité de la séquence est linéaire en n.
- On pourra donc en déduire que la complexité amortie d'une opération `vector_push` est en $O(1)$.

Complexité amortie d'une séquence d'opérations

Méthode directe

- On considère une séquence de n opérations `vector_push` à partir d'une création avec `vector_create()` et on montre que la complexité de la séquence est linéaire en n.

Complexité amortie d'une séquence d'opérations

Méthode directe

- On considère une séquence de n opérations `vector_push` à partir d'une création avec `vector_create()` et on montre que la complexité de la séquence est linéaire en n.
- On pourra donc en déduire que la complexité amortie d'une opération `vector_push` est en $O(1)$.

Complexité amortie d'une séquence d'opérations

Méthode directe

- On considère une séquence de n opérations `vector_push` à partir d'une création avec `vector_create()` et on montre que la complexité de la séquence est linéaire en n.
- On pourra donc en déduire que la complexité amortie d'une opération `vector_push` est en $O(1)$.
- Méthode directe :

Complexité amortie d'une séquence d'opérations

Méthode directe

- On considère une séquence de n opérations `vector_push` à partir d'une création avec `vector_create()` et on montre que la complexité de la séquence est linéaire en n.
- On pourra donc en déduire que la complexité amortie d'une opération `vector_push` est en $O(1)$.
- Méthode directe :
  - Pour ajouter n éléments successivement à partir d'un tableau de taille initiale 1, on fait des redimensionnements avec copies pour des coûts respectifs de l'ordre de $1, 2, 4, \ldots, 2^k$ où $k = \lfloor \log_2(n) \rfloor$

Complexité amortie d'une séquence d'opérations

Méthode directe

- On considère une séquence de n opérations `vector_push` à partir d'une création avec `vector_create()` et on montre que la complexité de la séquence est linéaire en n.
- On pourra donc en déduire que la complexité amortie d'une opération `vector_push` est en $O(1)$.
- Méthode directe :
  - Pour ajouter n éléments successivement à partir d'un tableau de taille initiale 1, on fait des redimensionnements avec copies pour des coûts respectifs de l'ordre de $1, 2, 4, \ldots, 2^k$ où $k = \lfloor \log_2(n) \rfloor$
  - A ces redimensionnements s'ajoute un coût constant $O(1)$ pour chaque opération `vector_push` (comparaisons + écritures). On obtient donc une complexité encadrée par des multiples de :

$$n \times 1 + \sum_{i=0}^{k} 2^i = n + 2^{k+1} - 1 \simeq n + 2n - 1 = \Theta(n)$$

Complexité amortie d'une séquence d'opérations

Méthode directe

- On considère une séquence de n opérations `vector_push` à partir d'une création avec `vector_create()` et on montre que la complexité de la séquence est linéaire en n.
- On pourra donc en déduire que la complexité amortie d'une opération `vector_push` est en $O(1)$.
- Méthode directe :
  - Pour ajouter n éléments successivement à partir d'un tableau de taille initiale 1, on fait des redimensionnements avec copies pour des coûts respectifs de l'ordre de $1, 2, 4, \ldots, 2^k$ où $k = \lfloor \log_2(n) \rfloor$
  - A ces redimensionnements s'ajoute un coût constant $O(1)$ pour chaque opération `vector_push` (comparaisons + écritures). On obtient donc une complexité encadrée par des multiples de :

$$n \times 1 + \sum_{i=0}^{k} 2^i = n + 2^{k+1} - 1 \simeq n + 2n - 1 = \Theta(n)$$

Complexité d'une séquence d'opérations

Méthode du potentiel

Dans ce qui suit on compte le nombre d'accès (lecture/écriture) à des données de tableaux (`v->data` ou `old`).

- Pour alléger les notations, on note c pour `v->capacity` et s pour `v->size` et v le tableau qui subit une opération.

Complexité d'une séquence d'opérations

Méthode du potentiel

Dans ce qui suit on compte le nombre d'accès (lecture/écriture) à des données de tableaux (`v->data` ou `old`).

- Pour alléger les notations, on note c pour `v->capacity` et s pour `v->size` et v le tableau qui subit une opération.
- On peut poser comme potentiel :

$$\phi(v) \underset{\text{def}}{=} \max(0, 4s - 2c)$$

Comme s grossit de 1 en 1 à chaque opération `vector_push`, le redimensionnement n'arrive que si $s \simeq c$, et, dans ce cas il y a en gros $2s$ opérations d'accès (lecture/écriture).

Complexité d'une séquence d'opérations

Méthode du potentiel

Dans ce qui suit on compte le nombre d'accès (lecture/écriture) à des données de tableaux (`v->data` ou `old`).

- Pour alléger les notations, on note c pour `v->capacity` et s pour `v->size` et v le tableau qui subit une opération.
- On peut poser comme potentiel :

$$\phi(v) \underset{\text{def}}{=} \max(0, 4s - 2c)$$

Comme s grossit de 1 en 1 à chaque opération `vector_push`, le redimensionnement n'arrive que si $s \simeq c$, et, dans ce cas il y a en gros $2s$ opérations d'accès (lecture/écriture).

- Le coût réel pour chaque opération `push` est de 1 si pas de redimensionnement (1 écriture dans `v->data`) et de $2s + 1$ sinon (s copies de `old` vers `v->data` + 1 écriture dans `v->data`).

Complexité d'une séquence d'opérations

Méthode du potentiel

- **Si v n'est pas redimensionné**, le coût réel est 1 et donc le coût amorti est $a = 1 + \phi(\text{après}) - \phi(\text{avant})$

Complexité d'une séquence d'opérations

Méthode du potentiel

- **Si v n'est pas redimensionné**, le coût réel est 1 et donc le coût amorti est $a = 1 + \phi(\text{après}) - \phi(\text{avant})$
  - si $s + 1 \leq \frac{c}{2}$, alors les potentiels avant et après sont nuls. Donc $a = 1$.

Complexité d'une séquence d'opérations

Méthode du potentiel

- **Si v n'est pas redimensionné**, le coût réel est 1 et donc le coût amorti est $a = 1 + \phi(\text{après}) - \phi(\text{avant})$
  - si $s + 1 \leq \frac{c}{2}$, alors les potentiels avant et après sont nuls. Donc $a = 1$.
  - Si $s + 1 > \frac{c}{2}$, alors $s \geq \frac{c}{2}$ donc $4s \geq 2c$. Alors :

$$a = 1 + (4(s + 1) - 2c) - (4s - 2c) = 5$$

Complexité d'une séquence d'opérations

Méthode du potentiel

- **Si v n'est pas redimensionné**, le coût réel est 1 et donc le coût amorti est $a = 1 + \phi(\text{après}) - \phi(\text{avant})$
  - si $s + 1 \leq \frac{c}{2}$, alors les potentiels avant et après sont nuls. Donc $a = 1$.
  - Si $s + 1 > \frac{c}{2}$, alors $s \geq \frac{c}{2}$ donc $4s \geq 2c$. Alors :

$$a = 1 + (4(s+1) - 2c) - (4s - 2c) = 5$$

- **Si v est redimensionné**, coût réel : $1 + 2s$. Alors $c = s$ au coup d'avant et la nouvelle capacité est $c' = 2s$. Le coût amorti est encore 5. En effet :

$$a = 1+2s+(4(s+1)-2c')-(4s-2c) = 1+2s+(4(s+1)-4s)-(4s-2s)$$

Complexité d'une séquence d'opérations

Méthode du potentiel

- **Si v n'est pas redimensionné**, le coût réel est 1 et donc le coût amorti est $a = 1 + \phi(\text{après}) - \phi(\text{avant})$
  - si $s + 1 \leq \frac{c}{2}$, alors les potentiels avant et après sont nuls. Donc $a = 1$.
  - Si $s + 1 > \frac{c}{2}$, alors $s \geq \frac{c}{2}$ donc $4s \geq 2c$. Alors :

$$a = 1 + (4(s+1) - 2c) - (4s - 2c) = 5$$

- **Si v est redimensionné**, coût réel : $1 + 2s$. Alors $c = s$ au coup d'avant et la nouvelle capacité est $c' = 2s$. Le coût amorti est encore 5. En effet :

$$a = 1+2s+(4(s+1)-2c')-(4s-2c) = 1+2s+(4(s+1)-4s)-(4s-2s)$$

- Le coût amorti de la séquence est donc majoré par une constante. Par le corollaire du théorème d'amortissement, chaque opération de la séquence a une complexité amortie en $O(1)$.

Présentation

- *Liste chaînée* : collection ordonnée et de taille arbitraire d'éléments de même type.

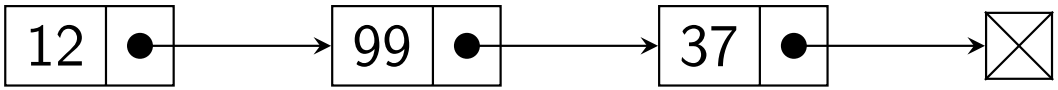
Présentation

- *Liste chaînée* : collection ordonnée et de taille arbitraire d'éléments de même type.
- Représentation en mémoire : succession de *cellules* faites d'un contenu et d'un pointeur vers une autre cellule.

Présentation

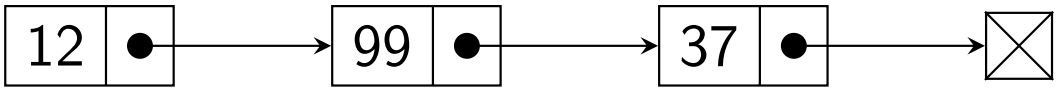
- *Liste chaînée* : collection ordonnée et de taille arbitraire d'éléments de même type.
- Représentation en mémoire : succession de *cellules* faites d'un contenu et d'un pointeur vers une autre cellule.
- Image : la liste chaînée peut être représentée par une vchaîne dont les maillons sont les cellules.

Liste simplement chaîné



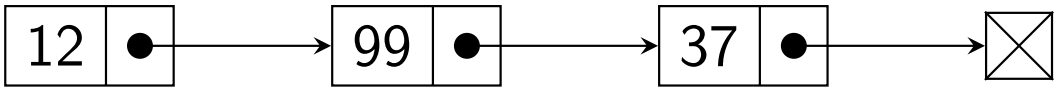
- Deux informations composent chaque élément de la liste chaînée :

Liste simplement chaîné



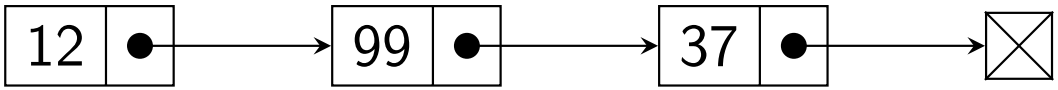
- Deux informations composent chaque élément de la liste chaînée :
  - la *valeur* associée à l'élément,

Liste simplement chaîné



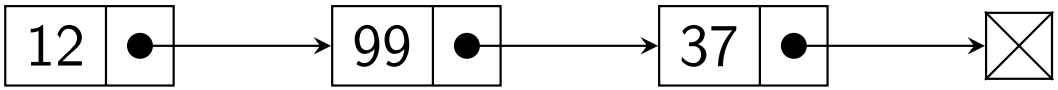
- Deux informations composent chaque élément de la liste chaînée :
  - la *valeur* associée à l'élément,
  - un *pointeur* vers l'élément suivant (ou successeur).

Liste simplement chaîné



- Deux informations composent chaque élément de la liste chaînée :
  - la *valeur* associée à l'élément,
  - un *pointeur* vers l'élément suivant (ou successeur).
- Comme un seul élément de la liste est pointé, l'accès se fait uniquement dans un sens. La fin de la liste est marquée par une valeur sentinelle, ici le pointeur NULL. L'usage d'un nœud sentinelle est aussi possible, notamment pour les listes cycliques.

Liste simplement chaîné



- Deux informations composent chaque élément de la liste chaînée :
  - la *valeur* associée à l'élément,
  - un *pointeur* vers l'élément suivant (ou successeur).
- Comme un seul élément de la liste est pointé, l'accès se fait uniquement dans un sens. La fin de la liste est marquée par une valeur sentinelle, ici le pointeur NULL. L'usage d'un nœud sentinelle est aussi possible, notamment pour les listes cycliques.
- Le premier élément de la liste a la valeur 12, le dernier a la valeur 37 et son pointeur est NULL.

Primitives

Les primitives sur les listes chaînées n'ont pas un nom aussi codifié que celles sur les piles. Citons en quelques unes :

- « Placement sur le premier élément » : place l'index sur le premier élément de la liste.

Primitives

Les primitives sur les listes chaînées n'ont pas un nom aussi codifié que celles sur les piles. Citons en quelques unes :

- « Placement sur le premier élément » : place l'index sur le premier élément de la liste.
- « Placement sur le dernier élément » : place l'index sur le dernier élément de la liste.

Primitives

Les primitives sur les listes chaînées n'ont pas un nom aussi codifié que celles sur les piles. Citons en quelques unes :

- « Placement sur le premier élément » : place l'index sur le premier élément de la liste.
- « Placement sur le dernier élément » : place l'index sur le dernier élément de la liste.
- « Placement sur l'élément suivant » : place l'index sur l'élément qui suit l'élément courant si c'est possible.

Primitives

Les primitives sur les listes chaînées n'ont pas un nom aussi codifié que celles sur les piles. Citons en quelques unes :

- « Placement sur le premier élément » : place l'index sur le premier élément de la liste.
- « Placement sur le dernier élément » : place l'index sur le dernier élément de la liste.
- « Placement sur l'élément suivant » : place l'index sur l'élément qui suit l'élément courant si c'est possible.
- « Placement sur l'élément précédent » : place l'index sur l'élément qui précède l'élément courant si c'est possible.

Primitives

Les primitives sur les listes chaînées n'ont pas un nom aussi codifié que celles sur les piles. Citons en quelques unes :

- « Placement sur le premier élément » : place l'index sur le premier élément de la liste.
- « Placement sur le dernier élément » : place l'index sur le dernier élément de la liste.
- « Placement sur l'élément suivant » : place l'index sur l'élément qui suit l'élément courant si c'est possible.
- « Placement sur l'élément précédent » : place l'index sur l'élément qui précède l'élément courant si c'est possible.
- « Liste est-elle vide ? » : Retourne vrai si la liste est vide, faux sinon.

Primitives

Les primitives sur les listes chaînées n'ont pas un nom aussi codifié que celles sur les piles. Citons en quelques unes :

- « Placement sur le premier élément » : place l'index sur le premier élément de la liste.
- « Placement sur le dernier élément » : place l'index sur le dernier élément de la liste.
- « Placement sur l'élément suivant » : place l'index sur l'élément qui suit l'élément courant si c'est possible.
- « Placement sur l'élément précédent » : place l'index sur l'élément qui précède l'élément courant si c'est possible.
- « Liste est-elle vide ? » : Retourne vrai si la liste est vide, faux sinon.
- « L'élément courant est-il le premier ? » : Retourne vrai si l'élément courant est le premier élément de la liste, faux sinon.

Primitives (suite)

- « L'élément courant est-il le dernier ? » : Retourne vrai si l'élément courant est le dernier élément de la liste, faux sinon.

Primitives (suite)

- « L'élément courant est-il le dernier ? » : Retourne vrai si l'élément courant est le dernier élément de la liste, faux sinon.
- « Nombre d'éléments » : renvoie le nombre d'éléments dans la liste.

Primitives (suite)

- « L'élément courant est-il le dernier ? » : Retourne vrai si l'élément courant est le dernier élément de la liste, faux sinon.
- « Nombre d'éléments » : renvoie le nombre d'éléments dans la liste.
- « Ajouter en queue » : ajoute un élément après le dernier élément de la liste (efficace seulement pour une liste doublement chaînée).

Primitives (suite)

- « L'élément courant est-il le dernier ? » : Retourne vrai si l'élément courant est le dernier élément de la liste, faux sinon.
- « Nombre d'éléments » : renvoie le nombre d'éléments dans la liste.
- « Ajouter en queue » : ajoute un élément après le dernier élément de la liste (efficace seulement pour une liste doublement chaînée).
- « Ajouter en tête » : ajoute un élément avant le premier élément de la liste.

Primitives (suite)

- « L'élément courant est-il le dernier ? » : Retourne vrai si l'élément courant est le dernier élément de la liste, faux sinon.
- « Nombre d'éléments » : renvoie le nombre d'éléments dans la liste.
- « Ajouter en queue » : ajoute un élément après le dernier élément de la liste (efficace seulement pour une liste doublement chaînée).
- « Ajouter en tête » : ajoute un élément avant le premier élément de la liste.
- « Insertion » : insère un élément avant l'élément courant.

Primitives (suite)

- « L'élément courant est-il le dernier ? » : Retourne vrai si l'élément courant est le dernier élément de la liste, faux sinon.
- « Nombre d'éléments » : renvoie le nombre d'éléments dans la liste.
- « Ajouter en queue » : ajoute un élément après le dernier élément de la liste (efficace seulement pour une liste doublement chaînée).
- « Ajouter en tête » : ajoute un élément avant le premier élément de la liste.
- « Insertion » : insère un élément avant l'élément courant.
- « Remplacement » : Remplace l'élément courant.

Primitives (suite)

- « L'élément courant est-il le dernier ? » : Retourne vrai si l'élément courant est le dernier élément de la liste, faux sinon.
- « Nombre d'éléments » : renvoie le nombre d'éléments dans la liste.
- « Ajouter en queue » : ajoute un élément après le dernier élément de la liste (efficace seulement pour une liste doublement chaînée).
- « Ajouter en tête » : ajoute un élément avant le premier élément de la liste.
- « Insertion » : insère un élément avant l'élément courant.
- « Remplacement » : Remplace l'élément courant.
- « Suppression » : Supprime l'élément courant.

Primitives (suite)

- « L'élément courant est-il le dernier ? » : Retourne vrai si l'élément courant est le dernier élément de la liste, faux sinon.
- « Nombre d'éléments » : renvoie le nombre d'éléments dans la liste.
- « Ajouter en queue » : ajoute un élément après le dernier élément de la liste (efficace seulement pour une liste doublement chaînée).
- « Ajouter en tête » : ajoute un élément avant le premier élément de la liste.
- « Insertion » : insère un élément avant l'élément courant.
- « Remplacement » : Remplace l'élément courant.
- « Suppression » : Supprime l'élément courant.

Nous en implanterons quelques unes en TP.

Implémentation par tableau

- On peut implémenter cette structure par un tableau t et un indice i indiquant la dernière case significative du tableau.

Implémentation par tableau

- On peut implémenter cette structure par un tableau t et un indice i indiquant la dernière case significative du tableau.
- Les cellules sont alors contiguës et on accède rapidement au k-ème élément. $O(1)$

Implémentation par tableau

- On peut implémenter cette structure par un tableau t et un indice i indiquant la dernière case significative du tableau.
- Les cellules sont alors contiguës et on accède rapidement au k-ème élément. $O(1)$
- Pour ajouter un élément : si $i < |t| - 1$, on incrémente l'indice i et on enregistre le nouvel elément en position i. $O(1)$

Implémentation par tableau

- On peut implémenter cette structure par un tableau t et un indice i indiquant la dernière case significative du tableau.
- Les cellules sont alors contiguës et on accède rapidement au k-ème élément. $O(1)$
- Pour ajouter un élément : si $i < |t| - 1$, on incrémente l'indice i et on enregistre le nouvel elément en position i. $O(1)$
- Pour supprimer le dernier élément, si $i > 0$, on décrémente simplement la taille. $O(1)$

Implémentation par tableau

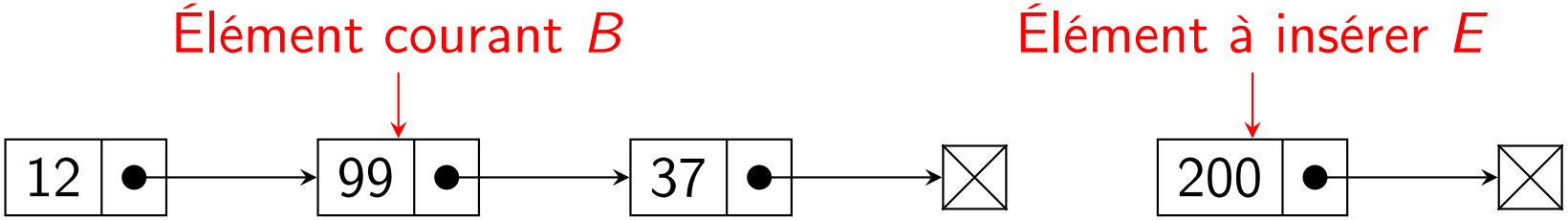
- On peut implémenter cette structure par un tableau t et un indice i indiquant la dernière case significative du tableau.
- Les cellules sont alors contiguës et on accède rapidement au k-ème élément. $O(1)$
- Pour ajouter un élément : si $i < |t| - 1$, on incrémente l'indice i et on enregistre le nouvel elément en position i. $O(1)$
- Pour supprimer le dernier élément, si $i > 0$, on décrémente simplement la taille. $O(1)$
- Problèmes : que faire si on veut ajouter un élément alors que $i = |t| - 1$?

Implémentation par tableau

- On peut implémenter cette structure par un tableau t et un indice i indiquant la dernière case significative du tableau.
- Les cellules sont alors contiguës et on accède rapidement au k-ème élément. $O(1)$
- Pour ajouter un élément : si $i < |t| - 1$, on incrémente l'indice i et on enregistre le nouvel elément en position i. $O(1)$
- Pour supprimer le dernier élément, si $i > 0$, on décrémente simplement la taille. $O(1)$
- Problèmes : que faire si on veut ajouter un élément alors que $i = |t| - 1$?
  - si $i = |t| - 1$ et si on veut ajouter un élément, on doit faire une réallocation qui va se traduire par la recherche d'une zone mémoire avec suffisament de cases contiguës et une copie de t dans la nouvelle zone. En $O(|t|)$.
    Voir la section 1 sur les tableaux redimensionnables.

Implémentation par chaînage (Ajout d'un élément)

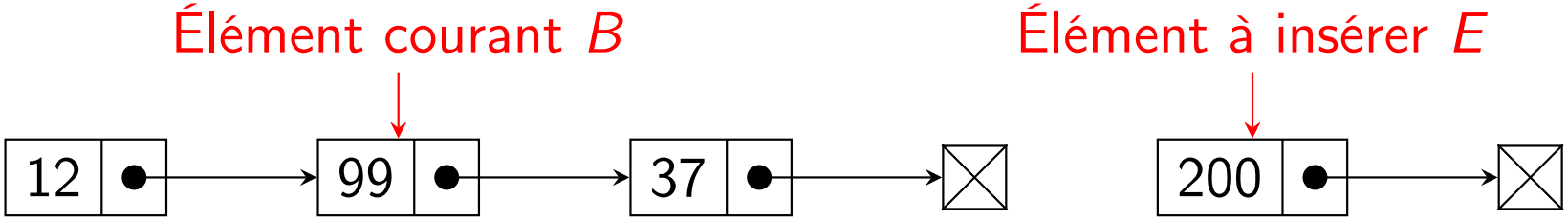
- Situation initiale :



On veut insérer un nouvel élément élément E de valeur 200 après l'élément B de valeur 99 (et donc avant l'élément C de valeur 37).

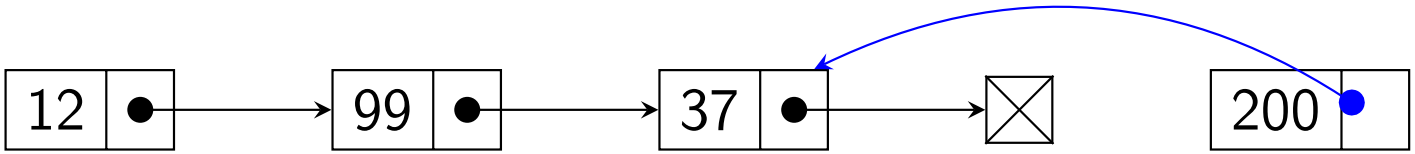
Implémentation par chaînage (Ajout d'un élément)

- Situation initiale :



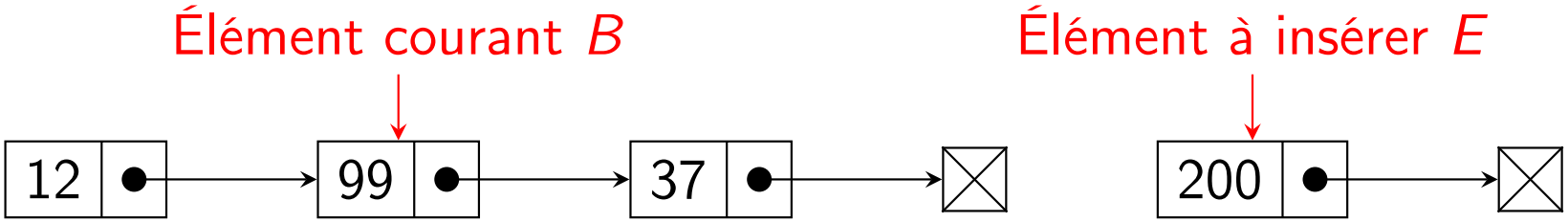
On veut insérer un nouvel élément élément E de valeur 200 après l'élément B de valeur 99 (et donc avant l'élément C de valeur 37).

- On change le pointeur (initialement NULL) de E : E pointe maintenant vers l'élément C.



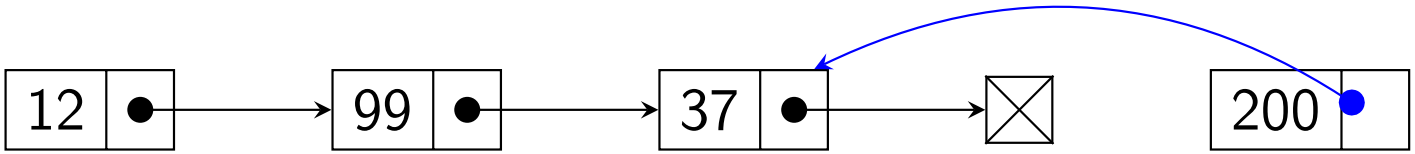
Implémentation par chaînage (Ajout d'un élément)

- Situation initiale :

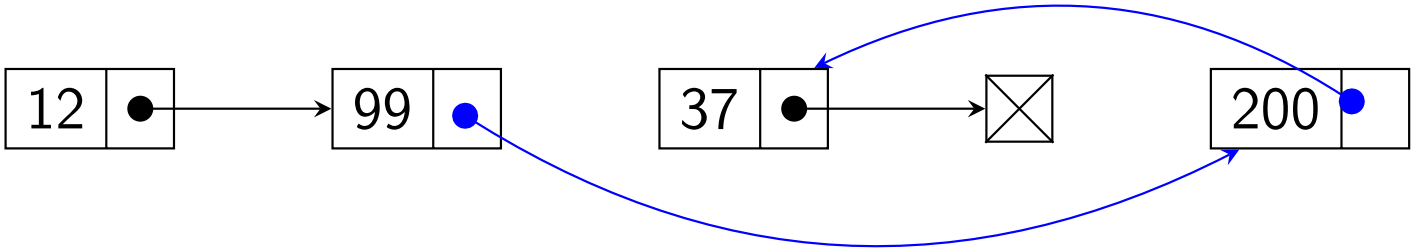


On veut insérer un nouvel élément élément E de valeur 200 après l'élément B de valeur 99 (et donc avant l'élément C de valeur 37).

- On change le pointeur (initialement NULL) de E : E pointe maintenant vers l'élément C.

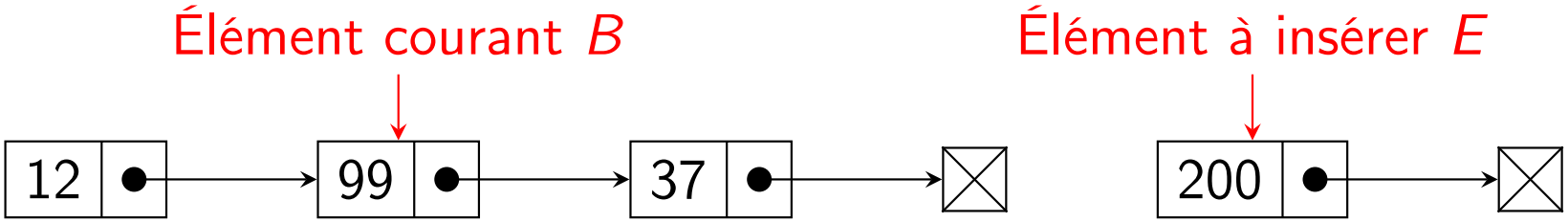


- On suprime le lien de B vers C et on fait pointer B vers E.



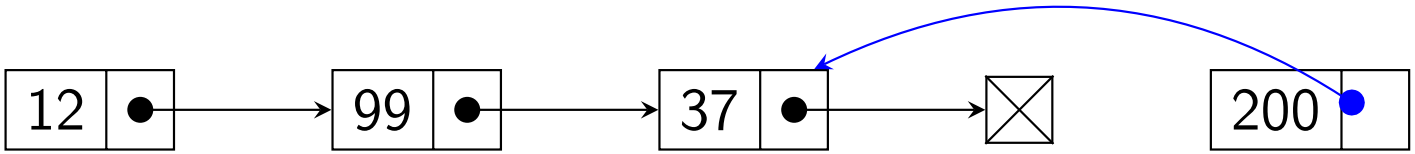
Implémentation par chaînage (Ajout d'un élément)

- Situation initiale :

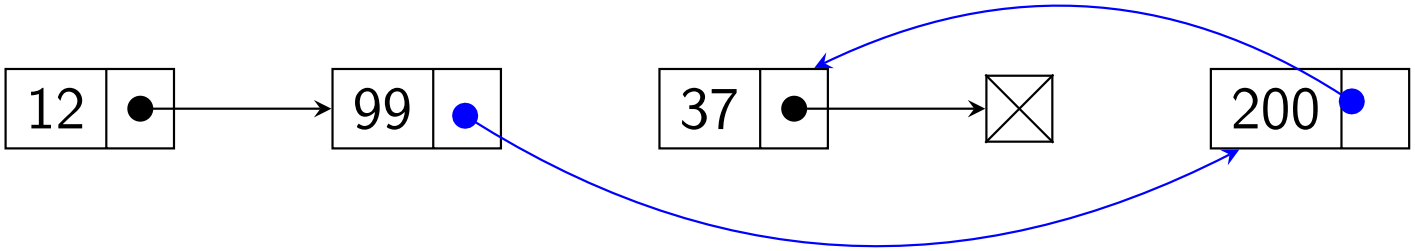


On veut insérer un nouvel élément élément E de valeur 200 après l'élément B de valeur 99 (et donc avant l'élément C de valeur 37).

- On change le pointeur (initialement NULL) de E : E pointe maintenant vers l'élément C.



- On suprime le lien de B vers C et on fait pointer B vers E.



Un type C

- Type d'un élément de la liste :

```
1  // openClassRoom et Wikipedia :
2  typedef struct Element Element;
3  struct Element{
4      int val;
5      Element *next;
6  };
7
```

Un type C

- Type d'un élément de la liste :

```
1  // openClassRoom et Wikipedia :
2  typedef struct Element Element;
3  struct Element{
4      int val;
5      Element *next;
6  };
7
```

- Structure de contrôle. On donne le type d'une liste. C'est une structure qui contient juste un pointeur vers le premier `Element` de la liste.

```
1  typedef struct Liste Liste;
2  struct Liste{
3    Element *first;
4    int size;//facultatif
5  };
```

Un type **list**

- On peut utiliser

```
1 type 'a list = Empty | List of 'a * 'a list
2 (*Exemple : -> 1 -> 2-> 3*)
3 let l = List (1,List(2,List(3,Empty)))
```

Un type **list**

- On peut utiliser

```
1 type 'a list = Empty | List of 'a * 'a list
2 (*Exemple : -> 1 -> 2-> 3*)
3 let l = List (1,List(2,List(3,Empty)))
```

- Un type réalisant les mêmes opérations est prédéfini en OCaml : le type `list` dont les primitives sont implantées dans le module **List**.

Un type **list**

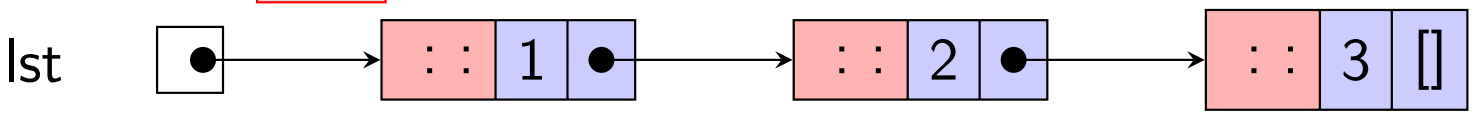
- On peut utiliser

```
1 type 'a list = Empty | List of 'a * 'a list
2 (*Exemple : -> 1 -> 2-> 3*)
3 let l = List (1,List(2,List(3,Empty)))
```

- Un type réalisant les mêmes opérations est prédéfini en OCaml : le type `list` dont les primitives sont implantées dans le module **List**.
- Comme en C, une valeur de type `list` est un pointeur vers un bloc mémoire de deux valeurs, l'élément et la suite de la liste.

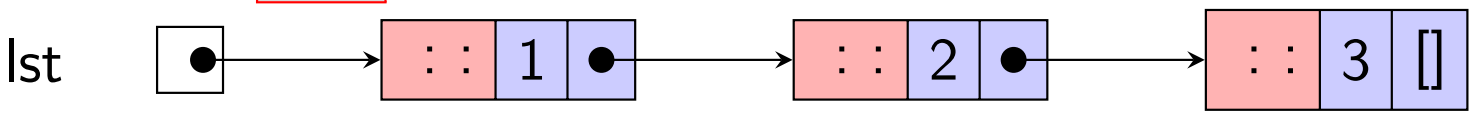
Un type **list**

- Par exemple, `let lst = 1::2::3::[]` alloue 3 blocs mémoires. Et la variable `lst` contient un pointeur vers la première cellule.



Un type **list**

- Par exemple, `let lst = 1::2::3::[]` alloue 3 blocs mémoires. Et la variable `lst` contient un pointeur vers la première cellule.



- On voit apparaître 3 parties par bloc mémoire contre 2 en C. Une partie de chaque bloc est en effet utilisée par OCaml pour stocker des méta-informations (notée ici : :) comme la nature du bloc et sa taille. Cette méta-information est utilisée en particulier par le Garbage Collector d'OCaml.

Listes en C vs OCaml

Persistance EnC, les listes sont mutables (on peut modifier `e->val` et `e->next`, tandis qu'en OCaml aucune partie de la liste n'est modifiable.

Listes en C vs OCaml

Persistance EnC, les listes sont mutables (on peut modifier `e->val` et `e->next`, tandis qu'en OCaml aucune partie de la liste n'est modifiable.

Polymorphisme En C les listes sont monomorphes (un seul type de données : celui indiqué dans la déclaration de structure). En OCaml, les listes sont polymorphes (il peut y avoir des listes de n'importe quoi).

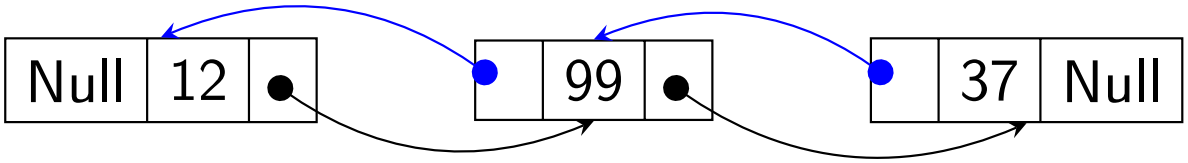
Listes en C vs OCaml

Persistance EnC, les listes sont mutables (on peut modifier `e->val` et `e->next`, tandis qu'en OCaml aucune partie de la liste n'est modifiable.

Polymorphisme En C les listes sont monomorphes (un seul type de données : celui indiqué dans la déclaration de structure). En OCaml, les listes sont polymorphes (il peut y avoir des listes de n'importe quoi).

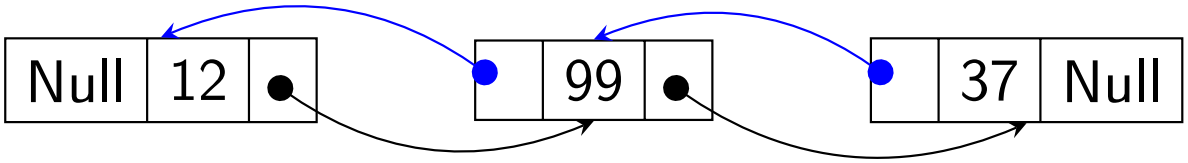
Homogéneité En OCaml comme en C, les listes sont homogènes (tous les éléments ont le même type que le premier). En Python les listes sont hétérogènes.

Listes doublement chaînées



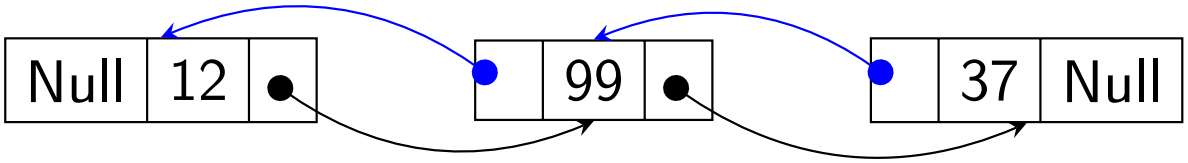
- Un pointeur vers l'élément précédent (ou prédécesseur) est ajouté. L'accès peut alors se faire indifféremment dans les deux sens : de successeur en successeur, ou de prédécesseur en prédécesseur.

Listes doublement chaînées



- Un pointeur vers l'élément précédent (ou prédécesseur) est ajouté. L'accès peut alors se faire indifféremment dans les deux sens : de successeur en successeur, ou de prédécesseur en prédécesseur.
- plus coûteuse en mémoire (un pointeur supplémentaire par élément) et en nombre d'instructions pour la mise à jour : une insertion coûte quatre copies de pointeurs, contre deux dans le cas d'une liste simplement chaînée.

Listes doublement chaînées



- Un pointeur vers l'élément précédent (ou prédécesseur) est ajouté. L'accès peut alors se faire indifféremment dans les deux sens : de successeur en successeur, ou de prédécesseur en prédécesseur.
- plus coûteuse en mémoire (un pointeur supplémentaire par élément) et en nombre d'instructions pour la mise à jour : une insertion coûte quatre copies de pointeurs, contre deux dans le cas d'une liste simplement chaînée.
- En revanche, à partir d'un élément courant, on peut insérer un nouvel élément avant ou après (pour une liste simplement chaînée, on peut insérer seulement après l'élément courant).

Cycles

- Une liste cyclique (ou circulaire) est créée lorsque le dernier élément possède une référence vers le premier élément (si la liste est doublement chaînée, alors le premier élément possède aussi une référence vers le dernier).

Cycles

- Une liste cyclique (ou circulaire) est créée lorsque le dernier élément possède une référence vers le premier élément (si la liste est doublement chaînée, alors le premier élément possède aussi une référence vers le dernier).
- L'utilisation de ce type de liste requiert des précautions pour éviter des parcours infinis, par exemple, lors d'une recherche vaine d'élément.

Cycles

- Une liste cyclique (ou circulaire) est créée lorsque le dernier élément possède une référence vers le premier élément (si la liste est doublement chaînée, alors le premier élément possède aussi une référence vers le dernier).
- L'utilisation de ce type de liste requiert des précautions pour éviter des parcours infinis, par exemple, lors d'une recherche vaine d'élément.
- Pour créer une liste circulaire à partir d'une liste simplemnent chaînée : faire pointer le pointeur du dernier élément (initialement Null) vers le premier élément.