TD XVI

TABLEAUX DYNAMIQUES

1 Types enregistrement en C : les `struct`

Introduction

On peut définir en C des types enregistrement :

```c
struct complexe {
  double re;
  double im;
}
```

On accède ensuite à un champ d'un complexe `z` par la notation `z.re` et `z.im`. Une fonction peut prendre une `struct` comme argument, par valeur, et aussi renvoyer une `struct` (deux choses qui ne sont pas possibles pour un tableau). Par exemple, la fonction suivante calcule le conjugué d'un nombre complexe, **et elle n'a pas d'effet secondaire** (puisque l'argument est passé par valeur) :

```c
struct complexe conjugue(struct complexe z){
  z.im = -z.im;
  return z;
}
```

Initialisation

Pour définir un objet de type `struct`, on peut :

- soit procéder en deux temps :

```c
struct complexe I;
I.re = 0.;
I.im = 1.;
```

- soit initialiser directement, avec cette syntaxe :

```c
struct complexe I = { .re = 0., .im = 1. };
```

Utilisation d'un `typedef`

Nous avons déjà vu qu'on pouvait utiliser le mot-clé `typedef` pour définir de nouveaux types (ou plutôt des alias de type) :

```c
// On définit entier comme un alias de int (intérêt douteux)
typedef entier int;

// On définit int_ptr comme un alias de int*, ce qui
// peut parfois rendre le code plus lisible.
typedef int_ptr int*;

// On définit rgb comme un tableau de trois int
typedef int rgb[3];
```

Il est également possible de définir un alias pour le type `struct complexe` :

```
typedef struct complexe complexe;
```

Remarque
On aurait pu choisir `toto` comme alias :

```
typedef struct complexe toto;
```

Cependant, le plus simple est de reprendre le nom utilisé pour la `struct` (autrement dit, de faire `typedef struct foo foo`).

On peut ensuite définir la fonction `conjugue`, par exemple, de manière plus concise :

```
complexe conjugue(complexe z){
  z.im = -z.im;
  return z;
}
```

Pointeur vers une `struct`

Il est très courant de manipuler des pointeurs vers des `struct` :

- soit parce que la `struct` est un peu grosse et qu'il est inefficace de la passer par valeur à une fonction (ce qui implique une copie) ;
- soit parce qu'on souhaite qu'une fonction puisse modifier la `struct` passée en argument (ce qui demande de passer en fait un pointeur vers cette `struct`).

En soi, cela ne pose aucun problème :

```
void conjugue_en_place(complexe* z){
  (*z).im = -(*z).im;
}
```

Cependant, les parenthèses autour de `*z` sont **obligatoires** ici : l'expression `*z.im` est lue comme `*(z.im)` (ce qui n'a aucun sens). Comme ces parenthèses deviennent très rapidement pénibles, on dispose d'une syntaxe spéciale :

> si `ps` est un pointeur vers une `struct`, alors `ps->champ` signifie `(*ps).champ`.

On peut donc écrire :

```
void conjuge_en_place(complexe* z){
  z->im = -z->im;
}
```

Attention à ne pas confondre les deux syntaxes : `s.champ` si `s` est une `struct`, `s->champ` si `s` est un pointeur vers une `struct` :

```
complexe z;
complexe* pz;
z.re = 3.;
pz->im = 1.;
```

2 Un tableau qui connaît sa taille

Comme nous l'avons vu, un tableau ne connaît pas sa taille [1], ce qui est très souvent problématique :

- il faut systématiquement penser à passer la taille comme paramètre supplémentaire aux fonctions, et bien sûr passer la bonne taille ;
- le compilateur ne peut pas ajouter de vérifications pour les accès hors-bornes.

Le deuxième point est particulièrement problématique : dans la plupart des langages, effectuer un accès hors-borne à un tableau résulte, de manière certaine, en une erreur bien définie à l'exécution. Si l'on veut aller chercher le dernier pourcent de performance, on peut souvent compiler avec une option demandant de désactiver cette vérification, mais c'est en général une très mauvaise idée. En C en revanche, un accès hors-borne donne un comportement non défini :

- si on a de la chance, cela résultera en une *segmentation fault* immédiate (quand la zone mémoire à laquelle on tente d'accéder n'appartient pas au processus) ;
- sinon, on va lire ou écrire dans une autre variable, et l'exécution va continuer avec un état du programme incohérent ;
- si on n'a vraiment pas de chance, on sera dans le deuxième cas, mais pas par hasard : un accès hors-borne est par nature une faille de sécurité, qui peut être exploitée.

Pour éviter ces problèmes, on peut tout simplement définir une `struct` qui contiendra la taille et un pointeur vers les données :

```c
struct int_array {
  int* data;
  int len;
};

typedef struct int_array int_array;
```

Pour créer un `int_array`, on peut utiliser la fonction suivante (que je vous invite à lire **attentivement**) :

```c
int_array* array_create(int len, int x){
    // On alloue le stockage pour la struct
    int_array* t = (int_array*)malloc(sizeof(int_array));
    // Et le stockage pour les données
    int* data = (int*)malloc(len * sizeof(int));
    for (int i = 0; i < len; i++){
        data[i] = x;
    }
    t->len = len;
    t->data = data;
    return t;
}
```

Assertions Pour tirer parti tirer du fait que nous connaissons la taille du tableau, il faut arrêter l'exécution du programme en cas d'accès hors-borne. Le plus simple est d'utiliser une assertion :

```c
#include <assert.h>

int main(void){
  ...
  // si n > 3, le programme s'arrête et affiche un message d'erreur
  assert(n <= 3);
  ...
}
```

1. Il y a une subtilité pour les tableaux alloués statiquement, mais globalement ça reste vrai.

Exercice XVI.1 p. 9

1. Écrire une fonction `array_get(int_array* t, int i)` qui renvoie l'élément d'indice `i` de `t`. Cette fonction vérifiera que l'accès est licite à l'aide d'une assertion.

```
int array_get(int_array* t, int i);
```

2. Écrire une fonction `array_set(int_array* t, int i, int x)` qui écrit `x` dans la case d'indice `i` de `t`. À nouveau, on utilisera une assertion pour vérifier la licéité de l'accès.

```
void array_set(int_array* t, int i, int x);
```

3. Écrire une fonction `array_delete(int_array* t)` qui libère tout le stockage associé à `t`.

```
void array_delete(int_array* t);
```

On supposera que `t` a été créé par un appel à `array_create`.

3 Tableaux dynamiques (ou vecteurs)

La structure abstraite de *tableau dynamique* est une extension de la structure abstraite de *tableau* : on peut toujours accéder facilement (et rapidement) à un élément quelconque par son indice, mais il est également possible d'ajouter ou de supprimer un élément. En règle générale, cette opération n'est possible qu'à l'extrémité droite du tableau[2].

Opération	Type	Effet
`get`	DynArray × `int` → Elt	Accès à un élément
`set`	DynArray × `int` × Elt → {}	Modification d'un élément
`push`	DynArray × Elt → {}	Ajout d'un élément à la fin du tableau
`pop`	DynArray → Elt	Récupération **et suppression** de l'élément le plus à droite
`create`	{} → DynArray	Création d'un tableau vide (fonction ne prenant pas d'argument)
`length`	DynArray → `int`	Nombre d'éléments présents

Figure XVI.1 – Signature d'un type DynArray impératif. On utilise {} pour désigner un type comme `void` ou `unit`.

Remarques

- Ce type abstrait est *très* utilisé en pratique, et de nombreux langages en fournissent une réalisation dans leur bibliothèque standard. C'est le cas de Python (type `list`), de Java (type `ArrayList`), de C++ (type `std::vector`)...
- Étrangement, la bibliothèque standard de OCaml ne propose pas de tableaux dynamiques (ça viendra sans doute un jour). Les deux bibliothèques tierces qui sont souvent utilisées en remplacement de la bibliothèque standard (`Batteries` et `Core`) proposent bien sûr cette structure de données.
- La bibliothèque standard du langage C ne propose pas non plus de tableaux dynamiques, ce qui n'est pas très surprenant puisqu'elle ne propose en fait aucune structure de données. Il existe bien évidemment d'innombrables implémentations tierces.
- On a donné une signature impérative, mais des versions fonctionnelles existent également (nous en rencontrerons cette année). Ces versions fonctionnelles sont cependant d'usage moins courant.

2. On peut concevoir des variantes permettant de le faire aux deux extrémités.

- Attention à ne pas confondre *tableau alloué dynamiquement* (ce qui signifie que les données sont sur le tas, et n'est pertinent, pour simplifier, qu'en C/C++) et *tableau dynamique* (qui est le type abstrait qui nous intéresse).

3.1 Réalisation naïve

On utilise la structure suivante :

```
struct int_dynarray {
  int len;
  int capacity;
  int* data;
};

typedef struct int_dynarray int_dynarray;
```

- L'entier `len` représente le nombre d'éléments *actuellement présents* dans le tableau.
- L'entier `capacity` représente la *capacité* du tableau, c'est-à-dire la taille du bloc vers lequel pointe `data`.
- Les éléments sont stockés à partir du début du bloc : il peut y avoir de la place libre à la fin du bloc (si `capacity > len`). Dans ce cas, les valeurs présentes dans les cases « libres » n'ont aucun sens.

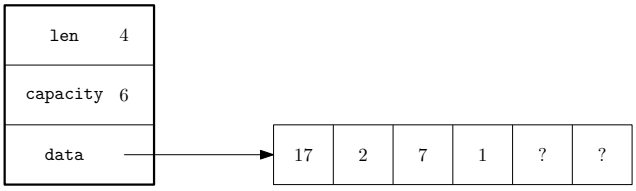

Figure XVI.2 – Un `int_dynarray_t` de capacité 6 contenant 4 éléments.

Les fonctions `get` et `set` fonctionnent exactement comme avant. Pour gérer les `pop` et les `push`, on peut imaginer procéder ainsi :

- un `pop` fait diminuer `len` de 1, et ne change pas `capacity` (il y a une erreur si `len` vaut zéro) ;
- pour un `push`, il y a deux cas :
  - si `len` $<$ `capacity`, on écrit le nouvel élément à droite et l'on incrémente `len` ;
  - si `len` $=$ `capacity`, on alloue un nouveau bloc `data`, de taille `capacity + 1`, on recopie l'ancien dans le nouveau et on y ajoute l'élément supplémentaire.

Exercice XVI.2 p. 9

1. Écrire une fonction `length(int_dynarray*)`.

```
int length(int_dynarray* t);
```

2. Écrire une fonction `make_empty(void)` renvoyant un pointeur vers un `int_dynarray` vide. On initialisera `data` à `NULL`.

```
int_dynarray* make_empty(void)
```

3. Modifier les fonctions `get` et `set` écrites plus haut pour qu'elles s'appliquent aux `int_dynarray`.

```
int get(int_dynarray* t, int i);
void set(int_dynarray* t, int i, int x);
```

4. Écrire la fonction `pop` (qui ne modifiera jamais la capacité du tableau) :

```
int pop(int_dynarray* t);
```

5. Écrire une fonction `resize(int_dynarray_t* t, int new_capacity)` qui alloue un nouveau bloc `data`, copie le contenu de l'ancien bloc dans le nouveau, met à jour les champs de `t` et libère l'ancien bloc.

```
void resize(int_dynarray* t, int new_capacity);
```

6. Écrire la fonction `push` :

```
void push(int_dynarray* t);
```

7. Écrire une fonction `delete(int_dynarray* t)` qui détruit le tableau dynamique pointé par `t` en libérant ce qu'il faut libérer.

```
void delete(int_dynarray* t);
```

Exercice XVI.3 – Analyse de la réalisation naïve p. 10

On considère une série de n opérations `push` successives sur un tableau dynamique initialement vide. Déterminer le coût total de ces opérations.
On considérera pour simplifier que les opérations `malloc` et `free` peuvent se faire en temps constant, mais cela ne change de toute façon pas le résultat ici.

3.2 Réalisation efficace

Le résultat de l'exercice XVI.3 montre que la réalisation naïve n'est pas satisfaisante : on souhaite que les opérations `push` et `pop` se fassent rapidement. Il n'est pas vraiment possible d'obtenir une complexité constante dans le pire des cas pour ces fonctions, mais on peut obtenir une complexité *amortie* en $O(1)$ pour `push` en utilisant la stratégie suivante :

- s'il reste de la place libre, on ajoute l'élément dans le tableau (comme pour la solution naïve) ;
- sinon, on procède aussi comme pour la solution naïve, sauf que le nouveau bloc alloué est de taille `2 * capacity`. Il faut ajouter un cas particulier si `capacity` est nul.

Exercice XVI.4 p. 11

Apporter les modifications nécessaires aux fonctions `push` et `resize` pour utiliser la nouvelle stratégie.

Exercice XVI.5 – Analyse amortie sans réduction de taille p. 11

1. Quelles sont les complexités des opérations `pop`, `get` et `set` ?
2. Si `t` est un tableau dynamique de longueur n, quelle est la complexité d'un `push` dans le pire cas ? dans le meilleur cas ?
3. On considère une série de n opérations `push` et `pop` sur un tableau initialement vide. Il n'y a aucune contrainte sur les opérations effectuées (sauf qu'on ne fait pas de `pop` sur un tableau vide) : on peut avoir n `push`, ou $n/2$ `push` suivis de $n/2$ `pop`, ou une alternance de `push` et de `pop`...
   Montrer que le coût total de cette série de n opérations est en $O(n)$.

Comme une série de n opérations (sur un tableau initialement vide) a un coût total en $O(n)$, on dira que la *complexité amortie* d'une opération `push` (ou `pop`) est en $O(1)$.

Cette complexité amortie est satisfaisante, mais notre stratégie a un gros défaut : la mémoire utilisée peut-être arbitrairement plus grande que celle nécessaire à stocker le nombre actuel d'éléments. En effet, la taille du bloc alloué ne diminue jamais lors d'une opération `pop`, et l'on peut donc avoir un tableau vide occupant une place proportionnelle au nombre maximum d'éléments qu'il a contenus par le passé.

Exercice XVI.6 p. 11

On propose la stratégie suivante :

- si `len` devient strictement inférieur à `capacity / 2` après un `pop`, on ré-alloue le bloc de données en lui donnant une taille `capacity / 2`;
- sinon, on procède comme avant.

Le strictement *inférieur garantit que l'on ne repasse jamais à une capacité de zéro, ce qui simplifie légèrement les choses.*

1. Apporter les modifications nécessaires à la fonction `pop`.
2. Montrer qu'une série de n opérations successives peut avoir un coût de l'ordre de n^2, et le mettre en évidence expérimentalement.

Exercice XVI.7 p. 12

Pour régler ce problème, on modifie légèrement la stratégie :

- si `len` devient strictement inférieur à `capacity / 4`, on ré-alloue un bloc de taille `capacity / 2`;
- sinon, on supprime l'élément sans ré-allouer.

En s'inspirant de la démonstration faite pour la réalisation d'une file fonctionnelle à l'aide de deux listes, montrer que la complexité amortie des opérations `pop` et `push` est en $O(1)$. On pourra prendre comme potentiel :

$$\Phi(t) = |2\mathtt{len}(t) - \mathtt{capacity}(t)|.$$

3.3 Opérations supplémentaires

Les opérations que nous avons définies jusqu'ici sont les seules opérations élémentaires sur un tableau dynamique.

Exercice XVI.8 p. 13

Dans cet exercice, on considère que les seuls moyens d'interagir avec un tableau dynamique sont les fonctions définies dans la signature donnée en figure XVI.1. Autrement dit, l'implémentation est « cachée » : on ne sait même pas que `int_dynarray` est défini comme une `struct`, et on ne connaît certainement pas les champs de cette `struct`.

1. Écrire une fonction permettant d'insérer un nouvel élément à un emplacement arbitraire `i` du tableau. Les éléments présents aux indices $j \geqslant i$ seront décalés d'une case vers la droite.

```
void insert_at(int_dynarray* t, int i, int x)
```

Remarque
Les valeurs acceptables pour `i` vont de 0 à la longueur du tableau incluse (dans ce cas, l'insertion revient à un `push`).

2. Déterminer la complexité de `insert_at` (en fonction de `i` et $\mathtt{len}(t)$).
3. Écrire une fonction permettant de supprimer un élément à un emplacement arbitraire, en récupérant sa valeur. Les éléments situés à droite seront décalés vers la gauche.

```
int extract_at(int_dynarray* t, int i)
```

Remarque
Les valeurs acceptables pour `i` vont de 0 à $n-1$ (où n est la longueur du tableau). Si $i = n-1$, l'opération équivaut à un `pop`.

4. Déterminer la complexité de cette fonction.

Exercice XVI.9 – Une variante du tri insertion

p. 13

On se propose d'écrire une variante du tri insertion sur les `int_dynarray`. Ce tri ne sera pas en place : on renverra un nouveau tableau (trié) sans modifier celui passé en paramètre. L'idée est la suivante, en notant `in` le tableau à trier :

- on crée un tableau vide `out` – tout au long de l'exécution de l'algorithme, ce tableau sera trié ;
- pour chaque élément de `in` :
  - on détermine à quelle position de `out` il faut l'insérer pour que `out` reste trié ;
  - on effectue l'insertion (à la position déterminée)
- on renvoie le tableau `out`.

1. Écrire une fonction `position(int_dynarray* t, int x)` qui renvoie le plus grand entier `i` tel que l'insertion de `x` en position `i` laisse le tableau `t` trié (en supposant qu'il était trié avant l'appel).

```c
int positon(int_dynarray* t, int x);
```

2. Écrire une fonction `insertion_sort(int_dynarray* t)` qui trie un tableau suivant l'algorithme décrit ci-dessus.

```c
int_dynarray* insertion_sort(int_dynarray* t);
```

3. Déterminer la complexité de cette fonction (dans le pire cas). On distinguera le nombre de comparaisons entre éléments du tableau effectuées du nombre d'opérations des autres opérations élémentaires.
4. Modifier la fonction `position` pour qu'elle effectue un nombre de comparaisons en $O(\log n)$ (où n est la longueur du tableau dans lequel on insère).
5. Peut-on imaginer une situation où cette amélioration est significative ?

Solutions

Correction de l'exercice XVI.1 page 4

Pour `array_delete`, il faut bien penser à libérer `t` (qui a été alloué sur le tas par `array_create`). Bien sûr, il faut libérer `t->data` **avant** de libérer `t` (sinon c'est un *use after free*).

```
int array_get(int_array* t, int i){
    assert(0 <= i && i < t->len);
    return t->data[i];
}

void array_set(int_array* t, int i, int x){
    int n = t->len;
    assert(0 <= i && i < t->len);
    t->data[i] = x;
}

void array_delete(int_array* t){
    free(t->data);
    free(t);
}
```

Correction de l'exercice XVI.2 page 5

Pas de grosse difficulté pour les premières fonctions ; la fonction `make_empty` est nettement plus simple que `array_create`.

```
int length(int_dynarray* t){
    return t->len;
}

int_dynarray* make_empty(void){
    int_dynarray* t = malloc(sizeof(int_dynarray));
    t->len = 0;
    t->capacity = 0;
    t->data = NULL;
    return t;
}

int get(int_dynarray* t, int i){
    assert(0 <= i && i < t->len);
    return t->data[i];
}

void set(int_dynarray* t, int i, int x){
    assert(0 <= i && i < t->len);
    t->data[i] = x;
}
```

Pour la fonction `pop`, il faut penser à mettre à jour la longueur :

```c
int pop_naif(int_dynarray* t){
    assert(t->len > 0);
    int x = t->data[t->len - 1];
    t->len--;
    return x;
}
```

C'est la fonction `resize` qui demande d'être attentif. Elle a plusieurs responsabilités :

- vérifier que la nouvelle capacité est suffisante (c'est optionnel, puisque cette fonction n'a pas vocation à être publique) ;
- allouer le nouveau stockage ;
- copier le tableau dans le nouveau stockage ;
- mettre a jour la capacité ;
- libérer l'ancien stockage.

```c
void resize(int_dynarray* t, int new_capacity){
    assert(t->len <= new_capacity);
    int* new_data = (int*)malloc(new_capacity * sizeof(int));
    for (int i = 0; i < t->len; i++){
        new_data[i] = t->data[i];
    }
    free(t->data);
    t->data = new_data;
    t->capacity = new_capacity;
}
```

La fonction `push` commence par redimensionner si c'est nécessaire, puis ajoute l'élément.

```c
void push_naif(int_dynarray* t, int x){
    if (t->len == t->capacity) {
        resize(t, 1 + t->capacity);
    }
    t->data[t->len] = x;
    t->len++;
}
```

La fonction `delete` est identique à celle écrite plus haut :

```c
void delete(int_dynarray* t){
    free(t->data);
    free(t);
}
```

Correction de l'exercice XVI.3 page 6

Une opération de redimensionnement a une complexité temporelle en $\Theta(k)$, où k est la taille du tableau actuel (puisqu'on néglige le coût du `malloc` et qu'il faut recopier les n éléments). Si l'on ne fait que des `push`, il faut redimensionner à chaque fois (avec notre stratégie actuelle). Le coût total est donc en :

$$\Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2)$$

Correction de l'exercice XVI.4 page 6

Pas de difficulté, le cas particulier à ne pas oublier est signalé par l'énoncé.

```
void push(int_dynarray* t, int x){
    if (t->len == t->capacity) {
        if (t->capacity == 0) {
            resize(t, 1);
        } else {
            resize(t, 2 * t->capacity);
        }
    }
    t->data[t->len] = x;
    t->len++;
}
```

Correction de l'exercice XVI.5 page 6

1. Ces trois opérations sont en temps constant.
2. Sans redimensonnement, un `push` s'effectue en temps constant (c'est le meilleur cas). S'il faut redimensionner, la complexité est en $\Theta(n)$ (n éléments à recopier).
3. On peut majorer brutalement pour commencer :
   - il y a au plus n `pop`, chacun en temps constant, donc un coût total en $O(n)$ pour ces opérations ;
   - il y a au plus n `push` ne causant pas de redimensionnement, de nouveau en $O(n)$ au total ;
   - les redimensionnements se font toujours vers une taille strictement plus grande, et le nombre d'éléments ne peut dépasser n. De plus, les redimensionnements n'ont lieu que quand on passe d'une taille 2^k à une taille 2^{k+1}. On peut donc majorer le coût total par :

$$\sum_{2^k \leqslant n} 2^k = \sum_{k=0}^{\lfloor \log_2 n \rfloor} 2^k = 2^{1+\lfloor \log_2 n \rfloor} - 1 = O(n)$$

   On obtient une complexité totale (pour l'ensemble des n opérations) en $O(n)$, et donc une complexité amortie de $O(1)$ par opération.

Correction de l'exercice XVI.6 page 7

1. Pas de difficulté :

```
int pop(int_dynarray* t){
    assert(t->len > 0);
    int x = t->data[t->len - 1];
    t->len--;
    if (2 * t->len < t->capacity) {
        resize(t, t->capacity / 2);
    }
    return x;
}
```

2. Le problème est que l'on peut avoir une série de redimensionnements quand on fait osciller la longueur du tableau autour d'une puissance de 2.
   Plus précisément, soit $n = 2^k + 1$. On fait :

- 2^{k-1} `push` successifs, ce qui donne $\texttt{len} = \texttt{capacity} = 2^{k-1}$;
- un `push` supplémentaire, ce qui donne $\texttt{len} = 2^{k-1}+1$ et $\texttt{capacity} = 2^k$;
- ensuite, une série de 2 `pop` suivis de 2 `push` (cette série sera de longueur 2^{k-3}).

À chaque fois, le deuxième `pop` nous fait passer à $\texttt{len} = 2^{k-1}-1$ et $\texttt{capacity} = 2^k$, donc il déclenche un redimensionnement. Ce redimensionnement se fait en $\Theta(2^k)$ et nous fait passer à $\texttt{len} = 2^{k-1}-1$ et $\texttt{capacity} = 2^{k-1}$. Après un `pop`, on a $\texttt{len} = \texttt{capacity}$, donc le deuxième déclenche un redimensionnement (temps $\Theta(2^k)$) et l'on se retrouve à $\texttt{len} = 2^{k-1}+1$ et $\texttt{capacity} = 2^k$, c'est-à-dire au point de départ.

Au total, il y aura donc 2^{k-2} redimensionnements, chacun en temps $\Theta(2^k)$: on obtient donc du $\Theta(2^{2k}) = \Theta(n^2)$.

Correction de l'exercice XVI.7 page 7

On considère $t_0, \dots, t_{n-1}$ les états successifs du tableau (t_0 est donc vide), et l'on note C_i le coût de l'opération qui fait passer de t_i à t_{i+1}. On prend toutes les constantes multiplicatives égales à 1 pour les complexités, ce qui est possible sans perte de généralité (on pourrait multiplier le potentiel par une constante), et l'on distingue les cas suivant la nature de la i-ème opération :

- Si c'est un `push` sans redimensionnement, alors $\Phi(t_{i+1}) = |2 + 2\texttt{len}(t_i) - \texttt{capacity}(t_i)| \leqslant 2 + \Phi(t_i)$ et $C_i = 1$. On a donc :
$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leqslant 3$$
- Si c'est un `push` avec redimensionnement, on a nécessairement $\Phi(t_i) = |2\texttt{len}(t_i) - \texttt{len}(t_i)| = \texttt{len}(t_i)$ et $\Phi(t_{i+1}) = 0$. De plus, $C_i = \texttt{len}(t_i) + 1$ (copie des éléments actuels et ajout du nouvel élément). Donc :
$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leqslant 1$$
- Si c'est un `pop` sans redimensionnement, alors $\Phi(t_{i+1}) = |2\texttt{len}(t_i) - 2 - \texttt{capacity}(t_i)| \leqslant 2 + \Phi(t_{i+1})$ et $C_i = 1$, donc :
$$C_i + \Phi(t_{i+1}) - \Phi(t_i) \leqslant 3$$
- Si c'est un `pop` avec redimensionnement, alors nécessairement $\texttt{capacity}(t_i) = 4\texttt{len}(t_i)$ d'où $\Phi(t_i) = 2\texttt{len}(t_i)$. D'autre part, $\texttt{len}(t_{i+1}) = \texttt{len}(t_i) - 1$ et $\texttt{capacity}(t_{i+1}) = \texttt{capacity}(t_i)/2 = 2\texttt{len}(t_i)$. On obtient $\Phi(t_{i+1}) = \texttt{len}(t_i) + 1$. Comme $C_i = \texttt{len}(t_i)$, le résultat final est :
$$C_i + \Phi(t_{i+1}) - \Phi(t_i) = \texttt{len}(t_i) + \texttt{len}(t_i) + 1 - 2\texttt{len}(t_i) = 1$$

On en déduit
$$\sum_{i=0}^{n-1} (C_i + \Phi(t_{i+1}) - \Phi(t_i)) \leqslant 3n.$$

Or, d'autre part, on a :
$$\sum_{i=0}^{n-1} (C_i + \Phi(t_{i+1}) - \Phi(t_i)) = \sum_{i=0}^{n-1} C_i + \underbrace{\Phi(t_n)}_{\geqslant 0} - \underbrace{\Phi_{t_0}}_{=0} \geqslant \sum_{i=0}^{n-1} C_i$$

On en déduit donc :
$$\sum_{i=0}^{n-1} C_i \leqslant 3n = O(n)$$

On a donc bien une complexité amortie en $O(1)$ pour `pop` et `push`.

Correction de l'exercice XVI.8 page 7

1. On respecte bien la consigne, en ne regardant pas sous le capot de `t`. Il faut donc commencer par faire un `push` (avec une valeur quelconque, qui sera de toute façon écrasée), puis décaler les éléments pour faire une place et enfin insérer l'élément.

```
void insert_at(int_dynarray* t, int i, int x){
    int n = length(t);
    assert(0 <= i && i <= n);
    push(t, x);
    for (int j = n; j > i; j--){
        set(t, j, get(t, j - 1));
    }
    set(t, i, x);
}
```

   L'assertion au début de la fonction n'est en fait pas strictement nécessaire :
   - si $i < 0$, le dernier passage dans la boucle se fait pour $j = i+1 \leqslant 0$, donc le `get(t, j - 1)` va forcément lever une assertion (ce qui termine l'exécution du programme) ;
   - si $i > n$, le dernier `set(t, i, x)` va lever une assertion, puisque `length(t)` vaut n à cet instant.
2. En notant n la longueur de `t`, on fait un nombre d'opérations proportionnel à $n - i$, et chacune de ces opérations est en temps constant. La complexité est donc en $O(n - i)$.
3. Le principe est similaire : on récupère la valeur à renvoyer, on décale tout vers la gauche, en on finit par un `pop` (dont on ignore la valeur de retour) pour mettre à jour la taille du tableau.

```
int pop_at(int_dynarray* t, int i){
    int n = length(t);
    int x = get(t, i);
    for (int j = i; j < n - 1; j++){
        set(t, j, get(t, j + 1));
    }
    pop(t);
    return x;
}
```

4. Même idée : la complexité est en $O(n - i)$.

Correction de l'exercice XVI.9 page 8

1. La fonction est marginalement plus simple à écrire en partant de la gauche du tableau, mais partir de la droite permet d'obtenir un tri s'exécutant en temps linéaire si le tableau est déjà trié (ou presque).

```
int position_linear(int_dynarray* t, int x){
    int i = length(t) - 1;
    while (i >= 0 && get(t, i) > x){
        i--;
    }
    return i + 1;
}
```

2. Pas de difficulté.

```
int_dynarray* insertion_sort(int_dynarray* t){
    int_dynarray* out = make_empty();
    int n = length(t);
    for (int i = 0; i < n; i++){
        int x = get(t, i);
        int pos = position_linear(out, x);
        insert_at(out, pos, x);
    }
    return out;
}
```

3. Les deux fonctions s'exécutent d'autant plus lentement que l'élément doit être inséré vers la gauche du tableau. Si le tableau initial est trié par ordre (strictement) décroissant, toutes les insertions se feront au début, ce qui est donc le pire cas. Il y aura alors $i-1$ comparaisons pour trouver l'emplacement du i-ème élément, puis $\Theta(i)$ opérations pour l'insérer. Au total, on a dans le pire cas $\Theta(n^2)$ comparaisons et $\Theta(n^2)$ opérations de lecture/écriture dans le tableau.
4. On peut chercher la position d'insertion de manière dichotomique : la fonction ci-dessous a exactement la même spécification que `position_linear` mais ne fait que $O(\log(\texttt{length}(t)))$ comparaisons (et opérations).

```
int position(int_dynarray* t, int x){
    int start = 0;
    int end = length(t);
    // Invariant :
    //   - les éléments d'indice >= end sont > x
    //   - les éléments d'indice < start sont <= x
    while (end > start){
        int mid = start + (end - start) / 2;
        int xmid = get(t, mid);
        if (x >= xmid){
            start = mid + 1;
        } else {
            end = mid;
        }
    }
    return start;
}
```

5. Avec cette fonction, le tri insertion effectue $O(n \log n)$ comparaisons et $O(n^2)$ opérations élémentaires sur les tableaux. Dans certains cas, le coût d'une comparaison peut être nettement supérieur à celui d'une opération élémentaire sur les tableaux : par exemple, si l'on trie un tableau de pointeurs vers des objets compliqués, pour lesquels la comparaison est très coûteuse. La nouvelle version peut alors avoir un comportement qui ressemble à du $O(n \log n)$ même pour des valeurs de n assez grandes (10 000 par exemple).