TP 2 : Tableaux dynamiques

1 Un tableau qui connait sa taille

Comme nous l'avons vu, un tableau ne connaît pas sa taille 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. Cela arrive 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 = malloc(sizeof(int_array));
    // Et le stockage pour les données
    int *data = malloc(len * sizeof(int));
    for (int i = 0; i < len; i++) {
        data[i] = x;
    }
    t->len = len;
    t->data = data;
    return t;
}
```

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. 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`.

2 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.

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 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. 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.
- ⇨ 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.

2.1 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.

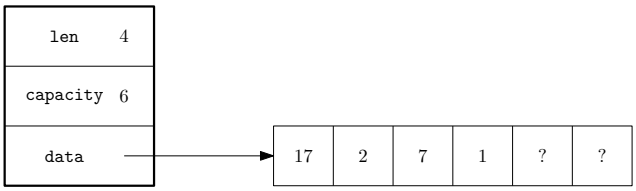

FIGURE 2 – Un int dynarray de capacité 6 contenant 4 éléments.

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.

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, int x);
```

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);
```

8. 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.

2.2 Réalisation efficace

Le résultat de la section précédente 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 $\mathrm{O}(1)$ 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.

1. Apporter les modifications nécessaires aux fonctions `push` et `resize` pour utiliser la nouvelle stratégie.
2. Quelles sont les complexités des opérations `pop`, `get` et `set` ?
3. Si t est un tableau dynamique de longueur n, quelle est la complexité d'un `push` dans le pire cas ? dans le meilleur cas ?

On souhaite montrer que la complexité amortie des fonctions `push` et `pop` est en $\mathrm{O}(1)$, c'est-à-dire que si on part d'un tableau dynamique vide, la complexité de n appels successifs à l'une de ces fonctions est en $\mathrm{O}(n)$. Nous allons utiliser pour cela la méthode du potentiel. On définit le potentiel Φ sur l'ensemble des tableaux dynamiques par :

$$\Phi(t) \coloneqq |4\,\mathrm{length}(t) - 2\,\mathrm{capacity}(t)|\,.$$

On note $t_0, t_1, \ldots, t_n$ les états successifs du tableau dynamique. L'état t_0 est son état initial, et pour tout $k \in [\![1, n]\!]$, t_k est l'état du tableau après la k-ième opération :

$$t_0 \to t_1 \to t_2 \to \cdots \to t_{n-1} \to t_n.$$

Pour tout $k \in [\![1, n]\!]$, on note C_k la complexité de l'opération qui fait passer la pile de l'état t_{k-1} à l'état t_k. Dans cette question, les seules opérations prises en compte sont les accès en lecture et en écriture à un élément d'un tableau.

4. Montrer qu'il existe $K \geqslant 0$ tel que

$$\forall k \in [\![1, n]\!]\,, \quad C_k + [\Phi(t_k) - \Phi(t_{k-1})] \leqslant K.$$

5. En déduire que

$$\sum_{k=1}^{n} C_k \leqslant Kn$$

et conclure.

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

2.3 Opérations supplémentaires

Les opérations que nous avons définies jusqu'ici sont les seules opérations élémentaires sur un tableau dynamique. 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 **??**. Autrement dit, l'implémentation est « cachée » : on ne sait même pas que `int_dynarray` est défini comme une `struct`, et on ne connaît 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)
```

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

2. Déterminer la complexité de `insert_at` en fonction de i et $\mathrm{len}(t)$.
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)
```

   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.

2.4 Une variante du tri insertion

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.

```
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.

```
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 ?

2.5 Réduction de taille

La complexité amortie des opérations `push` et `pop` 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. 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.

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.

3. Montrer que la complexité amortie des opérations `pop` et `push` est en $O(1)$.