TP n°12 – Tableaux redimensionnables

d'après un TP de Nicolas Pécheux

1 Tableaux redimensionnables

Dans ce TP on veut implémenter une version d'un **tableau redimensionnable** dans le style des "listes" Python. Contrairement à la version vue en cours, on ne peut pas redimensionner le tableau à la taille désirée, mais on peut ajouter (append) un élément à droite du tableau, en affectant plus de place si on en manque.

On nommera le type correspondant $resizearray$. Ce type doit avoir les primitives suivantes, on note elt le type contenu dans notre tableau redimensionnable :

- Création d'un tableau vide avec **`create`** : $void \to resizearray$
- Accès à l'élément d'indice i avec **`get`** : $resizearray \times int \to elt$
- Modification de l'élément d'indice i avec **`set`** : $resizearray \times int \times elt \to void$
- Ajout d'un élément à la fin du tableau avec **`push`** : $resizearray \times elt \to void$
- Récupération et suppression de l'élément le plus à droite avec **`pop`** : $resizearray \to elt$
- Compter le nombre d'éléments présents avec **`length`** : $resizearray \to int$
- Supprimer la structure de données en mémoire avec **`delete`** : $resizearray \to void$

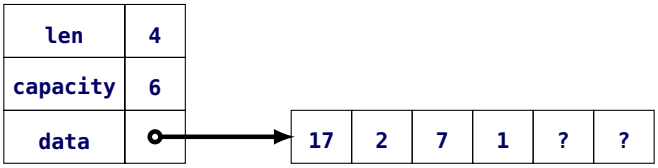
2 Réalisation naïve en C

En C on ne sait pas créer des types polymorphes. On créera donc un tableau redimensionable d'entiers.

On considère la structure suivante :

```c
struct resizearray {
  int capacity;
  int len;
  int* data;
  };
typedef struct resizearray resizearray;
```

• L'entier **`capacity`** représente la capacité du tableau, c'est-à-dire la taille du bloc vers lequel pointe data.
• L'entier **`len`** représente le nombre de cases du tableau utilisées pour stocker des valeurs.
• 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.



Exemple - Un **`resizearray`** de capacité 6 contenant 4 éléments.

Les fonctions **`get`** et **`set`** fonctionnent exactement comme pour un tableau. Pour gérer **`pop`** et **`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 dans la première case libre et on incrémente **`len`** ;
  - si **`len == capacity`**, on alloue un nouveau bloc de mémoire **`data`**, de taille **`capacity + 1`**, on recopie le contenu de l'ancien champ **`data`** et on ajoute la nouvelle valeur dans la case vide.

- **Q1.** Dessiner le tableau redimensionnable de taille 3 et de capacité 3 contenant 2, 12, 5, dans cet ordre. Dessiner son état après une opération **`pop`**. Dessiner son état lorsqu'on ajoute 4, puis son état si on ajoute 6.
- **Q2.** Écrire la fonction `int length(resizearray* t)` qui donne la longueur du tableau redimensionnable.
- **Q3.** Écrire la fonction `resizearray* create(void)` renvoyant un pointeur vers un tableau redimensionnable vide. On initialisera **`data`** à **`NULL`** et **`capacity`** à 0.
- **Q4.** Écrire les fonctions **`get`** et **`set`**. N'oubliez pas la programmation défensive (des assert). Leur signature sera :

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

- **Q5.** Écrire la fonction `int pop(resizearray* t)`.

- **Q6.** Écrire une fonction `void resize(resizearray* t, int new_capacity)` qui alloue un nouveau tableau, copie le contenu de l'ancien tableau dans le nouveau et met à jour les champs de t. On pensera à libérer l'ancien tableau. **La fonction resize est une fonction outil qui peut être réutilisée dans la partie 3. Il ne s'agit pas d'une primitive.**
- **Q7.** Écrire la fonction `void push(resizearray* t, int x)` en utilisant `resize`.
- **Q8.** Écrire une fonction `void delete(resizearray* t)` qui détruit le tableau redimensionnable pointé par t en libérant tout l'espace occupé par celui-ci.

3 Réalisation efficace en C

Avec l'implémentation précédente, la complexité de chaque **push** est en grand O du champ **capacity** (on réalloue à chaque fois un bloc car on a pas assez de place), sauf si on a précédemment utilisé **pop**. Sachant qu'on part d'une capacité nulle, pour avoir un tableau assez grand pour une utilisation lambda (écrire un algorithme de tri, ...) on va avoir beaucoup de créations de tableaux et donc de complexité.

Il n'est pas vraiment possible d'obtenir une complexité constante dans le pire des cas pour les fonctions **push** et **pop**, 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** et pas **capacity+1**. Attention au cas **capacity==0**.

- **Q9.** Apporter les modifications nécessaires à la fonction fonction **push** pour utiliser la nouvelle stratégie.
- **Q10.** Quelles sont les complexités des opérations **pop**, **get** et **set** ?
- **Q11.** Si **t** est un tableau redimensionable de longueur n, quelle est la complexité d'un **push** dans le pire cas et dans le meilleur cas ?
- **Q12.** On considère une série de n opérations **push** ou **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**, etc. 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** 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 pour 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 inférieure 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.

- **Q13.** Apporter les modifications nécessaires à la fonction **pop**.
- **Q14.** Montrer qu'une série de n opérations successives peut avoir un coût de l'ordre de n^2.

Pour régler ce problème, on modifie légèrement la stratégie :

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

On peut alors montrer que la complexité amortie des opérations **pop** et **push** est en $O(1)$.

4 Opérations supplémentaires

Jusqu'ici on a défini les opérations élémentaires sur les tableaux redimensionnables. On va désormais écrire des fonctions moins élémentaires, en se plaçant de l'autre côté de la barrière d'abstraction : **il est strictement interdit de manipuler les tableaux redimensionnables avec autres choses que les primitives définies jusqu'ici**.

4.1 Insertion et suppression à une position arbitraire

- **Q15.** 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.
  Les valeurs acceptables pour i vont de 0 à la longueur du tableau incluse (dans ce cas, l'insertion revient à un **push**). Sa signature sera :

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

- **Q16.** Déterminer la complexité de `insert_at` (en fonction de i et du champ **`len`** de t).
- **Q17.** Écrire une fonction permettant de supprimer un élément à un emplacement arbitraire i et renvoyer sa valeur. Les éléments situés à droite seront décalés vers la gauche. 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`**. La signature sera :

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

- **Q18.** Déterminer la complexité de cette fonction.

4.2 Une variante du tri insertion

On se propose d'écrire une variante du tri insertion sur les `resizearray`. 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`**.

- **Q19.** Écrire une fonction **`int`** `position(resizearray* 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).
- **Q20.** Écrire une fonction `resizearray* insertion_sort(resizearray* tableau t)` suivant l'algorithme décrit ci-dessus.